

# Proyecto Estructuras de Datos

Maria Paula Rodríguez Ruiz and Gabriel Riaño Beltrán and Samuel Montoya  
Piedrahíta

<sup>1</sup> Pontificia Universidad Javeriana, Bogotá D.C, Colombia  
riañogabriel@javeriana.edu.co  
rodriguezr.mp@javeriana.edu.co  
samuel-montoya@javeriana.edu.co

**Abstract.** This project focuses on the implementation of a three-dimensional polygonal mesh manipulation and analysis system using data structures in C++. A polygonal mesh, in the field of computer graphics, is a fundamental structure composed of vertices, edges and faces that in whole allows the digital representation of the shape of 3D objects. The goal is that the developed system allows these meshes to be managed through textual commands in an interactive console, providing key functionalities such as loading meshes from text files, generating bounding boxes, finding nearest vertices and calculating shortest paths between vertices.

For better distribution the project is going to be divided into three main components. The first one is responsible for the basic organization and manipulation of meshes, allowing 3D objects to be loaded, listed and deleted in memory. The second component focuses on identifying nearest vertices within objects, while the third addresses the problem of finding shortest paths between specific points in the mesh.

This system is essential for different applications in the fields of simulation, video games, animation, etc. The project not only will provide a solid foundation for manipulating polygonal meshes, but also is going to offer an intuitive user interface to perform complex operations easily and efficiently.

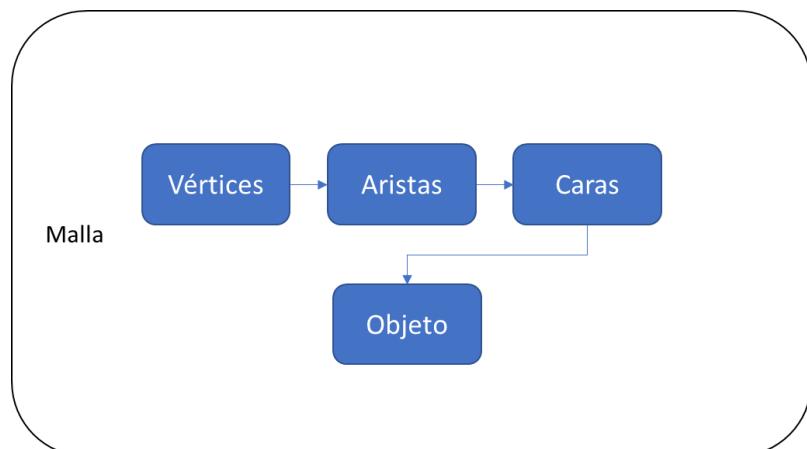
**Keywords:** Polygonal mesh, Data structures, Euclidian distance, C++.

## 1 Descripción del problema

La manipulación y análisis de mallas poligonales tridimensionales es fundamental en la computación gráfica y el modelado 3D, es de hecho según (Tiigimägi, S., 2024) la forma más antigua de representación geométrica utilizada en los gráficos por ordenador para crear objetos en el espacio 3D, ya sea un videojuego, un producto 3D o un personaje de dibujos animados que se este modelando, todo empieza a partir de una malla. Por eso todos los programas de modelado 3D más populares, como Maya, 3d Max y Blender, te proporcionan herramientas para crear, texturizar, renderizar y animar mallas poligonales en 3D.

No obstante, también se menciona que aunque encuentra su aplicación a través de diversas técnicas, no es la solución definitiva puesto que todavía hay objetos que no se pueden crear con las representaciones de las mallas, por ejemplo no puede abarcar las superficies curvas y la mayoría de objetos orgánicos en general.

Ahora bien, (3DCadPortal., 2024) ahonda más respecto a las mallas y nos indica que se fundamentan en vértices, ejes(aristas) y caras como se evidencia en la Figura 1. Las caras generalmente se caracterizan por una forma de triángulo o cuadriláteros, y la mayoría de los programas modeladores 3D las utilizan para visualizar o renderizar una superficie, por lo que a mayor número de polígonos mayor será la precisión, un ejemplo de esto es la Figura 2 que parece ser un cuadrado doblado que sigue el proceso mencionado previamente: pasa de vértices a aristas, de estas a caras, y de estas a la malla final, donde es evidente que de haber más caras la imagen se vería un poco más nítida y menos cuadriculada.



*Figura 1. Diagrama representativo de una malla poligonal. Elaboración propia*



Figura 2. Ejemplo de una malla poligonal. (PNGWing., 2024)

En otras palabras, una malla poligonal es un compuesto de vértices (coordenadas X, Y y Z), aristas (líneas que conectan los vértices) y caras (algo así como muchos planos 2D donde sus bordes son un conjunto de vértices determinados).

Ya conociendo la importancia que estas tienen, parece evidente decir que un sistema que pueda manipularlas y hacer cálculos sobre ellas sería una posible solución útil para poner en práctica, y es por eso que planteamos el presente proyecto para el que tenemos muy presente que aunque los datos a almacenar sean “sencillos” a primera vista, la organización de estos datos es vital para poder realmente definir y manipular las mallas poligonales.

## 2 Componente 1: Organización de la Información

Este componente tiene como fin la organización y gestión de la información de las mallas poligonales, de este componente se establecen los pilares sobre los cuales se construirán todas las operaciones posteriores del sistema.

### Objetivos de los comandos:

1. **Cargar Archivo de Malla:** Este comando permite al sistema leer y cargar la información de un objeto 3D desde un archivo de texto que está estructurado. Es fundamental para inicializar la memoria con los datos de la malla, o que habilita las operaciones posteriores sobre el objeto.
2. **Listar Objetos en Memoria:** Permite obtener una visión general de los objetos cargados en el momento en la memoria, evidenciando la cantidad de vértices, aristas y caras de cada malla. Es fundamental para verificar el almacenamiento adecuado de los datos.

3. **Generar Caja Envolvente:** Calcula una caja envolvente que contiene al objeto 3D, esta facilita la manipulación y análisis espacial del objeto. Se añade como un objeto nuevo en la memoria, brindando herramientas adicionales.
4. **Generar Caja Envolvente Global:** Calcula una caja envolvente que contiene a todos los objetos 3D incluyendo las cajas, por lo que es útil para saber que espacio cubico ocupan todos los objetos 3D. Esta caja no será agregada a la colección de mallas del sistema para distinguirla de mejor manera. E
5. **Descargar Objetos.** Elimina de la memoria los datos de un objeto 3D que ya no se quiera utilizar, por lo que es vital para la correcta organización y almacenamiento de la información.
6. **Guardar Malla en Archivo:** Se encarga de guardar la información de un objeto 3D en un archivo de texto (.txt) lo que es útil en términos de persistencia de la información.
7. **Salir:** Termina la ejecución de la aplicación. Es fundamental para finalizar la ejecución de la aplicación de forma que no consuma más recursos.

## 2.1 Descripción de los Comandos

### 1. Cargar Archivo de Malla:

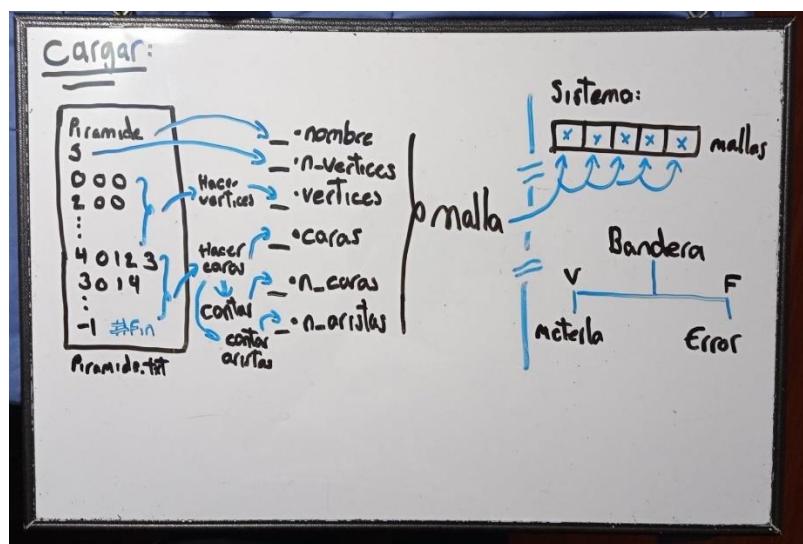


Figura 3. Diseño del comando cargar. Elaboración propia

En la Figura 3, podemos observar la idea que se plantea para realizar la acción requerida por el comando *cargar*.

**Descripción:** En primer lugar, se debe utilizar el nombre del archivo que ingresa el usuario para poder abrirlo y leer su contenido que contiene la malla a cargar. Con la información almacenada en el archivo se debe construir una malla, la cual una vez en el sistema, se debe verificar que no exista aun dentro de la colección. En caso de que ya exista la malla no se debe agregar a la colección y se le debe advertir al usuario. Por otra parte, si el archivo contiene información que no es apta para la lectura de mallas, se advertirá al usuario.

**Requisitos previos:**

- Archivo de Entrada: El archivo debe existir y ser accesible desde el programa. Debe estar en el formato adecuado para que la función leerArchivo pueda interpretarlo.
- Comparación entre Mallas: El TAD de Malla debe soportar la comparación entre objetos.
- Método leerArchivo: Esta función debe estar implementada para leer archivos y devolver una instancia válida de Malla.

**Proceso interno de la función correspondiente a este comando:**

- 1) Lectura del Archivo:
  - i. Se llama a leerArchivo(nomarch), que intenta leer el archivo especificado por nomarch y devolver una malla.
  - ii. Si leerArchivo() no puede leer correctamente el archivo, devolverá un objeto Malla no válido, o sea vacío.
- 2) Verificación de Duplicados:
  - i. La función recorre el contenedor mallas para comprobar si ya existe un objeto igual al que se está intentando cargar.
  - ii. Si encuentra una coincidencia, se activa una bandera y se muestra un mensaje indicando que el objeto ya está cargado, y termina el proceso.
- 3) Verificación de la Malla Válida:
  - i. La función compara m con un objeto Malla vacío.
  - ii. Si el objeto leído es igual a una Malla vacía, indica que la carga del archivo no fue exitosa (el archivo estaba vacío o en un formato no valido), y se activa una bandera.
- 4) Carga del Objeto:
  - i. Si la bandera no ha sido activada después de las verificaciones anteriores, significa que el archivo era válido, el objeto no estaba duplicado, y la malla no estaba vacía.
  - ii. En este caso, se añade el objeto al contenedor de mallas, y se muestra un mensaje de éxito en la consola.

**Resultados e Interpretación:**

- (Archivo vacío o incompleto): Si el objeto leído es igual a una malla vacía, se deduce que el archivo no contenía un objeto 3D válido. En

este caso, la función no carga el objeto y puede mostrar un mensaje indicando el problema.

- (Archivo no existe o ilegible): Si leerArchivo no puede abrir o leer el archivo correctamente, no se puede crear una malla válida, y el comportamiento sería similar al de un archivo vacío o incompleto.
- (Objeto ya existe): Si la función detecta que un objeto idéntico ya ha sido cargado en memoria, evita cargarlo nuevamente y muestra un mensaje informando al usuario.
- (Resultado exitoso): Si todas las condiciones son satisfactorias (archivo válido, malla única y no vacía), la función carga el objeto y muestra un mensaje confirmando el éxito de la operación.

#### Verificación de Ejecución Correcta:

- Mensajes en Consola: La forma más directa de verificar que la función se ha ejecutado correctamente es revisar el mensaje mostrado en la consola. Cada posible escenario tiene un mensaje específico que indica el resultado de la operación.
- Estado del contenedor mallas: Despues de la ejecución de la función, el contenedor de mallas debe contener el nuevo objeto si la carga fue exitosa.

#### 2. Listar Objetos en Memoria:

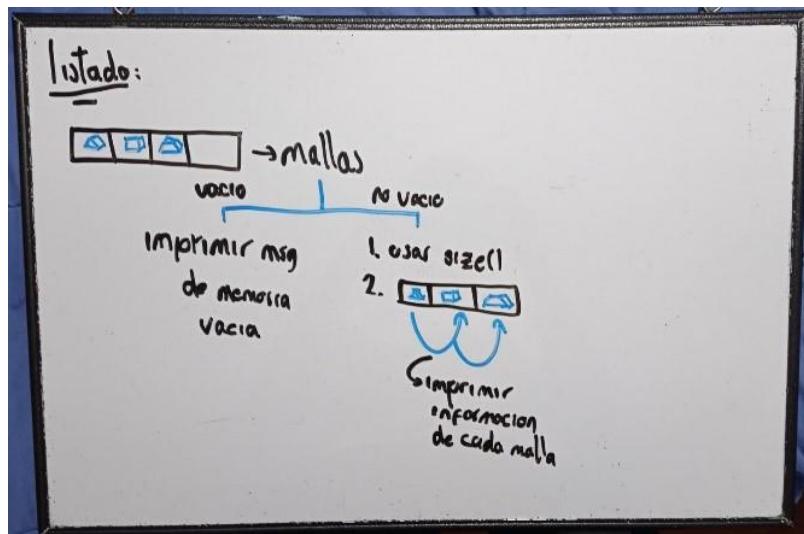


Figura 4. Diseño del comando listado. Elaboración propia

En la Figura 4, podemos observar la idea que se plantea para realizar la acción requerida por el comando *listado*.

**Descripción:** Este comando se implementa en dos partes, en la malla y en el sistema. En primer lugar, el sistema verifica si hay mallas cargadas en la memoria y en caso de no tener imprime en pantalla la advertencia. Por otra parte, sí hay mallas en memoria se itera sobre la colección de mallas e interiormente de cada malla se realizará un proceso que muestre el nombre del objeto y cantidad de vértices, aristas y caras. Así, se visualizan todos los objetos cargados en memoria.

#### **Requisitos previos:**

- Objetos Cargados: Debe haber objetos cargados en el vector mallas para que la función tenga algo que listar.
- Método info(): Cada malla debe tener implementado un método info() que proporcione la información necesaria para el listado.

#### **Proceso interno de la función correspondiente a este comando:**

- 1) Verificación de Memoria Vacía:
  - i. La función primero verifica si el contenedor de mallas está vacío.
  - ii. Si está vacío, se muestra el mensaje "Ningún objeto ha sido cargado en memoria." y la función termina.
- 2) Listado de Objetos en Memoria:
  - i. Si mallas no está vacío, la función verifica si env\_global es una malla vacía.
  - ii. Sin Objeto Global:
- 3) Si env\_global es una malla vacía, se cuenta y se lista únicamente el número de objetos en mallas.
- 4) Muestra un mensaje indicando la cantidad de objetos en memoria y luego recorre el contenedor de mallas imprimiendo su información.
  - i. Con Objeto Global:
- 5) Si env\_global no es una malla vacía, se incluye este en la cuenta total de objetos en memoria.
- 6) Muestra un mensaje indicando la cantidad total de objetos (sumando una) y luego recorre mallas, llamando el método de información para cada objeto.
- 7) Finalmente, se muestra la información de env\_global para incluirlo en el listado.

#### **Resultados e Interpretación:**

- (Memoria Vacía): Si no hay objetos en mallas, se muestra el mensaje "Ningún objeto ha sido cargado en memoria." Esto indica que no hay datos almacenados en memoria para listar.
- (Resultado Exitoso): Si hay objetos en mallas, se muestra un mensaje con el número de objetos cargados en memoria seguido de un listado detallado:
  - Para cada objeto, se muestra la información básica, como el nombre del objeto, la cantidad de vértices, aristas, y caras.

- Si env\_global está presente, también se incluye en el listado, aumentando el total de objetos en 1.

#### Verificación de Ejecución Correcta:

- Mensajes en Consola: Se revisa que los mensajes que se muestran en la consola para determinar si la función ha sido ejecutada correctamente. Estos mensajes dirán si la memoria está vacía o cuántos objetos están presentes.
- Salida Detallada: Se debe asegurar que la información de cada objeto (nombres, cantidad de vértices, aristas, y caras) sea correcta y corresponda con los datos esperados.
- Verificación de env\_global: Si se espera que env\_global esté presente en la memoria, se verifica que su información también esté incluida en el listado.

### 3. Generar Caja Envolvente:

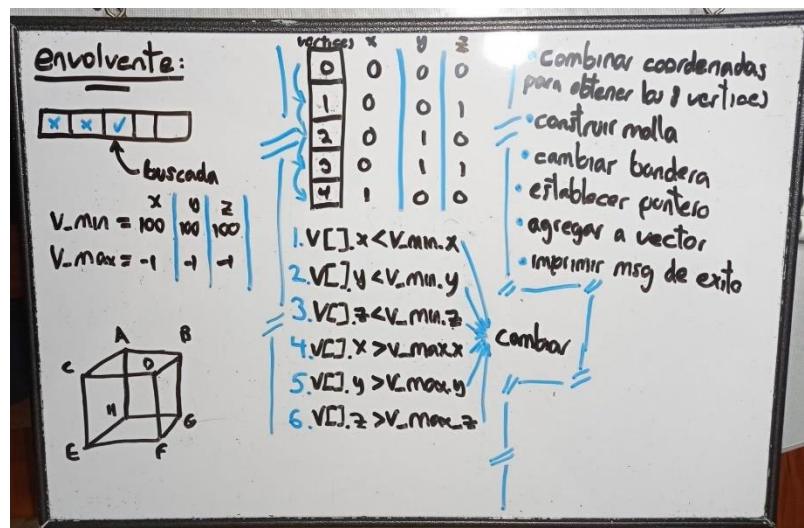


Figura 5. Diseño del comando envolvente objeto. Elaboración propia

En la Figura 5, podemos observar la idea que se plantea para realizar la acción requerida por el comando *envolvente objeto*.

**Descripción:** Primero, esta función busca la malla correspondiente al nombre que ingrese el usuario, si no la encuentra se imprime por pantalla el debido mensaje de error. Pero en caso de encontrar la malla, se calcula su caja envolvente de la siguiente forma: se comparan todos los vértices que componen la malla en términos de coordenadas. Por medio de un ciclo se construyen dos vértices que contienen los valores máximos y mínimos en cada eje del plano 3D. Con estos se realiza una combinación para obtener 8 vértices que compondrán una malla nueva que representara la caja que envuelve la

malla original. Creada la caja, se cambia el indicador que informa si la malla es una caja envolvente y adicionalmente almacenar la información de la caja dentro de la malla que ingreso el usuario, de forma que se pueda relacionar una malla con su caja.

**Requisitos previos:**

- Objeto en Memoria: El objeto 3D que se va a envolver debe estar cargado en memoria y disponible para su búsqueda.
- Método buscarMalla: Este método debe ser capaz de encontrar un objeto por su nombre en la lista de objetos en memoria.

**Proceso interno de la función correspondiente a este comando:**

- 1) Búsqueda del Objeto en Memoria:
  - i. La función comienza llamando a buscarMalla(nombre), que busca en la memoria un objeto 3D con el nombre especificado.
  - ii. Si buscarMalla() no encuentra el objeto, devuelve una malla vacía, y la función muestra el mensaje "El objeto nombre\_objeto no ha sido cargado en memoria." y termina la ejecución.
- 2) Inicialización de Puntos Extremos:
  - i. Si el objeto es encontrado, se inicializan dos vértices: pmin y pmax.
  - ii. pmin se inicializa con valores máximos para X, Y, y Z, mientras que pmax se inicializa con los valores mínimos posibles.
- 3) Cálculo de pmin y pmax:
  - i. La función recorre todos los vértices del objeto para calcular pmin y pmax.
  - ii. Para cada vértice, se actualizan los valores de pmin y pmax comparando las coordenadas X, Y, y Z del vértice actual con las de pmin y pmax.
- 4) Creación de la Caja Envolvente:
  - i. Una vez determinados pmin y pmax, se definen los 8 vértices de la caja envolvente, que corresponden a las esquinas del cubo.
  - ii. Se crea una nueva malla llamada mallaEnvolvente con estos vértices y 6 caras, cada una representando una cara del cubo.
- 5) Almacenamiento en Memoria:
  - i. El nuevo objeto mallaEnvolvente se agrega al contenedor de mallas.
  - ii. Además, se actualiza el objeto original para que contenga una referencia al objeto mallaEnvolvente.
- 6) Salida en Consola:

- i. Finalmente, la función muestra un mensaje indicando que la caja envolvente ha sido generada y agregada a los objetos en memoria con el nombre env\_nombre\_objeto.

#### Resultados e Interpretación:

- (Objeto no existe): Si el objeto nombre\_objeto no está cargado en memoria, se muestra el mensaje "El objeto nombre\_objeto no ha sido cargado en memoria." Esto indica que no se pudo calcular la caja envolvente porque el objeto no está disponible.
- (Resultado exitoso): Si la caja envolvente se calcula con éxito, se muestra el mensaje "La caja envolvente del objeto nombre\_objeto se ha generado con el nombre env\_nombre\_objeto y se ha agregado a los objetos en memoria." Esto confirma que la operación fue exitosa y que la caja envolvente ha sido agregada correctamente.

#### Verificación de Ejecución Correcta:

- Mensajes en Consola: Revisa el mensaje que se muestra en la consola después de la ejecución de la función para confirmar si la operación fue exitosa o si el objeto no fue encontrado.
- Estado del Contenedor Mallas: Se verifica que el contenedor mallas contenga el nuevo objeto env\_nombre\_objeto, lo cual confirma que la caja envolvente ha sido agregada a la memoria.

#### 4. Generar Caja Envolvente Global:

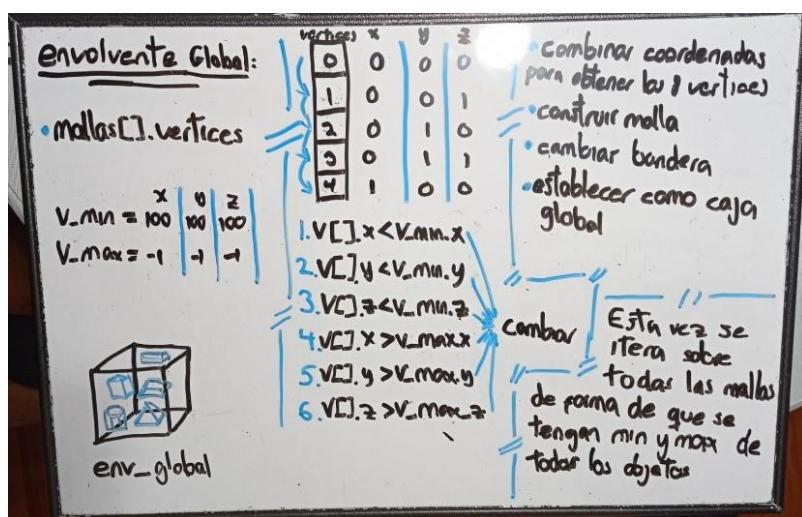


Figura 6. Diseño del comando envolvente. Elaboración propia

En la Figura 6, podemos observar la idea que se plantea para realizar la acción requerida por el comando *envolvente*.

**Descripción:** Este comando realiza un proceso prácticamente igual al del comando descrito anteriormente, en cuanto a que maneja el mismo método para sacar las coordenadas máximas y mínimas de todos los ejes. La diferencia radica en que en esta función itera sobre todas las mallas del sistema en lugar de solo una. Es de resaltar que en nuestro diseño se plantea que la caja global no se agrega directamente a la colección de mallas, sino que esta es un atributo adicional que pertenece al sistema. No obstante, para el llamado del comando de listado se imprime la información de esta caja simulando que hace parte de la colección.

**Requisitos previos:**

- Objetos en Memoria: Debe haber objetos 3D cargados en memoria para que la función pueda calcular la caja envolvente global.
- Contenedor de Mallas: Debe estar correctamente inicializado y contener objetos Malla válidos para que la función pueda recorrerlos y calcular los valores extremos de pmin y pmax.

**Proceso interno de la función correspondiente a este comando:**

- 1) Verificación de Objetos en Memoria:
  - i. La función comienza verificando si el contendor de mallas está vacío.
  - ii. Si no hay objetos en memoria, se muestra el mensaje "Ningun objeto ha sido cargado en memoria." y la función termina.
- 2) Inicialización de Puntos Extremos:
  - i. Si hay objetos en memoria, se inicializan dos vértices: pmin y pmax.
  - ii. pmin se inicializa con valores máximos posibles para X, Y, y Z, mientras que pmax se inicializa con los valores mínimos posibles.
- 3) Cálculo de pmin y pmax:
  - i. La función recorre cada objeto en la colección de objetos cargados.
  - ii. Para cada objeto, se obtiene la lista de vértices y se calcula pmin y pmax, actualizándolos según sea necesario para abarcar todos los vértices de todos los objetos.
- 4) Creación de la Caja Envolvente Global:
  - i. Con pmin y pmax determinados, se definen los 8 vértices de la caja envolvente global, que corresponden a las esquinas del cubo.
  - ii. Se crean las caras del cubo utilizando estos vértices.
- 5) Almacenamiento en Memoria:

- i. Se crea un nuevo objeto Malla llamado mallaEnvolvente con estos vértices y 6 caras, cada una representando una cara del cubo.
  - ii. Este nuevo objeto se almacena en la variable env\_global del sistema, que representa la caja envolvente global.
- 6) Salida en Consola:
- i. Finalmente, la función muestra un mensaje indicando que la caja envolvente global ha sido generada y agregada a los objetos en memoria con el nombre env\_global.

**Resultados e Interpretación:**

- (Memoria vacía): Si no hay objetos cargados en memoria, se muestra el mensaje "Ningun objeto ha sido cargado en memoria." Esto indica que no se pudo calcular la caja envolvente porque no hay objetos en memoria.
- (Resultado exitoso): Si la caja envolvente se calcula con éxito, se muestra el mensaje "La caja envolvente de los objetos en memoria se ha generado con el nombre env\_global y se ha agregado a los objetos en memoria." Esto confirma que la operación fue exitosa y que la caja envolvente global ha sido correctamente calculada y almacenada.

**Verificación de Ejecución Correcta:**

- Mensajes en Consola: Se revisa el mensaje mostrado en la consola para confirmar si la operación fue exitosa o si no había objetos en memoria.
- Estado de env\_global: Se verifica que la variable env\_global contenga un objeto Malla válido, lo cual confirma que la caja envolvente global ha sido calculada y almacenada correctamente.

**5. Descargar Objetos:**

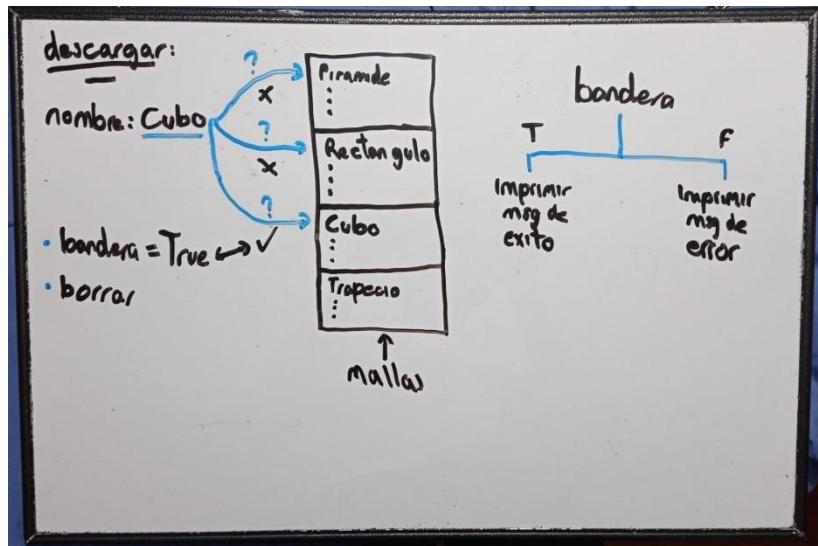


Figura 7. Diseño del comando descargar. Elaboración propia

En la Figura 7, podemos observar la idea que se plantea para realizar la acción requerida por el comando *descargar*.

**Descripción:** Para este comando se debe recibir como parámetro el nombre de la malla. Primero se utilizará una bandera booleana con el fin de verificar si se eliminó el objeto. Esta iniciar como falso. Posteriormente se recorre la colección de mallas, si el nombre de la malla coincide con el buscado se elimina el objeto y se cambia la bandera a verdadero. Por último, dependiendo del valor de la bandera de imprime por pantalla el mensaje correspondiente a la eliminación o si no está debidamente cargado en memoria.

#### Requisitos previos:

- Objeto en Memoria: Debe haber objetos cargados en memoria para que la función tenga algún efecto
- Colección de Mallas: Debe estar correctamente inicializado y contener objetos Malla válidos, permitiendo la iteración y eliminación de elementos.

#### Proceso interno de la función correspondiente a este comando:

- 1) Inicialización:
  - i. Se define una bandera como falso, la cual servirá para indicar si el objeto ha sido encontrado y eliminado.
- 2) Búsqueda y Eliminación del Objeto:
  - i. La función entra en un bucle, recorriendo cada objeto en la colección de mallas.

- ii. Para cada objeto, compara su nombre con el nombre dado.
  - iii. Si encuentra un objeto cuyo nombre coincide, lo elimina del vector mallas usando erase, actualiza la bandera a verdadero, y sale del bucle.
- 3) Salida en Consola:
- i. Caso 1: Objeto Encontrado y Eliminado: Si la bandera es verdadera, se muestra el mensaje "El objeto nombre ha sido eliminado de la memoria de trabajo."
  - ii. Caso 2: Objeto No Encontrado: Si la bandera es falsa, se muestra el mensaje "El objeto nombre no ha sido cargado en memoria."

**Resultados e Interpretación:**

- (Objeto no existe): Si el objeto identificado por nombre\_objeto no está cargado en memoria, la función mostrará el mensaje "El objeto nombre\_objeto no ha sido cargado en memoria." Esto indica que no se realizó ninguna eliminación porque el objeto no se encontraba en memoria.
- (Resultado exitoso): Si el objeto es encontrado y eliminado exitosamente, se mostrará el mensaje "El objeto nombre\_objeto ha sido eliminado de la memoria de trabajo." Esto confirma que la operación se realizó correctamente y que el objeto ya no está presente en la memoria del sistema.

**Verificación de Ejecución Correcta:**

- Mensajes en Consola: Se revisa el mensaje que se muestra en la consola tras la ejecución de la función para verificar si la eliminación fue exitosa o si el objeto no existía en memoria.
- Estado del Contenedor de Mallas: Para confirmar que el objeto fue eliminado, se puede inspeccionar el contenedor y comprobar que ya no contiene un objeto con el nombre nombre\_objeto.

**6. Guardar Malla en Archivo:**

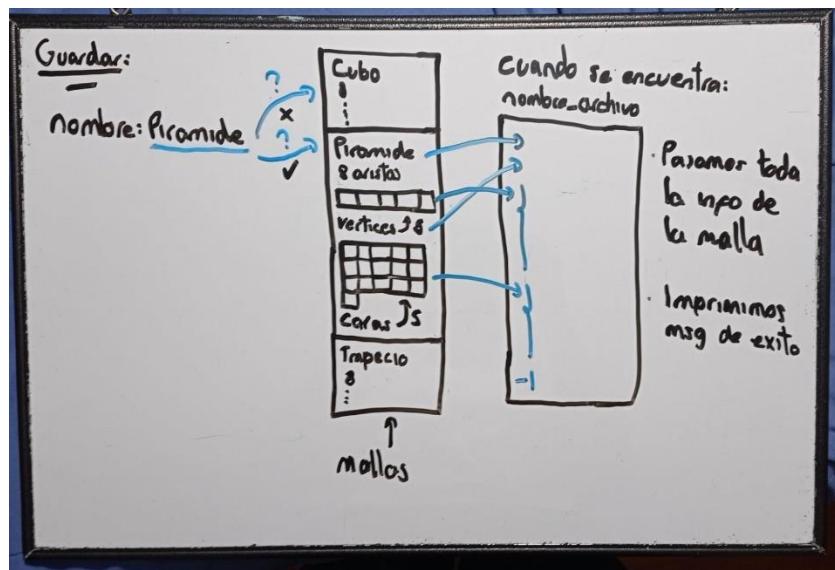


Figura 8. Diseño del comando guardar. Elaboración propia

En la Figura 8, podemos observar la idea que se plantea para realizar la acción requerida por el comando *guardar*.

**Descripción:** Este comando como primera acción busca la malla requerida utilizando el nombre que llega por parámetro. En caso de no encontrar la malla requerida se imprimirá el mensaje de error, pero sí se encuentra la malla, se creará un archivo de texto el cual almacenará la información de esta. La información que se escribe en el archivo será: el nombre del objeto, número de vértices, coordenadas de los vértices y las caras. Por último, se escribe un “-1” en el archivo indicando la finalización de la escritura e imprime el mensaje de confirmación de guardado.

#### Requisitos previos:

- Objeto en Memoria: Debe existir un objeto cargado en memoria con el nombre *nombre\_objeto* para poder ser guardado.
- Acceso al Sistema de Archivos: El sistema debe tener permisos para escribir en el archivo *nomarch*, pues si no se tienen permisos, la función no podrá guardar los datos.

#### Proceso interno de la función correspondiente a este comando:

##### 1) Búsqueda del Objeto:

- i. La función comienza buscando el objeto identificado por nombre en la memoria utilizando la función *buscarMalla()*.

- ii. Si el objeto no se encuentra, se muestra un mensaje de error indicando que el objeto no ha sido cargado en memoria y la función termina.
- 2) Apertura del Archivo:
- i. Si el objeto es encontrado, la función intenta abrir un archivo con el nombre nomarch usando un flujo de salida de datos.
- 3) Escritura de la Información del Objeto en el Archivo:
- i. Nombre del Objeto: Se escribe el nombre del objeto en la primera línea del archivo.
  - ii. Número de Vértices: En la siguiente línea, se escribe el número total de vértices del objeto.
  - iii. Coordenadas de los Vértices: Luego, la función itera sobre la colección de vértices del objeto, escribiendo las coordenadas (x, y, z) de cada vértice en líneas separadas.
  - iv. Información de las Caras: Posteriormente, la función itera sobre la colección de caras del objeto. Para cada cara, se escribe el número de vértices que la componen seguido de los índices de esos vértices. Cada cara se guarda en una línea separada.
  - v. Indicador de Fin: Finalmente, se escribe un -1 en una línea para indicar el fin de los datos.
- 4) Cierre del Archivo:
- i. Despues de escribir toda la información, se cierra el archivo para asegurar que los datos se guarden correctamente.
- 5) Mensaje de Confirmación:
- i. Si el proceso se completó exitosamente, se muestra un mensaje en la consola indicando que la información del objeto ha sido guardada exitosamente en el archivo nomarch.

#### **Resultados e Interpretación:**

- (Objeto no existe): Si el objeto identificado por nombre\_objeto no está cargado en memoria, se mostrará el mensaje "El objeto nombre\_objeto no ha sido cargado en memoria." Esto significa que no se realizó ninguna acción de guardado porque el objeto no se encontró en la memoria del sistema.
- (Resultado exitoso): Si la operación de guardado es exitosa, se mostrará el mensaje "La información del objeto nombre\_objeto ha sido guardada exitosamente en el archivo nombre\_archivo." Esto confirma que la información del objeto se guardó correctamente en el archivo especificado.

#### **Verificación de Ejecución Correcta:**

- Contenido del Archivo: Se puede revisar el archivo nomarch para asegurarse de que contiene la información correcta en el formato esperado. El archivo debe comenzar con el nombre del objeto, seguido

del número de vértices, las coordenadas de cada vértice, y la información de las caras, terminando con un -1.

- Mensaje en Consola: El mensaje en consola debe indicar claramente si el proceso fue exitoso o si el objeto no fue encontrado en memoria.

## 7. Salir:

**Descripción:** El comando salir está diseñado para terminar la ejecución del programa. Su función es simple: cuando el usuario ingresa el comando "salir", el programa finaliza inmediatamente.

### Proceso de Ejecución:

- 1) Detección del Comando salir:
  - i. Cuando el usuario ingresa un comando, el programa lo almacena en una colección de cadenas de texto.
  - ii. La primera palabra del comando es comparada con "salir", que corresponde a uno de los elementos de la colección de comandos.
- 2) Finalización del Programa:
  - i. Si el comando ingresado coincide con "salir", el programa ejecuta la función exit(1);.
  - ii. La función exit(1); inmediatamente termina la ejecución del programa.
  - iii. No se ejecutan más instrucciones después de esta llamada, y el programa se detiene por completo.

### Resultados e Interpretación:

- No tiene salida en pantalla:
  - El comando salir no produce ninguna salida visible en la pantalla. Simplemente cierra la aplicación.

### Verificación de Ejecución Correcta:

- Proceso Cerrado: Se puede verificar en el administrador de tareas (en Windows) o usando comandos como ps (en Linux) para confirmar que el proceso del programa ya no está en ejecución.

## 2.2 Diseño

A continuación, se muestra un bosquejo del diseño inicial (Figura 9) que se planteó para modelar el programa para el primer componente:

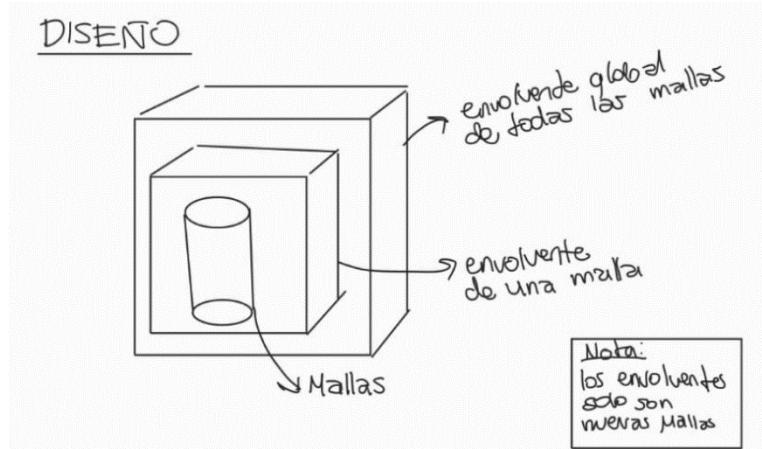


Figura 9. Diseño inicial. Elaboración propia

Con esta base planteada se procedió al diseño de TAD's para la correcta ejecución de los comandos.

**Descripción de los Tipos Abstractos de Datos (TADs):**

A continuación, se presenta la especificación de los tres TAD's creados para la realización de este proyecto, empezando con la la especificación del TAD de Sistema en la Figura 10:

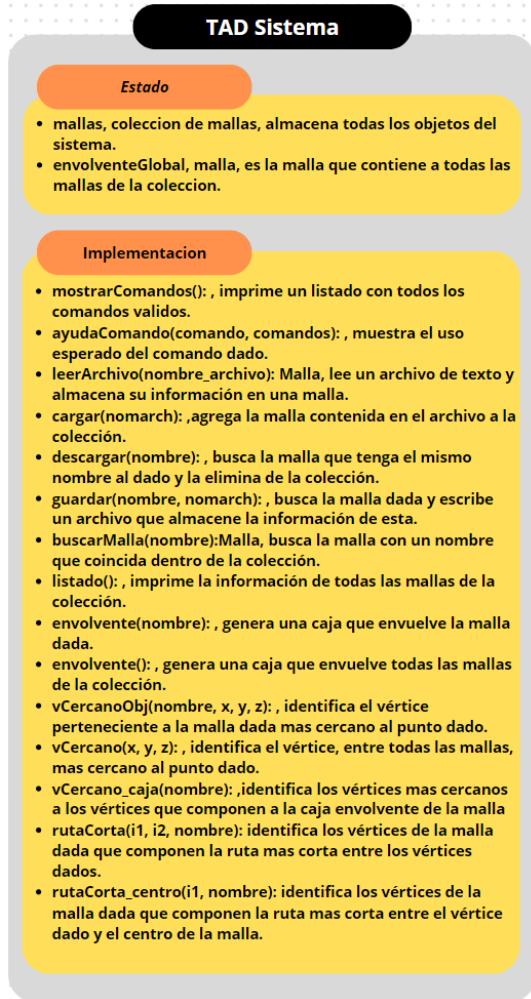


Figura 10. Especificación de sistema. Elaboración propia

Para el sistema se plantea que contenga dos atributos: **mallas**, siendo una colección de mallas; y **envolventeGlobal** que contendría una caja que envuelva a todos los objetos de la colección. En cuanto a su implementación, se tendrán 15 funciones que facilitarán el cumplimiento de las acciones requeridas por los comandos. Ahora se mostrará la especificación del TAD de malla en la Figura 11:

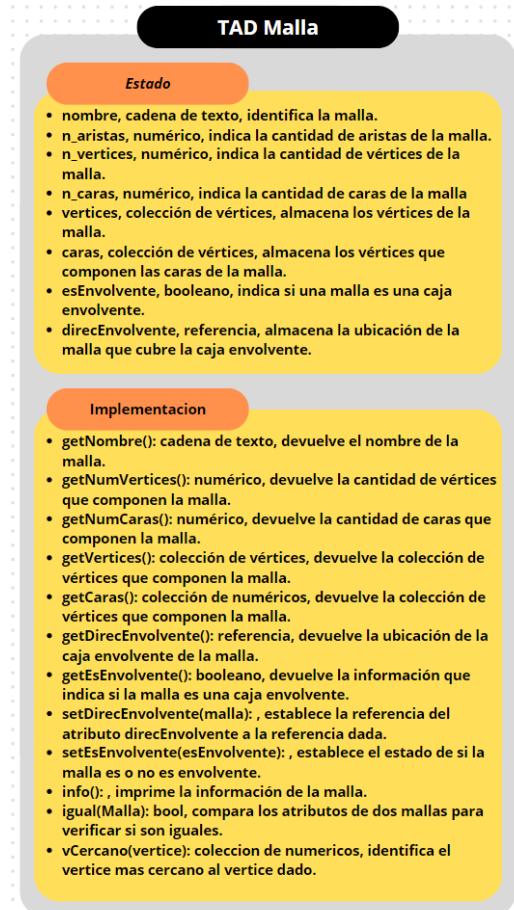
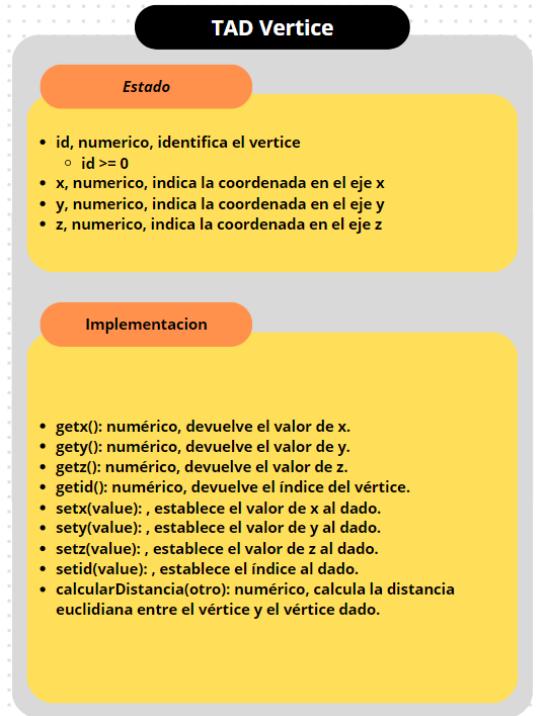


Figura 11. Especificación de malla. Elaboración propia

Para este TAD se considera la información que se recolecta de los archivos, de forma que cuando se lea un archivo de texto se pueda construir de la mejor forma una malla. La malla cuenta con 12 funciones que son llamadas por el sistema para ejecutar los comandos. Por último, se muestra el TAD de vértice en la Figura 12:



*Figura 12. Especificación de vértice. Elaboración propia*

Para este TAD se tiene un id y las coordenadas (x,y,z) que representan un punto del plano 3D. En cuanto a la implementación se tienen los getters y setters de sus atributos y una función adicional que calcula la distancia euclíadiana entre dos vértices, esta función será utilizada en otro componente por lo cual será ignorada momentáneamente.

#### **Diagrama de Relación de TADs:**

A continuación, se muestra el diagrama de relación entre los TAD's utilizados por el programa:

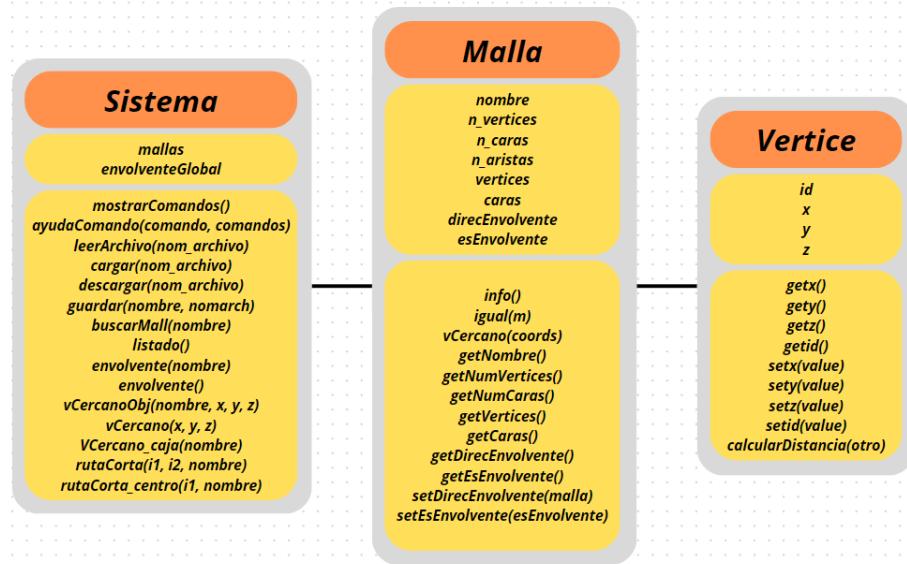


Figura 13. Diagrama de relación de TAD's. Elaboración propia

Es de resaltar que el main del programa actúa como un intermediario entre las interacciones realizadas por el usuario y el TAD de sistema, pues cuando se ingresa un comando, el main invoca un método del sistema que cumple con lo requerido por el usuario. Ya en el sistema, se gestiona la información de las mallas cargadas por medio de métodos propios del TAD de sistema y por métodos que forman parte del TAD de Malla. Un ejemplo que demuestra la integración entre los TAD's es cuando se usa el comando de “envolvente”, pues requiere el uso del TAD de vértice, que facilita las operaciones para crear las mallas que representan las cajas envolventes.

### 2.3 Plan de Pruebas

#### Planificación de Pruebas

Previo a la codificación, fue realizado un proceso de análisis respecto al funcionamiento del comando *envolvente* para de esa forma crear una tabla que presentara el nombre de la prueba, el resultado esperado y los datos de entrada, la justificación de esta y como se va a ejecutar; lo anterior, de forma que fuera posible evaluar la mayor cantidad de casos de prueba posibles que demostrarían la funcionalidad y calidad del comando, que a su vez aboga por el componente mismo.

Como bien se mencionó en la descripción de los comandos, el comando *envolvente* calcula una caja envolvente que contiene a todos los objetos 3D incluyendo las cajas; ahora bien, en el espacio tridimensional en el que nos encontramos un objeto 3D puede

ubicarse en cualquiera de los 8 octantes formados a partir de una combinación diferente de signos que se ve así:

1. Primer octante:  $(x > 0), (y > 0), (z > 0)$
2. Segundo octante:  $(x < 0), (y > 0), (z > 0)$
3. Tercer octante:  $(x < 0), (y < 0), (z > 0)$
4. Cuarto octante:  $(x > 0), (y < 0), (z > 0)$
5. Quinto octante:  $(x > 0), (y > 0), (z < 0)$
6. Sexto octante:  $(x < 0), (y > 0), (z < 0)$
7. Séptimo octante:  $(x < 0), (y < 0), (z < 0)$
8. Octavo octante:  $(x > 0), (y < 0), (z < 0)$

Por lo que, nos pareció preciso que los casos de prueba consistieran en comprobar que el cálculo de este envolvente global funcionara sin importar los octantes en los que se encontraran los objetos 3D o incluso si se encontraban en medio de ellas, lo que se ve reflejado en la tabla 1.

La estrategia que tenemos la intención de utilizar es: primero, crear 5 archivos de texto (representante cada uno de un objeto 3D) en el que 4 de ellos estén en los primeros cuatro octantes mientras que el último este en medio del primer y segundo octante; donde, para evaluar si efectivamente se forma el objeto deseado se grafiquen sus puntos y segmentos. Después, crear otros 5 archivos de texto (representante cada uno de un objeto 3D) en el que todos sean el espejo sobre el eje X e Y de un objeto 3D de los 5 anteriores, de forma que tengamos 5 objetos espejo donde, para nuevamente para evaluar si efectivamente se forma el objeto deseado se grafiquen sus puntos y segmentos. De esta forma, tendríamos un total de 10 archivos, que nos permitirían no solo probar la efectividad del comando como mencionamos en el párrafo anterior, sino también añadir infinidad de nuevas pruebas de las cuales añadiremos 4: calcular la caja envolvente de un objeto 3D específico y ver si la incluye en la caja envolvente global, calcular una caja envolvente global para todos los objetos 3D en los primeros cuatro octantes y/o para los últimos cuatro y calcular una caja envolvente global sin objetos 3D en memoria; logrando así un total de 15 casos de prueba.

**Tabla 1**  
*Plan de pruebas*

Nombre de la prueba	Resultado esperado y datos de entrada	Justificación	Cómo se ejecutará
envolvente con un objeto 3D en el primer octante	envolvente global asignado y calculado	Evaluar la funcionalidad y calidad del comando con un objeto	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema

	->Archivo de texto del objeto 3D en el primer octante	3D en el primer octante	3 Ejecutar los siguientes comandos:  \$ cargar objOct1.txt \$ envolvente
envolvente con un objeto 3D en el segundo octante	envolvente global asignado y calculado  ->Archivo de texto del objeto 3D en el segundo octante	Evaluar la funcionalidad y calidad del comando con un objeto 3D en el segundo octante	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar los siguientes comandos:  \$ cargar objOct2.txt \$ envolvente
envolvente con un objeto 3D en el tercer octante	envolvente global asignado y calculado  ->Archivo de texto del objeto 3D en el tercer octante	Evaluar la funcionalidad y calidad del comando con un objeto 3D en el tercer octante	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar los siguientes comandos:  \$ cargar objOct3.txt \$ envolvente
envolvente con un objeto 3D en el cuarto octante	envolvente global asignado y calculado  ->Archivo de texto del objeto 3D en el cuarto octante	Evaluar la funcionalidad y calidad del comando con un objeto 3D en el cuarto octante	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar los siguientes comandos:  \$ cargar objOct4.txt \$ envolvente
envolvente con un objeto 3D en el quinto octante	envolvente global asignado y calculado  ->Archivo de texto del objeto 3D en el quinto octante	Evaluar la funcionalidad y calidad del comando con un objeto 3D en el quinto octante	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar los siguientes comandos:  \$ cargar objOct5.txt \$ envolvente

envolvente con un objeto 3D en el sexto octante	envolvente global asignado y calculado  ->Archivo de texto del objeto 3D en el sexto octante	Evaluar la funcionalidad y calidad del comando con un objeto 3D en el sexto octante	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar los siguientes comandos:  \$ cargar objOct6.txt \$ envolvente
envolvente con un objeto 3D en el séptimo octante	envolvente global asignado y calculado  ->Archivo de texto del objeto 3D en el séptimo octante	Evaluar la funcionalidad y calidad del comando con un objeto 3D en el séptimo octante	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar los siguientes comandos:  \$ cargar objOct7.txt \$ envolvente
envolvente con un objeto 3D en el octavo octante	envolvente global asignado y calculado  ->Archivo de texto del objeto 3D en el octavo octante	Evaluar la funcionalidad y calidad del comando con un objeto 3D en el octavo octante	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar los siguientes comandos:  \$ cargar objOct8.txt \$ envolvente
envolvente con un objeto 3D en el primer y segundo octante	envolvente global asignado y calculado  ->Archivo de texto del objeto 3D en el primer y segundo octante	Evaluar la funcionalidad y calidad del comando con un objeto 3D entre el primer y segundo octante	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar los siguientes comandos:  \$ cargar objOct1y2.txt \$ envolvente
envolvente con un objeto 3D espejo en el primer y segundo octante	envolvente global asignado y calculado  ->Archivo de texto del objeto 3D	Evaluar la funcionalidad y calidad del comando con un objeto 3D espejo entre el primer	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar los siguientes comandos:

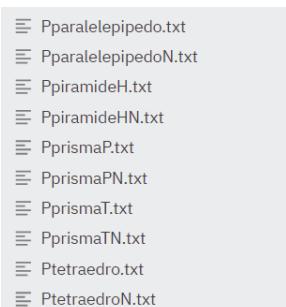
	espejo en el primer y segundo octante	y segundo octante	\$ cargar objOct1y2E.txt \$ envolvente
envolvente de un objeto 3D específico y ver si la incluye en el envolvente global	envolvente global asignado y calculado = envolvente específico  ->Archivo de texto del objeto 3D	Evaluar la funcionalidad y calidad del comando con las cajas específicas de los objetos 3D	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar los siguientes comandos:  \$ cargar obj1.txt \$ envolvente nombreObj1 \$ envolvente
envolvente para todos los objetos 3D en los primeros cuatro octantes	envolvente global asignado y calculado  ->Archivo de texto de los objetos 3D del primer al cuarto octante	Evaluar la funcionalidad y calidad del comando con objetos 3D entre el primer y cuarto octante	1 Agregar los archivos de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar los siguientes comandos:  \$ cargar objOct1.txt \$ cargar objOct2.txt \$ cargar objOct3.txt \$ cargar objOct4.txt \$ envolvente
envolvente para todos los objetos 3D en los últimos cuatro octantes	envolvente global asignado y calculado  ->Archivo de texto de los objetos 3D del cuarto al octavo octante	Evaluar la funcionalidad y calidad del comando con objetos 3D entre el cuarto y octavo octante	1 Agregar los archivos de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar los siguientes comandos:  \$ cargar objOct5.txt \$ cargar objOct6.txt \$ cargar objOct7.txt \$ cargar objOct8.txt \$ envolvente
envolvente para todos los objetos 3D en los ocho octantes	envolvente global asignado y calculado  ->Archivo de texto de los objetos 3D entre el	Evaluar la funcionalidad y calidad del comando con objetos 3D entre el	1 Agregar los archivos de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema

	de todos los octantes	todos los octantes	3 Ejecutar los siguientes comandos:  \$ cargar objOct1.txt \$ cargar objOct2.txt \$ cargar objOct3.txt \$ cargar objOct4.txt \$ cargar objOct5.txt \$ cargar objOct6.txt \$ cargar objOct7.txt \$ cargar objOct8.txt \$ cargar objOct1y2.txt \$ cargar objOct1y2E.txt \$ envolvente
envolvente sin objetos 3D en memoria	Visualización del mensaje de error correspondiente	Evaluar la funcionalidad y calidad del comando si no hay objetos 3D	1 Compilar y ejecutar el sistema 2 Ejecutar el siguiente comando:  \$ envolvente

*Nota: La tabla permite ver la planeación de casos de prueba para evaluar el comando envolvente, elaboración propia.*

### Ejecución de Pruebas

Siguiendo la estrategia presentada en la planeación de pruebas se hizo la respectiva creación de archivos de forma que quedaron los archivos presentados en la Figura 14, de los cuales en la Figura 15 se presenta un ejemplo comparativo de dos archivos con objetos 3D que son espejos, la Figura 16 muestra la gráfica de los primeros 5 objetos 3D y la Figura 17 muestra la gráfica de los últimos 5 objetos 3D, ambas graficas realizadas con ayuda de la graficadora 3D de Geogebra.



*Figura 14. Creación de los 10 archivos. Elaboración propia*

	Pparalelepipedo.txt	Pparalelepipedo_Negativo.txt
1	Paralelepipedo	Paralelepipedo_Negativo
2	8	8
3	-6 2 2	-6 2 -2
4	-4 2 2	-4 2 -2
5	-3 4 2	-3 4 -2
6	-5 4 2	-5 4 -2
7	-7 2 4	-7 2 -4
8	-5 2 4	-5 2 -4
9	-4 4 4	-4 4 -4
10	-6 4 4	-6 4 -4
11	4 0 1 2 3	4 0 1 2 3
12	4 4 5 6 7	4 4 5 6 7
13	4 0 1 5 4	4 0 1 5 4
14	4 1 2 6 5	4 1 2 6 5
15	4 2 3 7 6	4 2 3 7 6
16	4 3 0 4 7	4 3 0 4 7
17	-1	-1
18		

Figura 15. Comparación archivos 2 objetos espejo. Elaboración propia

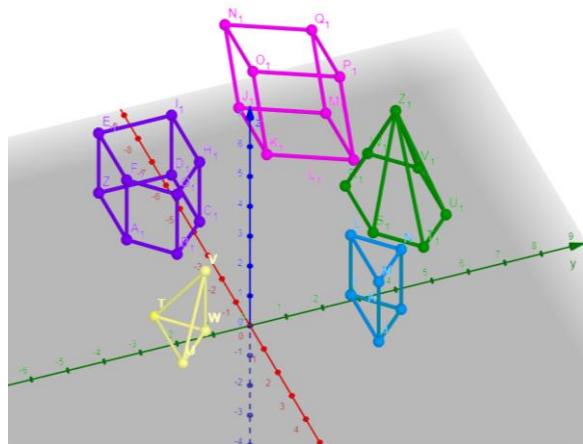


Figura 16. Representación de las primeras 5 mallas en Geogebra. Elaboración propia

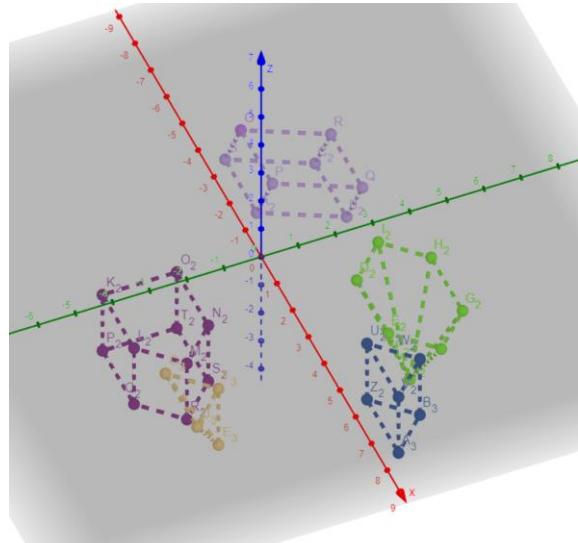


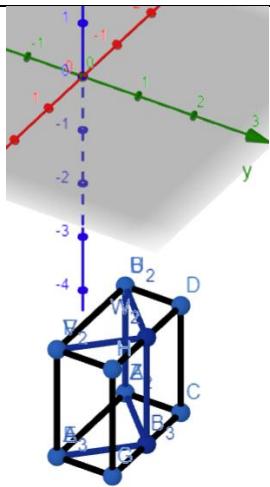
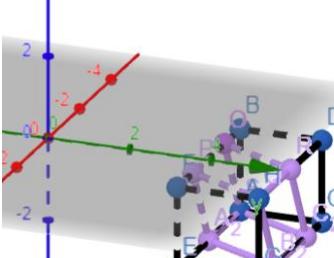
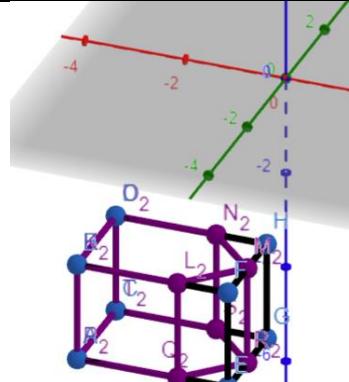
Figura 17. Representación de las segundas 5 mallas en Geogebra. Elaboración propia

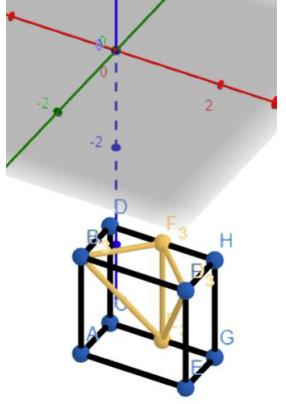
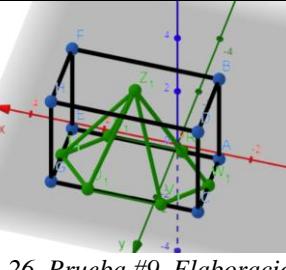
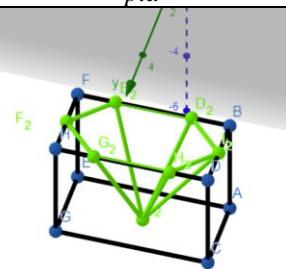
Ahora, para realizar ejecutar cada prueba se seguirá el orden de comandos presentado en la columna “Como se ejecutará”, y para documentar su correcto funcionamiento se presentará en una tabla el nombre de la prueba y sus datos de entrada, la imagen de los comandos ejecutados, y el resultado visual graficado con ayuda de Geogebra que nos permite una identificación eficiente de posibles errores, esta se presentará a continuación en la Tabla 2.

**Tabla 2**  
*Ejecución de las pruebas*

Nombre de la prueba y datos de entrada	Comandos ejecutados	Resultado presentado visualmente
envolvente con un objeto 3D en el primer octante ->PprismaT.txt	\$ cargar PprismaT.txt \$ envolvente	

		<i>Figura 18. Prueba #1. Elaboración propia</i>
envolvente con un objeto 3D en el segundo octante ->Pparalelepipedo.txt	\$ cargar Pparalelepipedo.txt  \$ envolvente	
envolvente con un objeto 3D en el tercer octante ->PprismaP.txt	\$ cargar PprismaP.txt  \$ envolvente	
envolvente con un objeto 3D en el cuarto octante ->Ptetraedro.txt	\$ cargar Ptetraedro.txt  \$ envolvente	

<p>envolvente con un objeto 3D en el quinto octante -&gt; PprismaTN.txt</p>	<p>\$ cargar PprismaTN.txt \$ envolvente</p>	
<p>envolvente con un objeto 3D en el sexto octante -&gt; PparalelepipedoN.txt</p>	<p>\$ cargar PparalelepipedoN.txt \$ envolvente</p>	
<p>envolvente con un objeto 3D en el séptimo octante -&gt; PprismaPN.txt</p>	<p>\$ cargar PprismaPN.txt \$ envolvente</p>	

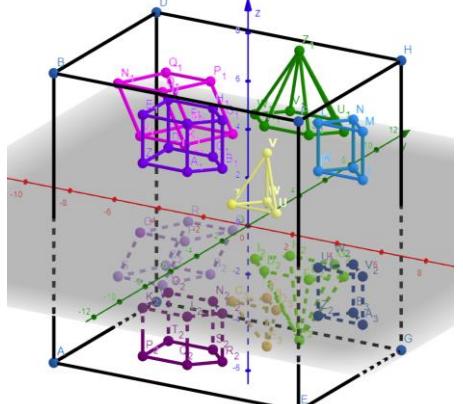
<p>envolvente con un objeto 3D en el octavo octante -&gt; Ptetraedro.txt</p>	<p>\$ cargar PtetraedroN.txt \$ envolvente</p>	 <p><i>Figura 25. Prueba #8. Elaboración propia</i></p>
<p>envolvente con un objeto 3D en el primer y segundo octante -&gt; PpiramideH.txt</p>	<p>\$ cargar PpiramideH.txt \$ envolvente</p>	 <p><i>Figura 26. Prueba #9. Elaboración propia</i></p>
<p>envolvente con un objeto 3D espejo en el primer y segundo octante -&gt; PpiramideHN.txt</p>	<p>\$ cargar PpiramideHN.txt \$ envolvente</p>	 <p><i>Figura 27. Prueba #10. Elaboración propia</i></p>

<p>envolvente de un objeto 3D específico y ver si la incluye en el envolvente global  -&gt;PpiramideHN.txt</p>	<pre>\$ cargar PpiramideHN.txt \$ envolvente PpiramideHexagonal_Negativo \$ envolvente</pre>	<pre>\$ cargar PpiramideHN.txt El objeto PpiramideHexagonal_Negativo ha sido cargado exitosamente desde el archivo PpiramideHN.txt. Elapsed time: 8.023e-05s Processing finished at Fri Aug 23 00:05:35 2024  \$ envolvente PpiramideHexagonal_Negativo (-2, 3, -6) (-2, 3, -3) (-2, 5, -6) (-2, 5, -3) (2, 3, -6) (2, 3, -3) (2, 5, -6) (2, 5, -3)  La caja envolvente del objeto PpiramideHexagonal_Negativo se ha generado con el nombre env_PpiramideHexagonal_Negativo y se ha agregado a los objetos en memoria. Elapsed time: 5.716e-05s Processing finished at Fri Aug 23 00:05:47 2024  \$ envolvente (-2, 3, -6) (-2, 3, -3) (-2, 5, -6) (-2, 5, -3) (2, 3, -6) (2, 3, -3) (2, 5, -6) (2, 5, -3)  La caja envolvente de los objetos en memoria se ha generado con el nombre env_glob al y se ha agregado a los objetos en memoria. Elapsed time: 5.957e-05s Processing finished at Fri Aug 23 00:05:50 2024</pre>
<p>envolvente para todos los objetos 3D en los primeros cuatro octantes  -&gt;PprismaT.txt -&gt;Pparalelepipedo.txt -&gt;PprismaP.txt -&gt;Ptetraedro.txt</p>	<pre>\$ cargar PprismaT.txt \$ cargar Pparalelepipedo.txt \$ cargar PprismaP.txt \$ cargar Ptetraedro.txt \$ envolvente</pre>	
<p>envolvente para todos los objetos 3D en los últimos cuatro octantes  -&gt;PprismaTN.txt -&gt;PparalelepipedoN.txt -&gt;PprismaPN.txt -&gt;PtetraedroN.txt</p>	<pre>\$ cargar PprismaTN.txt \$ cargar PparalelepipedoN.txt \$ cargar PprismaPN.txt \$ cargar PtetraedroN.txt \$ envolvente</pre>	

Figura 28. Prueba #11. Elaboración propia

Figura 29. Prueba #12. Elaboración propia

Figura 30. Prueba #13. Elaboración propia

envolvente para todos los objetos 3D en los ocho octantes -> PprismaT.txt ->Pparalelepipedo.txt ->PprismaP.txt -> Ptetraedro.txt -> PprismaTN.txt ->PparalelepipedoN.txt -> PprismaPN.txt -> PtetraedroN.txt -> PpiramideH.txt ->PpiramideHN.txt	\$ cargar PprismaT.txt \$ cargar Pparalelepipedo.txt \$ cargar PprismaP.txt \$ cargar Ptetraedro.txt \$ cargar PprismaTN.txt \$ cargar PparalelepipedoN.txt \$ cargar PprismaPN.txt \$ cargar PtetraedroN.txt \$ cargar PpiramideH.txt \$ cargar PpiramideHN.txt \$ envolvente	
envolvente sin objetos 3D en memoria	\$ envolvente	<pre>\$ envolvente Ningun objeto ha sido cargado en memoria. Elapsed time: 2.6431e-05s Processing finished at Fri Aug 23 00:43:55 2024</pre>

Nota: La tabla permite ver la ejecución de los casos de prueba para evaluar el comando *envolvente*. Para la elaboración de la columna “comandos ejecutados” se asume que ya se tiene el compilado ejecutado. La grafica puede verse directamente desde Geogebra a través del siguiente enlace: <https://www.geogebra.org/3d/cjyp8ejb>. Elaboración propia.

Es así como concluye nuestro plan de pruebas del componente 1, donde se puede evidenciar visualmente a lo largo de las 15 pruebas como no solo el comando *envolvente* es perfectamente funcional, sino que el componente mismo funciona de maravilla y con

calidad, sea al usar el comando cargar o cualquier otro lo que demuestra un desarrollo exitoso del primer componente.

## 2.4 Corrección Componente 1

En la sustentación se identificaron dos problemas en las funcionalidades del componente:

1. **Carga de Archivo de Malla:** El componente no permitía la correcta carga de archivos que contenían espacios como /n, /r o /t entre líneas, ni leía correctamente los archivos con números de tipo flotante. Esto afectaba directamente el funcionamiento del sistema, ya que la imposibilidad de cargar objetos impedía la prueba del resto de las funcionalidades.
2. **Verificación de parámetros de entrada para los comandos:** El sistema aceptaba un número incorrecto de parámetros (ya fuera superior o inferior al necesario) y seguía funcionando mientras tuviera los parámetros requeridos. Esto afectaba la interfaz con la que interactuaba el usuario.

Para corregir estos problemas, se realizaron las siguientes modificaciones en el código:

1. **Inclusión de la función *trim* en el TAD: Sistema:** Esta corrección, presentada en la figura 33, se implementó para eliminar los tres tipos de espacios (como /n, /r y /t) que impedían la lectura correcta de los archivos. Su uso se refleja en la funcionalidad de carga de archivos, donde, una vez obtenida una línea del archivo, se le aplica la función trim para limpiar los espacios innecesarios. A partir de ahí, la línea se divide en subcadenas más pequeñas con las que se puede trabajar correctamente, lo que resuelve la primera parte del problema relacionado con la carga de archivos.

```
void Sistema::trim (string &s){
    // Eliminar espacios y caracteres especiales al principio y al final
    s.erase(first: & s.begin(), [last: & std::find_if(first:s.begin(), [last:s.end(), [pred: [](unsigned char c)>bool {
        return!std::isspace(c) && c !='\n'&& c !='\t'&& c !='\r';
    }])); 
    s.erase(first: & std::find_if(first: & s.rbegin(), [last: & s.rend(), [pred: [](unsigned char c)>bool {
        return!std::isspace(c) && c !='\n'&& c !='\t'&& c !='\r';
    }].base(), [last: & s.end()]);
}
```

Figura 33. Función trim. Elaboración propia

2. **Modificación de atributos en TAD: Vértice y en la creación de variables temporales:** Estas correcciones, presentadas en las figuras 34 y 35, se implementaron para corregir la imposibilidad de leer archivos con números de tipo flotante, por lo que se resuelve la segunda parte del problema relacionado con la carga de archivos.

```

float x;    if (getline( [&] file, [ &] line)) {
            trim( [&] line);
float y;    istringstream ss(line);
float z;    Vertice vertice;
            float tempX, tempY, tempZ;

```

Figura 34 y 35. Modificación valores flotantes. Elaboración propia

3. **Modificación del programa principal para verificar los parámetros de entrada:** Esta corrección, presentada en la figura 36, se implementó para realizar una verificación más exhaustiva a los parámetros de entrada antes de hacer el llamado a la funcionalidad del sistema comprobando la cantidad total de parámetros ingresados, de forma que se resuelve el problema relacionado con la interfaz de usuario.

```

} else if (cin[0] == comandos[3]){
    if (cin.size() == 2) {
        sistema.descargar(& cin[1]); //Recibe el nombre del objeto a descargar y lo
    } else {
        cout << "El comando: 'descargar' requiere menos o mas parametros, revise el comando"
    }

} else if (cin[0] == comandos[4]){
    if (cin.size() == 3) {
        sistema.guardar(& cin[1], & cin[2]); //Recibe nombre archivo y formato a guardar
    } else {
        cout << "El comando: 'guardar' requiere menos o mas parametros, revise el comando"
    }

} else if (cin[0] == comandos[5]){
    if (cin.size() == 5) {
        sistema.vCercanoObj(& cin[4], stof(cin[1]), stof(cin[2]), stof(cin[3])); //Recibe (x,y,z)
    } else if (cin.size() == 4) {
        sistema.vCercano(stof(cin[1]), stof(cin[2]), stof(cin[3])); //Recibe (x,y,z)
    } else {
        cout << "El comando: 'v-cercano' requiere menos o mas parametros, revise el comando"
    }

} else if (cin[0] == comandos[6]){ //Vertices cercanos a la caja delimitada por
    if (cin.size() == 2) {
        sistema.vCercano_caja(& cin[1]); //Recibe el nombre de la caja delimitadora
    } else {
        cout << "El comando: 'v-cercanos-caja' requiere menos o mas parametros, revise el comando"
    }

} else if (cin[0] == comandos[7]){ //Ruta mas corta entre dos vertices del objeto
    if (cin.size() == 4) {
        sistema.rutaCorta(stoi(cin[1]), stoi(cin[2]), & cin[3]); //Recibe vertices
    } else {
        cout << "El comando: 'ruta-corta' requiere menos o mas parametros, revise el comando"
    }
}

```

Figura 36. Verificación de parámetros de entrada. Elaboración propia

Para comprobar que los cambios efectuados solucionaron los problemas encontrados fue diseñado y ejecutado un nuevo plan de pruebas; por lo que se creó una tabla que presentara el nombre de la prueba, el resultado esperado y los datos de entrada, la justificación de esta y como se va a ejecutar, enfocada en demostrar que la funcionalidad y calidad del componente es la adecuada.

Como bien se mencionó previamente se encontraron 2 problemas principales: En cuanto a la carga de archivos y en cuanto a la comprobación de parámetros. Por lo que, nos pareció preciso que los casos de prueba consistieran en comprobar que el sistema es capaz de leer los dos archivos principales que en la sustentación no fueron exitosos (sea por los espacios o por contener números flotantes) y revisar con dos ejemplos que salga el mensaje de error de ingresarse parámetros incorrectos, lo que se ve reflejado en la tabla 3.

La estrategia que tenemos la intención de utilizar es: primero descargar los 2 archivos de texto (representantes cada uno de un objeto 3D) que no habían sido posibles de leer y segundo ejecutar el comando ‘cargar’ para cada uno de ellos para así ejecutar los dos primeros casos de prueba. Posteriormente comprobar los parámetros de ingresar el comando ‘listado objetos’ y ‘cargar nombredelarchivo nombredelobjeto’ logrando así un total de 4 casos de prueba.

**Tabla 3**  
*Plan de pruebas*

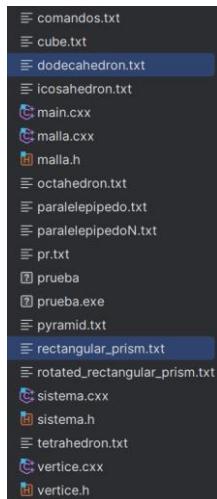
Nombre de la prueba	Resultado esperado y datos de entrada	Justificación	Cómo se ejecutará
Carga de un archivo que contiene espacios como /n, /r o /t entre líneas	Objeto cargado en memoria del sistema ->Archivo de texto del objeto 3D	Evaluar la funcionalidad y calidad del componente con al cargar un archivo con espacios	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar el siguiente comando: \$ cargar archivo.txt
Carga de un archivo que contiene números de tipo flotante	Objeto cargado en memoria del sistema ->Archivo de texto del objeto 3D	Evaluar la funcionalidad y calidad del componente con al cargar un archivo con números flotantes.	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar el siguiente comando: \$ cargar archivo.txt

Validación #1 de parámetros ingresados en la interfaz de usuario	Mensaje de error por cantidad de parámetros incorrecta	Evaluar la funcionalidad y calidad del componente ejecutando con más parámetros incluso junto a los necesarios	1 Compilar y ejecutar el sistema 2 Ejecutar el siguiente comando:  \$ listado objetos
Validación #2 de parámetros ingresados en la interfaz de usuario	Mensaje de error por cantidad de parámetros incorrecta	Evaluar la funcionalidad y calidad del componente ejecutando con más parámetros incluso junto a los necesarios	1 Compilar y ejecutar el sistema 2 Ejecutar el siguiente comando:  \$ cargar nombredelarchivo nombredelobjeto

*NOTA: La tabla permite ver la planeación de casos de prueba para evaluar las correcciones del componente 1, elaboración propia.*

### Ejecución de Pruebas

Siguiendo la estrategia presentada en la planeación de pruebas se hizo la respectiva descarga de archivos de forma que quedaron los archivos presentados en la Figura 37, donde ‘dodecahedron.txt’ contiene de números flotantes y ‘rectangular\_prism.txt’ contiene los espacios extra.



*Figura 37. Descarga de los dos archivos a probar. Elaboración propia*

Ahora, para realizar ejecutar cada prueba se seguirá el orden de comandos presentado en la columna “Como se ejecutará”, y para documentar su correcto funcionamiento se presentará en una tabla el nombre de la prueba y sus datos de entrada, los comandos ejecutados, y el resultado visual en consola, esta se presentará a continuación en la Tabla 4.

**Tabla 4**  
*Ejecución de las pruebas*

Nombre de la prueba y datos de entrada	Comandos ejecutados	Resultado presentado visualmente
Carga de un archivo que contiene espacios como /n, /r o /t entre líneas ->rectangular_prism.txt	\$ cargar rectangular_prism.txt	\$ cargar rectangular_prism.txt El objeto Rectangular_Prism ha sido cargado exitosamente desde el archivo rectangular_prism.txt. Elapsed time: 0.00882501s Processing finished at Mon Oct 14 15:52:59 2024
		<i>Figura 38. Prueba #1. Elaboración propia</i>
Carga de un archivo que contiene números de tipo flotante ->dodecahedron.txt	\$ cargar dodecahedron.txt	\$ cargar dodecahedron.txt El objeto Dodecahedron ha sido cargado exitosamente desde el archivo dodecahedron.txt. Elapsed time: 0.00961848s Processing finished at Mon Oct 14 15:55:15 2024
		<i>Figura 39. Prueba #2. Elaboración propia</i>
Validación #1 de parámetros ingresados en la interfaz de usuario	\$ listado objetos	\$ listado objetos El comando: 'listado' no requiere parametros, revise el comando a ejecutar consultando la guia con el comando: 'ayuda'. Elapsed time: 5.1065e-05s Processing finished at Mon Oct 14 15:57:21 2024
		<i>Figura 40. Prueba #3. Elaboración propia</i>
Validación #2 de parámetros ingresados en la interfaz de usuario	\$ cargar nombreDelArchivo nombreObjeto	\$ cargar nombreArchivo nombreObjeto El comando: 'cargar', requiere menos o mas parametros, revise el comando a ejecutar consultando la guia con el comando: 'ayuda'. Elapsed time: 4.0103e-05s Processing finished at Mon Oct 14 15:58:13 2024
		<i>Figura 41. Prueba #4. Elaboración propia</i>

*Nota: La tabla permite ver la ejecución de los casos de prueba para evaluar las correcciones del componente 1. Elaboración propia.*

Es así como concluye nuestro plan de pruebas para las correcciones del componente 1, donde se puede evidenciar visualmente a lo largo de las 4 pruebas como el componente es funcional, demostrando que las correcciones solo cambiaron detalles de los TADs: Sistema (con la inclusión de la función *trim* y la modificación de las variables temporales) y Vértice (con la modificación de sus atributos) evitando el surgimiento de

nuevos problemas, mejorando el componente comprobando que ahora cumple con los requisitos esperados.

### 3 Componente 2: Vértices más cercanos

Este componente tiene como fin el desarrollo de funcionalidades principalmente enfocadas en la identificación de vértices más cercanos entre los objetos, por lo que al hacer uso de tantas características de las mallas poligonales (incluso de sus envolventes) permite evidenciar usos del sistema.

#### Objetivos de los comandos:

1. **Vértice cercano a un objeto:** Este comando permite al sistema determinar el vértice más cercano a un punto tridimensional específico (en términos de la distancia euclídea) dentro de un objeto particular. Esta funcionalidad es fundamental para realizar análisis detallados sobre la geometría del objeto y puede ser útil en aplicaciones como la visualización y la simulación.
2. **Vértice cercano a todos los objetos:** Este comando permite al sistema identificar el vértice más cercano a un punto dado de todos los objetos cargados en memoria, incluyendo las mallas envolventes. Esta capacidad es esencial para realizar comparaciones y análisis entre diferentes objetos en el sistema, facilitando tareas como la optimización de modelos y la interacción en entornos tridimensionales.
3. **Vectores cercanos de un objeto y su envolvente:** Este comando permite al sistema identificar los vértices de un objeto 3D que son más cercanos a las esquinas de su caja envolvente. Es crucial para optimizar procesos de visualización y análisis espacial, así como para mejorar la eficiencia en la manipulación de mallas complejas.

### 3.1 Descripción de los Comandos

#### 1. Vértice cercano a un objeto:

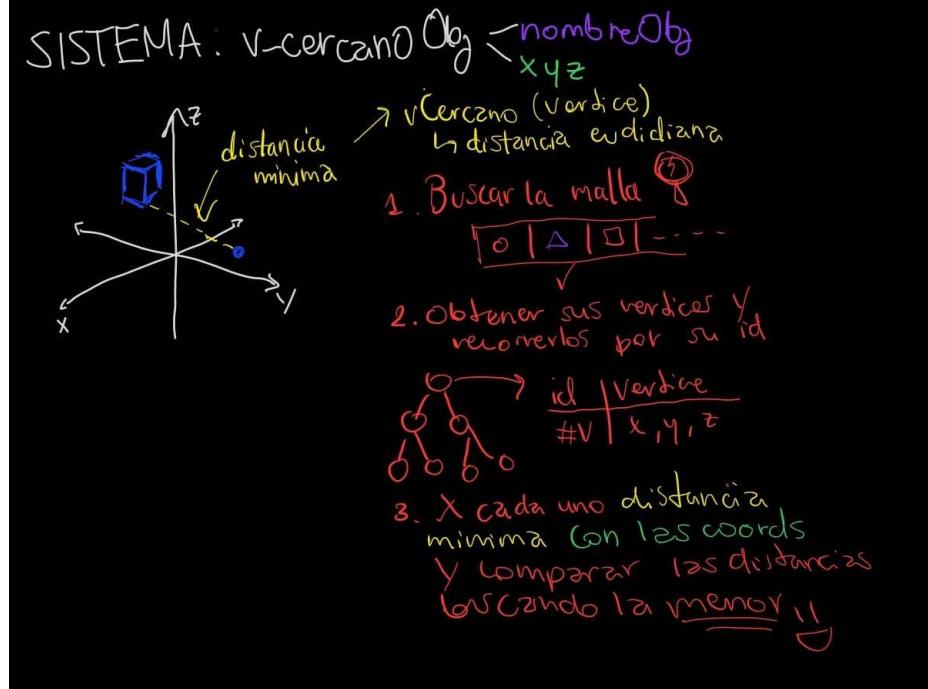


Figura 42. Diseño del comando  $v\_cercano \text{ x y z Objeto}$ . Elaboración propia

En la Figura 42, podemos observar la idea que se plantea para realizar la acción requerida por el comando  $v\_cercano \text{ x y z Objeto}$ .

**Descripción:** Este comando se implementa en dos partes, en la malla y en el sistema. El sistema comienza verificando si el nombre de la malla ingresado por el usuario corresponde a una malla que ya ha sido cargada en el sistema. Si la malla no ha sido cargada, se informa al usuario. En caso de que la malla exista, la función crea un punto tridimensional con las coordenadas proporcionadas y e interiormente de cada malla se realizará un proceso que calcula qué vértice de la malla es el más cercano a ese punto. Finalmente, la función muestra al usuario el índice del vértice más cercano, sus coordenadas y la distancia al punto de referencia.

#### Requisitos previos:

- Objeto cargado en memoria: La malla correspondiente al nombre ingresado debe haber sido previamente cargada en el sistema y su nombre debe coincidir exactamente con el que fue utilizado al cargarla en el sistema.
- Coordenadas del punto: Las coordenadas (x, y, z) deben ser números flotantes válidos, y representar un punto en el espacio

tridimensional al que se calculará la distancia más cercana desde los vértices de la malla.

- Método buscarMalla: Debe estar correctamente implementado para devolver una instancia válida de la malla si existe, o una malla vacía Malla() si no se encuentra el objeto con el nombre especificado.
- Función vCercano en la clase Malla: Esta función debe estar implementada para devolver un vector con el índice del vértice más cercano y la distancia a dicho vértice.

#### **Proceso interno de la función correspondiente a este comando:**

- 1) Búsqueda de la malla por nombre:
  - i. La función invoca buscarMalla(nombre) para intentar localizar la malla asociada con el nombre proporcionado.
  - ii. Si la malla no se encuentra (es decir, si buscarMalla devuelve una malla vacía Malla()), se muestra el mensaje de que no ha sido cargada en memoria al usuario. En este caso, la función termina, ya que no puede continuar sin una malla cargada.
- 2) Creación del vértice de referencia:
  - i. Si la malla fue encontrada, se crea un vértice temporal vert con las coordenadas del punto (x, y, z) proporcionadas por el usuario.
  - ii. Este vértice representa el punto de referencia desde el cual se buscará el vértice más cercano en la malla.
- 3) Cálculo del vértice más cercano:
  - i. Se invoca m.vCercano(vert), que devuelve un vector con dos elementos:
    1. Índice del vértice más cercano dentro de la malla.
    2. Distancia (euclíadiana) entre el vértice más cercano y el punto dado (x, y, z).
  - ii. Si el vector está vacío, significa que hubo un problema al calcular la distancia o no se encontraron vértices válidos. En ese caso, no se muestra ningún mensaje de éxito.
- 4) Muestra del resultado:
  - i. Si el vector no está vacío, se extrae el índice del vértice más cercano con valores[0] y la distancia con valores[1].
  - ii. Luego, se obtiene el vértice correspondiente del vector de vértices de la malla, y se imprimen en pantalla las coordenadas del vértice más cercano y la distancia calculada.

#### **Resultados e Interpretación:**

- (Objeto no encontrado): Si el mensaje "El objeto [nombre] no ha sido cargado en memoria" aparece, esto indica que el objeto con el nombre dado no ha sido cargado en el sistema. Debe verificarse si el nombre es correcto y si la malla ha sido previamente cargada.

- (Resultado exitoso): El mensaje "El vértice i (vx, vy, vz) del objeto [nombre] es el más cercano al punto (px, py, pz), a una distancia de valor\_distancia" implica:
  - i: El índice del vértice más cercano dentro de la malla.
  - (vx, vy, vz): Las coordenadas del vértice más cercano.
  - (px, py, pz): Las coordenadas del punto de referencia proporcionadas por el usuario.
  - valor\_distancia: La distancia calculada entre el vértice más cercano y el punto dado
- (Resultado no encontrado): Si la función no devuelve ningún resultado ni imprime un mensaje de éxito, es posible que: No se haya encontrado ningún vértice válido en la malla o la malla esté vacía o malformada.

**Verificación de Ejecución Correcta:**

- Mensajes en Consola: La forma más directa de verificar que la función se ha ejecutado correctamente es revisar el mensaje mostrado en la consola. Cada posible escenario tiene un mensaje específico que indica el resultado de la operación.
- Estado del contenedor mallas: Después de ejecutar la función, si se imprimió el mensaje de éxito, se puede comparar el vértice mostrado en la consola con los vértices existentes en la malla para validar que el índice y las coordenadas son correctos.

**2. Vértice cercano a todos los objetos:**

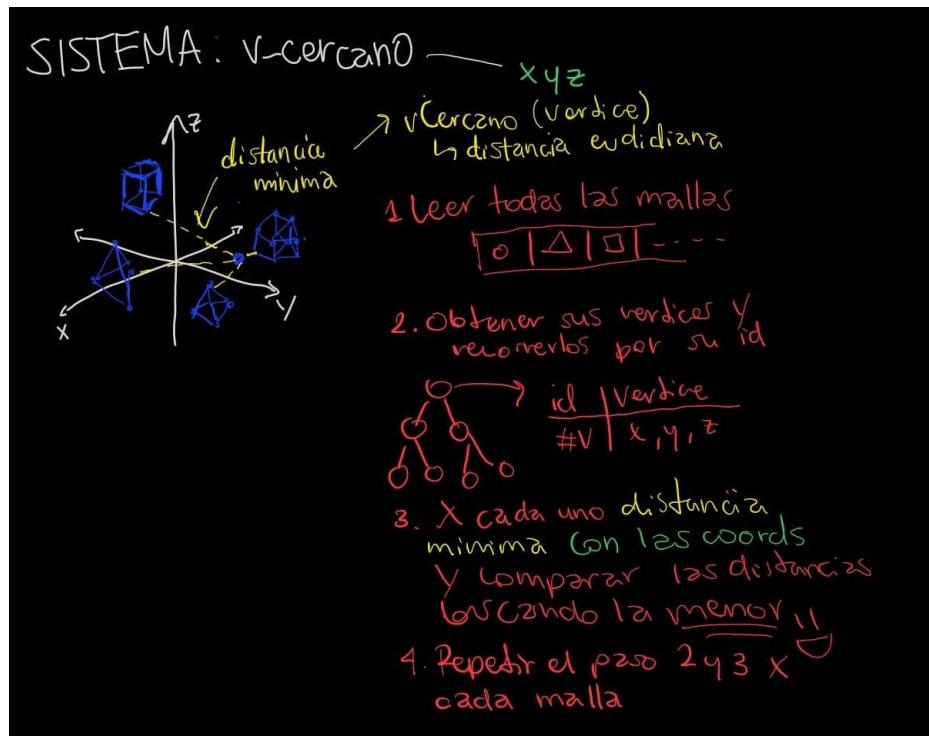


Figura 43. Diseño del comando *v\_cercano x y z*. Elaboración propia

En la Figura 43, podemos observar la idea que se plantea para realizar la acción requerida por el comando *v\_cercano x y z*.

**Descripción:** Este comando se implementa en dos partes, en la malla y en el sistema. El sistema comienza creando un punto tridimensional con las coordenadas proporcionadas y leyendo cada una de las mallas dentro de la memoria del sistema e interiormente de cada malla se realizará un proceso que calcula qué vértice de la malla es el más cercano a ese punto. Finalmente, la función muestra al usuario el índice del vértice más cercano, sus coordenadas y la distancia al punto de referencia.

#### Requisitos previos:

- Objetos Cargados: Debe haber objetos cargados en el vector *mallas* para que la función tenga algo que listar.
- Coordenadas del punto: Las coordenadas (*x*, *y*, *z*) deben ser números flotantes válidos que representan un punto en el espacio tridimensional.
- Función *vCercano* en la clase *Malla*: Esta función debe estar implementada para devolver un vector con el índice del vértice más cercano y la distancia a dicho vértice

#### Proceso interno de la función correspondiente a este comando:

- 1) Verificación de Memoria Vacía:

- i. La función primero verifica si el contenedor de mallas está vacío.
  - ii. Si está vacío, se muestra el mensaje "Ningún objeto ha sido cargado en memoria." y la función termina.
- 2) Inicialización y variables auxiliares:
- i.  $d_{min}$  guarda la menor distancia encontrada, inicializada con el valor más alto posible.
  - ii.  $malla$  y  $vert$  almacenan el índice de la malla y el vértice más cercano, respectivamente.
- 3) Iteración sobre las mallas:
- i. La función recorre todas las mallas cargadas en memoria utilizando un iterador
  - ii. Para cada malla, se llama a  $vCercano(vertice)$ , que devuelve un vector con dos valores:
    - 1. El índice del vértice más cercano
    - 2. La distancia entre el vértice y el punto ( $x, y, z$ )
  - iii. Si la distancia calculada es menor que  $d_{min}$ , se actualizan las variables  $d_{min}$ ,  $malla$  y  $vert$  con los nuevos valores
- 4) Muestra del resultado:
- i. Una vez que se ha encontrado el vértice más cercano, se obtiene su información de la malla correspondiente
  - ii. Se imprime en consola el índice del vértice más cercano, sus coordenadas, el nombre del objeto al que pertenece, y la distancia

#### **Resultados e Interpretación:**

- (Memoria Vacía): Si no hay objetos en mallas, se muestra el mensaje "Ningún objeto ha sido cargado en memoria." Esto indica que no hay datos almacenados en memoria para listar.
- (Resultado Exitoso): El mensaje "El vértice  $i$  ( $vx, vy, vz$ ) del objeto [nombre] es el más cercano al punto ( $px, py, pz$ ), a una distancia de  $valor\_distancia$ " implica:
  - $i$ : El índice del vértice más cercano dentro de la malla.
  - $(vx, vy, vz)$ : Las coordenadas del vértice más cercano.
  - $(px, py, pz)$ : Las coordenadas del punto de referencia.
  - $valor\_distancia$ : La distancia entre el vértice más cercano y el punto

#### **Verificación de Ejecución Correcta:**

- Mensajes en Consola: Se revisa que los mensajes que se muestran en la consola para determinar si la función ha sido ejecutada

correctamente. Estos mensajes dirán si la memoria está vacía o cuál es el vértice correspondiente.

- Estado de las mallas: Si se muestra el mensaje de éxito, se puede validar que el vértice mostrado es el correcto al revisar las coordenadas y el índice del vértice en la malla correspondiente.

### 3. Vectores cercanos de un objeto y su envolvente:

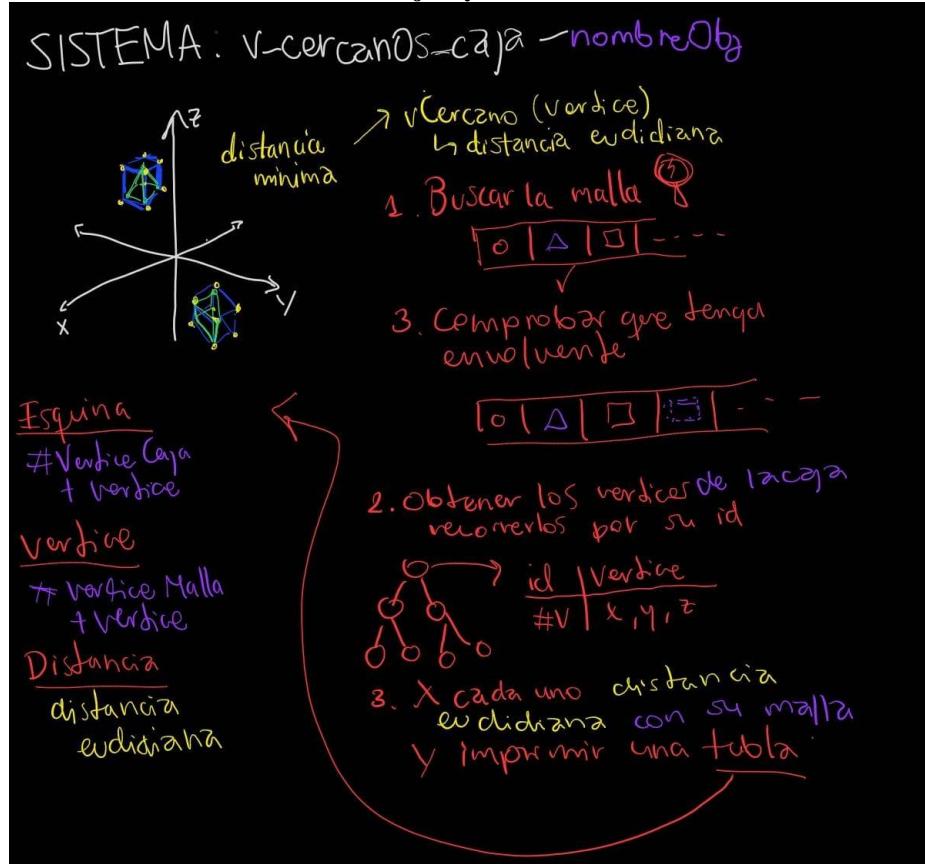


Figura 44. Diseño del comando *v\_cercano\_caja Objeto*. Elaboración propia

En la Figura 44, podemos observar la idea que se plantea para realizar la acción requerida por el comando *v\_cercano\_caja Objeto*.

**Descripción:** Este comando se implementa en dos partes, en la malla y en el sistema. El sistema comienza verificando si el nombre de la malla ingresado por el usuario corresponde a una malla que ya ha sido cargada en el sistema. Si la malla no ha sido cargada, se informa al usuario. En caso de que la malla exista, la función comprueba que el envolvente este calculado y de ser así la función procede a identificar el vértice de la malla más cercano a cada una de las ocho esquinas de la caja envolvente con el cálculo que devuelve la función interna a la malla. Finalmente, se muestra una tabla

que detalla las coordenadas de cada esquina, el vértice más cercano y la distancia calculada entre ambos puntos

**Requisitos previos:**

- Objeto en Memoria: El objeto 3D que se va a envolver debe estar cargado en memoria y disponible para su búsqueda.
- Caja envolvente calculada: El objeto debe tener su caja envolvente calculada. De lo contrario, la función no podrá realizar la búsqueda de vértices cercanos a las esquinas
- Función vCercano en la clase Malla: Esta función debe estar implementada para devolver un vector que contenga el índice del vértice más cercano y la distancia a dicho vértice.

**Proceso interno de la función correspondiente a este comando:**

- 1) Búsqueda del Objeto en Memoria:
  - i. La función comienza llamando a buscarMalla(nombre), que busca en la memoria un objeto 3D con el nombre especificado.
  - ii. Si buscarMalla() no encuentra el objeto, devuelve una malla vacía, y la función muestra el mensaje "El objeto nombre\_objeto no ha sido cargado en memoria." y termina la ejecución.
- 2) Verificación de la caja envolvente:
  - i. Si la malla fue encontrada, la función verifica si la caja envolvente ha sido calculada, si no la función finaliza en este punto.
- 3) Iteración sobre las esquinas de la caja envolvente:
  - i. Si la caja envolvente existe, la función obtiene los vértices de la caja envolvente y los vértices del objeto original.
  - ii. Se itera sobre las 8 esquinas de la caja envolvente, identificando el vértice más cercano de la malla a cada una de esas esquinas mediante la función vCercano.
- 4) Muestra de resultados:
  - i. Para cada esquina, la función muestra en pantalla el vértice más cercano junto con la distancia euclíadiana calculada. La salida está organizada en formato de tabla.

**Resultados e Interpretación:**

- (Objeto no existe): Si el objeto nombre\_objeto no está cargado en memoria, se muestra el mensaje "El objeto nombre\_objeto no ha sido cargado en memoria." Esto indica que no se pudo calcular la caja envolvente porque el objeto no está disponible.
- (Caja envolvente no calculada) Si el mensaje "No ha calculado aún la caja envolvente del objeto [nombre]" aparece, indica que aún no

se ha generado la caja envolvente del objeto, y no es posible continuar hasta que esto se haga

- (Resultado exitoso): Si el proceso se ejecuta correctamente, se mostrará una tabla con los vértices más cercanos a cada esquina de la caja envolvente. Cada fila de la tabla mostrará:
  - **Esquina:** Coordenadas (ex, ey, ez) de la esquina de la caja envolvente.
  - **Vértice más cercano:** El índice y las coordenadas (vx, vy, vz) del vértice más cercano en la malla.
  - **Distancia:** La distancia euclíadiana calculada entre la esquina y el vértice

#### **Verificación de Ejecución Correcta:**

- Mensajes en Consola: Revisa el mensaje que se muestra en la consola después de la ejecución de la función para confirmar si la operación fue exitosa según el estado de la malla y su envolvente.
- Revisión de los vértices: Si se muestra una tabla con los vértices más cercanos, se puede validar comparando las coordenadas de los vértices de la malla y las esquinas de la caja envolvente

#### **4. Vértice más cercano a un vértice**

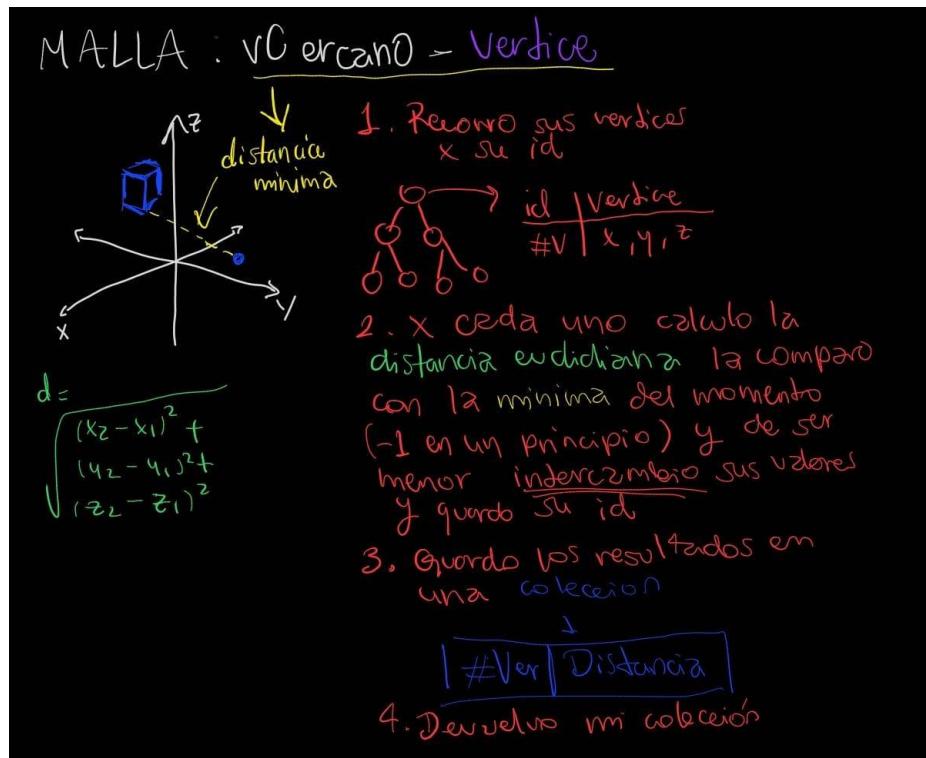


Figura 45. Diseño de la función *v\_cercano(Vertice)*. Elaboración propia

En la Figura 45, podemos observar la idea que se plantea para realizar la acción requerida por la función *v\_cercano(Vertice)* la cual es importante precisar pues como se pudo evidenciar el en proceso interno de las funciones correspondientes a los últimos 3 comandos, es a esta función del TAD: Malla a la que se refieren los 3 comandos para encontrar el cálculo del vértice más cercano.

**Descripción:** Esta función es la segunda parte de los últimos 3 comandos (la de la malla) y tiene como propósito identificar el vértice de una malla que esté más cercano a un punto dado (definido por las coordenadas del vértice pasado como parámetro), permitiendo obtener tanto el índice del vértice más cercano como la distancia entre ambos puntos. La función recorre todos los vértices de la malla y, mediante el cálculo de la distancia euclidiana, selecciona el vértice cuya distancia a las coordenadas ingresadas sea la menor.

#### Requisitos previos:

- Malla cargada en memoria: La función depende de que los vértices de la malla ya estén cargados en memoria, de modo que puedan ser iterados y comparados con las coordenadas dadas

- Vértices correctamente definidos: Se espera que los vértices de la malla tengan coordenadas (x, y, z) válidas que representen posiciones en el espacio tridimensional
- Vector de vértices accesible: El acceso a los vértices de la malla debe estar correctamente implementado, de manera que se pueda iterar sobre ellos.

**Proceso interno de la función:**

- 1) Inicialización de variables:
  - i. La función comienza definiendo dos variables para llevar el control de la distancia mínima (`d_min`) y el índice del vértice más cercano (`vert_min`), ambos inicializados a valores que permitan identificar el vértice correcto a lo largo de la iteración
- 2) Iteración sobre los vértices de la malla:
  - i. La función recorre todos los vértices almacenados en la malla utilizando un bucle que va de 0 a `n_vertices`, que es el número total de vértices en la malla.
  - ii. Para cada vértice, se calcula la distancia euclíadiana entre el vértice actual y las coordenadas del vértice dado como parámetro utilizando la fórmula:  $d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$
  - iii. Si la distancia calculada es menor que la mínima registrada hasta ese momento, se actualizan tanto la distancia mínima como el índice del vértice más cercano
- 3) Devolución de resultados:
  - i. Si se ha encontrado un vértice más cercano (es decir, si `vert_min` es distinto de -1), la función almacena el índice del vértice más cercano y la distancia calculada en un vector de flotantes (valores), que será devuelto al final de la función.

**Resultados e Interpretación:**

- (Resultado exitoso): La función devuelve un vector con dos elementos: el índice del vértice más cercano y la distancia mínima entre dicho vértice y el punto de referencia.
- (Sin resultados): Si por alguna razón no se encuentran vértices válidos, la función devolverá un vector vacío. Esto indicaría que no fue posible realizar el cálculo de proximidad

**Verificación de Ejecución Correcta:**

- Comparación manual: Los resultados devueltos (índice y distancia) pueden ser comparados manualmente con las coordenadas de los vértices de la malla para verificar que el cálculo sea correcto.
- Mensajes en consola: Aunque esta función no muestra mensajes en consola, su correcta integración dentro del sistema puede ser

verificada con las funciones que la utilizan, observando los resultados que se muestran en pantalla

### 3.2 Diseño

A continuación, se muestra un bosquejo del diseño inicial (Figura 45) que se planteó para modelar el programa para el segundo componente:

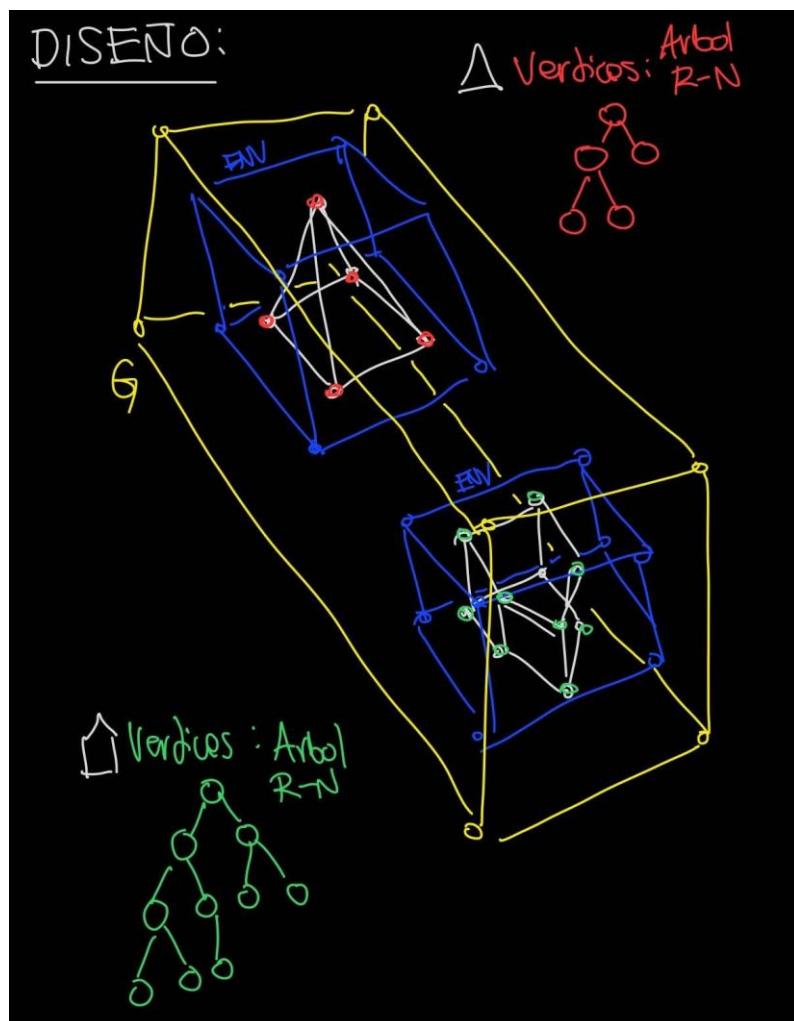


Figura 46. Diseño segundo componente. Elaboración propia

Con esta base planteada se procedió al diseño de TAD's para la correcta ejecución de los comandos.

### Descripción de los Tipos Abstractos de Datos (TADs):

A continuación, se presenta la especificación de los tres TAD's creados para la realización de este proyecto, empezando con la especificación del TAD de Sistema en la Figura 46:

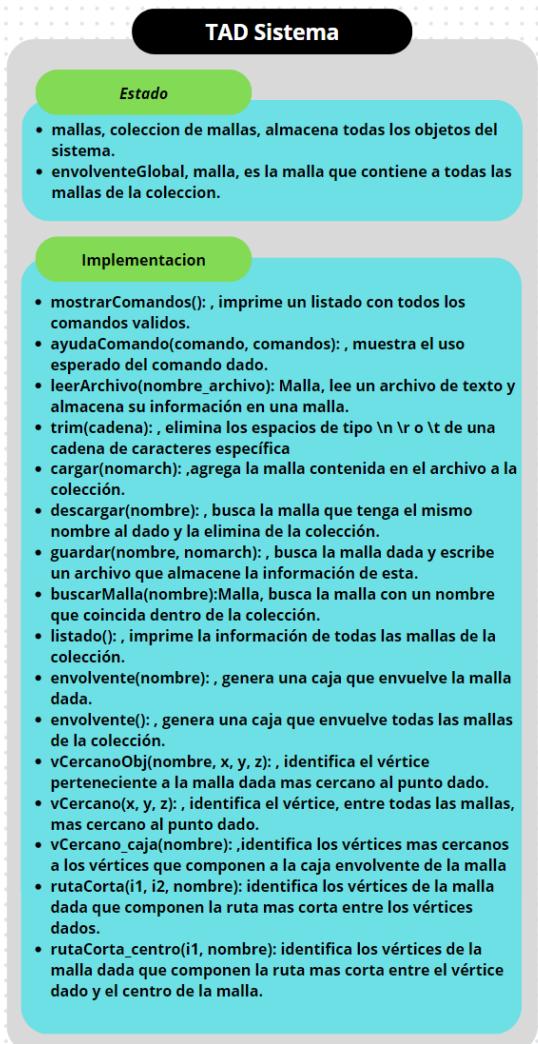


Figura 47. Especificación de sistema. Elaboración propia

Para el sistema se sigue planteando que contenga dos atributos: **mallas**, siendo una colección de mallas; y **envolventeGlobal** que contendría una caja que envuelva a todos los objetos de la colección. En cuanto a su implementación, se tendrán ahora 16 funciones que facilitarán el cumplimiento de las acciones requeridas por los comandos. Ahora se mostrará la especificación del TAD de malla en la Figura 47:

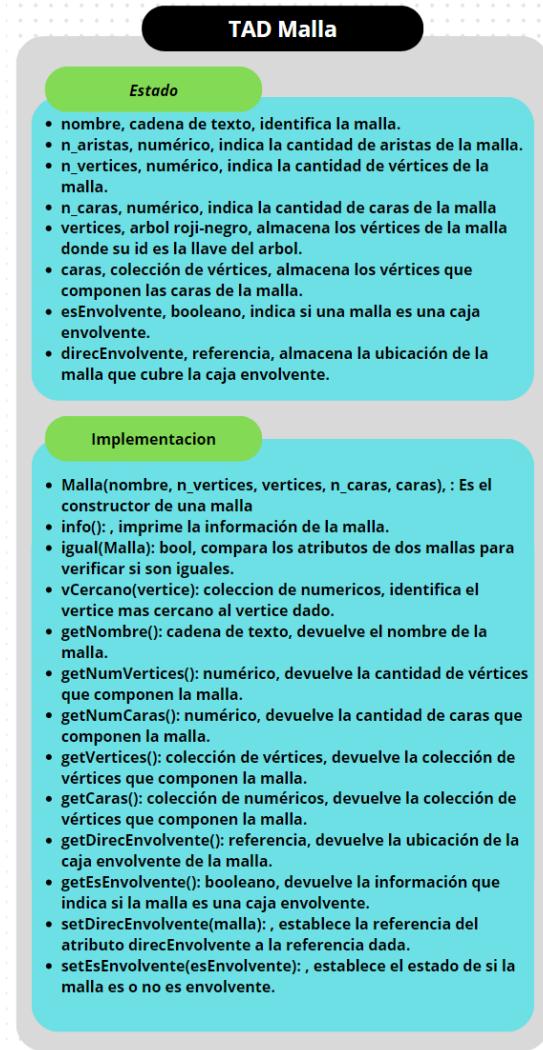


Figura 48. Especificación de malla. Elaboración propia

Para este TAD se considera la información que se recolecta de los archivos, de forma que cuando se lea un archivo de texto se pueda construir de la mejor forma una malla ahora guardando a los vértices en un árbol roji-negro. La malla cuenta con 13 funciones que son llamadas por el sistema para ejecutar los comandos. Por último, se muestra el TAD de vértice en la Figura 48:



Figura 49. Especificación de vértice. Elaboración propia

Para este TAD se tiene un id y las coordenadas (x,y,z) que representan un punto del plano 3D. En cuanto a la implementación se tienen los getters y setters de sus atributos y una función adicional que calcula la distancia euclíadiana entre dos vértices de la cual hicimos uso durante todo este componente junto a una que sobrecarga el operador == para que el árbol rojinegro se construya correctamente.

#### Diagrama de Relación de TADs:

A continuación, se muestra el diagrama de relación entre los TAD's utilizados por el programa:

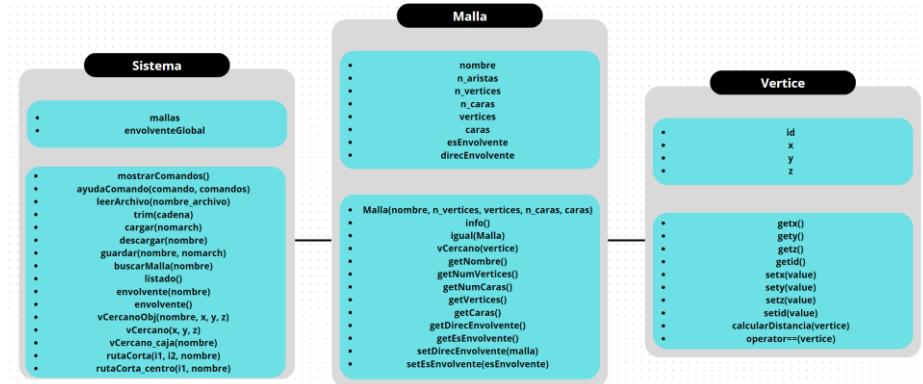


Figura 50. Diagrama de relación de TAD's. Elaboración propia

Es de resaltar que el main sigue funcionando como intermediario para las funcionalidades requeridas. El sistema, sigue gestionando la información de las mallas cargadas por medio de métodos propios del TAD de sistema y por métodos que forman parte del TAD de Malla. Un ejemplo que demuestra la integración entre los TAD's es cuando se usa cualquier comando de este componente como “v\_cercano x y z”, pues requiere el uso del TAD de vértice, que facilita las operaciones para comparar vertices, que están dentro de objetos de tipo TAD:Malla que hace uso de la funcionalidad vCercano(vertex) de la que a su vez hace llamado el TAD: Sistema en la función correspondiente.

### 3.3 Plan de Pruebas

#### Planificación de Pruebas

Previo a la codificación, fue realizado un proceso de análisis respecto al funcionamiento de los 3 comandos de vértices cercanos para de esa forma crear una tabla que presentara el nombre de la prueba, el resultado esperado y los datos de entrada, la justificación de esta y como se va a ejecutar; lo anterior, de forma que fuera posible evaluar la mayor cantidad de casos de prueba posibles que demostrarán la funcionalidad y calidad del componente en general.

Para ello, nos pareció preciso que los casos de prueba consistieran en comprobar que los 8 archivos que subió el profe funcionaran sin importar las diferencias entre estos; donde funcionar se refiere a que cargaran en el sistema y que a partir de ellos se pudieran ejecutar los 3 comandos, lo que se ve reflejado en la tabla 5.

La estrategia que tenemos la intención de utilizar es: primero, descargar los 8 archivos de texto (representante cada uno de un objeto 3D) para posteriormente hacer las siguientes pruebas: Por cada uno de ellos cargar el archivo y ejecutar vertice\_cercano para el y su envolvente, llegados a 4 de ellos ejecutar vertice\_cercano general y

posteriormente cargar los últimos 4 siguiendo la misma estrategia de los primeros y finalmente nuevamente ejecutar vertice\_cercano general para el mismo punto para ver si cambio; logrando así un total de 18 casos de prueba, pero vamos a juntar los dos casos que hay por cada archivo de forma que tendremos 10 en total.

**Tabla 5**  
*Plan de pruebas*

Nombre de la prueba	Resultado esperado y datos de entrada	Justificación	Cómo se ejecutará
Primer archivo con vértice cercano sobre él y su caja	Vértice cercano calculado para ambos casos ->Archivo de texto del objeto 3D	Evaluar la funcionalidad y calidad del comando con un objeto 3D diferente.	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar los siguientes comandos:  \$ cargar obj1.txt \$ v_cercano x y z Obj1 \$ envolvente Obj1 \$ v_cercanos_caja Obj1
Segundo archivo con vértice cercano sobre él y su caja	Vértice cercano calculado para ambos casos ->Archivo de texto del objeto 3D	Evaluar la funcionalidad y calidad del comando con un objeto 3D diferente.	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar los siguientes comandos:  \$ cargar obj2.txt \$ v_cercano x y z Obj2 \$ envolvente Obj2 \$ v_cercanos_caja Obj2
Tercer archivo con vértice cercano sobre él y su caja	Vértice cercano calculado para ambos casos ->Archivo de texto del objeto 3D	Evaluar la funcionalidad y calidad del comando con un objeto 3D diferente.	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar los siguientes comandos:  \$ cargar obj3.txt \$ v_cercano x y z Obj3 \$ envolvente Obj3 \$ v_cercanos_caja Obj3

Cuarto archivo con vértice cercano sobre él y su caja	Vértice cercano calculado para ambos casos ->Archivo de texto del objeto 3D	Evaluar la funcionalidad y calidad del comando con un objeto 3D diferente.	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar los siguientes comandos:  \$ cargar obj4.txt \$ v_cercano x y z Obj4 \$ envolvente Obj4 \$ v_cercanos_caja Obj4
Calcular el vértice cercano general sobre 8 objetos en memoria donde 4 con mallas y 4 son envolventes	Vértice cercano calculado para todos los objetos ->Cuatro archivos de texto de los objetos 3D de las últimas 4 pruebas	Evaluar la funcionalidad y calidad del comando con varios objetos 3D	1 Agregar los últimos 4 archivos de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema con los últimos 4 casos de prueba seguidos 3 Ejecutar los siguientes comandos:  \$ listado \$ v_cercano x y z
Quinto archivo con vértice cercano sobre él y su caja	Vértice cercano calculado para ambos casos ->Archivo de texto del objeto 3D	Evaluar la funcionalidad y calidad del comando con un objeto 3D diferente.	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar los siguientes comandos:  \$ cargar obj5.txt \$ v_cercano x y z Obj5 \$ envolvente Obj5 \$ v_cercanos_caja Obj5
Sexto archivo con vértice cercano sobre él y su caja	Vértice cercano calculado para ambos casos ->Archivo de texto del objeto 3D	Evaluar la funcionalidad y calidad del comando con un objeto 3D diferente.	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar los siguientes comandos:  \$ cargar obj6.txt \$ v_cercano x y z Obj6 \$ envolvente Obj6

			\$ v_cercanos_caja Obj6
Séptimo archivo con vértice cercano sobre él y su caja	Vértice cercano calculado para ambos casos ->Archivo de texto del objeto 3D	Evaluar la funcionalidad y calidad del comando con un objeto 3D diferente.	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar los siguientes comandos:  \$ cargar obj7.txt \$ v_cercano x y z Obj7 \$ envolvente Obj7 \$ v_cercanos_caja Obj7
Octavo archivo con vértice cercano sobre él y su caja	Vértice cercano calculado para ambos casos ->Archivo de texto del objeto 3D	Evaluar la funcionalidad y calidad del comando con un objeto 3D diferente.	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar los siguientes comandos:  \$ cargar obj8.txt \$ v_cercano x y z Obj8 \$ envolvente Obj8 \$ v_cercanos_caja Obj8
Calcular el vértice cercano general sobre 16 objetos en memoria donde 8 con mallas y 8 son envolventes	Vértice cercano calculado para todos los objetos ->Ocho archivos de texto de los objetos 3D de las ultimas 4 pruebas	Evaluar la funcionalidad y calidad del comando con varios objetos 3D	1 Agregar los últimos 8 archivos de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema con los últimos 8 casos de prueba seguidos 3 Ejecutar los siguientes comandos:  \$ listado \$ v_cercano x y z

*NOTA: La tabla permite ver la planeación de casos de prueba para evaluar el componente 2, elaboración propia.*

### Ejecución de Pruebas

Siguiendo la estrategia presentada en la planeación de pruebas se hizo la respectiva descarga e ingreso de archivos de forma que quedaron los archivos presentados en la Figura 51.

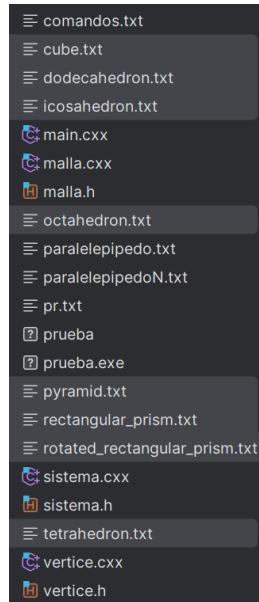


Figura 51. Descarga e ingreso de los 8 archivos. Elaboración propia

Ahora, para realizar ejecutar cada prueba se seguirá el orden de comandos presentado en la columna “Como se ejecutará”, y para documentar su correcto funcionamiento se presentará en una tabla el nombre de la prueba y sus datos de entrada, y la imagen de los comandos ejecutados junto a su resultado resaltado permitiendo una identificación eficiente de posibles errores, esta se presentará a continuación en la Tabla 6.

**Tabla 6**  
*Ejecución de las pruebas*

Nombre de la prueba y datos de entrada	Comandos ejecutados y resultado presentado visualmente

Primer archivo con vértice cercano sobre él y su caja ->cube.txt	<pre>\$ cargar cube.txt El objeto Cube ha sido cargado exitosamente desde el archivo cube.txt. Elapsed time: 0.0126361s Processing finished at Mon Oct 14 19:48:33 2024  \$ v_cercano 20 20 20 Cube El vertice 7 (1, 1, 1) del objeto Cube es el más cercano al punto (20, 20, 20), a una distancia de 32.909 Elapsed time: 0.000117781s Processing finished at Mon Oct 14 19:48:40 2024  \$ envolvente Cube La caja envolvente del objeto Cube se ha generado con el nombre env_Cube y se ha agregado a los objetos en memoria. Elapsed time: 0.000123781s Processing finished at Mon Oct 14 19:48:46 2024  \$ v_cercanos_caja Cube Los vertices del objeto Cube más cercanos a las esquinas de su caja envolvente son: Esquina          Vertice      Distancia 0 (-1,-1,-1)    0 (-1,-1,-1)    0 1 (1,-1,-1)     1 (1,-1,-1)    0 2 (1,1,-1)      3 (1,1,-1)     0 3 (-1,1,-1)     2 (-1,1,-1)    0 4 (-1,-1,1)     4 (-1,-1,1)    0 5 (1,-1,1)      5 (1,-1,1)     0 6 (1,1,1)        7 (1,1,1)      0 7 (-1,1,1)       6 (-1,1,1)     0 Elapsed time: 0.000161501s Processing finished at Mon Oct 14 19:48:51 2024</pre>
---	--

*Figura 52. Prueba #1. Elaboración propia*

<p>Segundo archivo con vértice cercano sobre él y su caja -&gt;dodecahedron.txt</p>	<pre>\$ cargar dodecahedron.txt El objeto Dodecahedron ha sido cargado exitosamente desde el archivo dodecahedron.txt. Elapsed time: 0.0085364s Processing finished at Mon Oct 14 19:52:44 2024  \$ v_cercano 10 10 10 Dodecahedron El vertice 0 (1, 1, 1) del objeto Dodecahedron es el más cercano al punto (10, 10, 10), a una distancia de 15.5885 Elapsed time: 0.000118382s Processing finished at Mon Oct 14 19:53:00 2024  \$ envolvente Dodecahedron La caja envolvente del objeto Dodecahedron se ha generado con el nombre env_Dodecahedron y se ha agregado a los objetos en memoria. Elapsed time: 0.000137745s Processing finished at Mon Oct 14 19:53:08 2024  \$ v_cercanos_caja Dodecahedron Los vertices del objeto Dodecahedron más cercanos a las esquinas de su caja envolvente son: Esquina           Vertice           Distancia 0 (-1.618,-1.618,-1.618)    7 (-1,-1,-1)      1.0704 1 1 (1.618,-1.618,-1.618)     3 (1,-1,-1)      1.07041 2 (1.618,1.618,-1.618)     1 (1,1,-1)      1.07041 3 (-1.618,1.618,-1.618)    5 (-1,1,-1)      1.07041 4 (-1.618,-1.618,1.618)    6 (-1,-1,1)      1.07041 5 (1.618,-1.618,1.618)     2 (1,-1,1)      1.07041 6 (1.618,1.618,1.618)      0 (1,1,1)       1.07041 7 (-1.618,1.618,1.618)     4 (-1,1,1)       1.07041 Elapsed time: 0.000276043s Processing finished at Mon Oct 14 19:53:17 2024</pre>
---	---

*Figura 53. Prueba #2. Elaboración propia*

<p>Tercer archivo con vértice cercano sobre él y su caja -&gt;icosahedron.txt</p>	<pre>\$ cargar icosahedron.txt El objeto Icosahedron ha sido cargado exitosamente desde el archivo icosahedron.txt. Elapsed time: 0.00803663s Processing finished at Mon Oct 14 19:54:58 2024  \$ v_cercano 20 20 20 Icosahedron El vertice 0 (0, 1, 1.618) del objeto Icosahedron es el más cercano al punto (20, 20, 20), a una distancia de 33 .1496 Elapsed time: 0.000122715s Processing finished at Mon Oct 14 19:55:11 2024  \$ envolvente Icosahedron La caja envolvente del objeto Icosahedron se ha generado con el nombre env_Icosahedron y se ha agregado a los objetos en memoria. Elapsed time: 0.000140579s Processing finished at Mon Oct 14 19:55:18 2024  \$ v_cercanos_caja Icosahedron Los vertices del objeto Icosahedron más cercanos a las esquinas de su caja envolvente son: Esquina      Vertice      Distancia 0 (-1.618,-1.618,-1.618)      3 (0,-1,-1.618)      1.7 3201 1 (1.618,-1.618,-1.618)      3 (0,-1,-1.618)      1.73 201 2 (1.618,1.618,-1.618)      1 (0,1,-1.618)      1.7320 1 3 (-1.618,1.618,-1.618)      1 (0,1,-1.618)      1.732 01 4 (-1.618,-1.618,1.618)      2 (0,-1,1.618)      1.732 01 5 (1.618,-1.618,1.618)      2 (0,-1,1.618)      1.7320 1 6 (1.618,1.618,1.618)      0 (0,1,1.618)      1.73201 7 (-1.618,1.618,1.618)      0 (0,1,1.618)      1.73201 Elapsed time: 0.000175519s Processing finished at Mon Oct 14 19:55:27 2024</pre>
---	--

Figura 54. Prueba #3. Elaboración propia

<p>Cuarto archivo con vértice cercano sobre él y su caja -&gt;octahedron.txt</p>	<pre>\$ cargar octahedron.txt El objeto Octahedron ha sido cargado exitosamente desde el archivo octahedron.txt. Elapsed time: 0.00807566s Processing finished at Mon Oct 14 19:57:18 2024  \$ v_cercano 10 10 10 Octahedron El vertice 0 (1, 0, 0) del objeto Octahedron es el más cercano al punto (10, 10, 10), a una distancia de 16.7631 Elapsed time: 9.319e-05s Processing finished at Mon Oct 14 19:57:35 2024  \$ envolvente Octahedron La caja envolvente del objeto Octahedron se ha generado con el nombre env_Octahedron y se ha agregado a los objetos en memoria. Elapsed time: 0.000125416s Processing finished at Mon Oct 14 19:57:41 2024  \$ v_cercanos_caja Octahedron Los vertices del objeto Octahedron más cercanos a las esquinas de su caja envolvente son: Esquina      Vertice      Distancia 0 (-1,-1,-1)    1 (-1,0,0)    1.41421 1 (1,-1,-1)    0 (1,0,0)    1.41421 2 (1,1,-1)    0 (1,0,0)    1.41421 3 (-1,1,-1)    1 (-1,0,0)    1.41421 4 (-1,-1,1)    1 (-1,0,0)    1.41421 5 (1,-1,1)    0 (1,0,0)    1.41421 6 (1,1,1)    0 (1,0,0)    1.41421 7 (-1,1,1)    1 (-1,0,0)    1.41421 Elapsed time: 0.000183102s Processing finished at Mon Oct 14 19:57:51 2024</pre>
--	---

*Figura 55. Prueba #4. Elaboración propia*

<p>Calcular el vértice cercano general sobre 8 objetos en memoria donde 4 con mallas y 4 son envolventes</p> <p>-&gt;cube.txt  -&gt;dodecahedron.txt  -&gt;icosahedron.txt  -&gt;octahedron.txt</p>	<pre>\$ listado Hay 8 objetos en memoria: Cube contiene 8 vertices, 18 aristas y 12 caras. env_Cube contiene 8 vertices, 12 aristas y 6 caras. Dodecahedron contiene 20 vertices, 38 aristas y 14 caras . env_Dodecahedron contiene 8 vertices, 12 aristas y 6 caras. Icosahedron contiene 12 vertices, 30 aristas y 21 caras. env_Icosahedron contiene 8 vertices, 12 aristas y 6 caras. Octahedron contiene 6 vertices, 12 aristas y 8 caras. env_Octahedron contiene 8 vertices, 12 aristas y 6 caras . Elapsed time: 8.9303e-05 Processing finished at Mon Oct 14 20:01:39 2024  \$ v_cercano 15 15 15 El vertice 6 (1.618, 1.618, 1.618) del objeto env_Dodecahedron es el más cercano al punto (15, 15, 15), a una distancia de 23.1783 Elapsed time: 0.000154487s Processing finished at Mon Oct 14 20:01:50 2024</pre>
---	--

Figura 56. Prueba #5. Elaboración propia

<p>Quinto archivo con vértice cercano sobre él y su caja -&gt;pyramid.txt</p>	<pre>\$ cargar_pyramid.txt El objeto Pyramid ha sido cargado exitosamente desde el archivo pyramid.txt. Elapsed time: 0.00783316s Processing finished at Mon Oct 14 20:03:47 2024  \$ v_cercano 20 20 20 Pyramid El vertice 1 (1, 0, 1) del objeto Pyramid es el más cerc ano al punto (20, 20, 20), a una distancia de 33.4963 Elapsed time: 0.000106449s Processing finished at Mon Oct 14 20:04:02 2024  \$ envolvente Pyramid La caja envolvente del objeto Pyramid se ha generado con el nombre env_Pyramid y se ha agregado a los objetos en memoria. Elapsed time: 0.000136929s Processing finished at Mon Oct 14 20:04:07 2024  \$ v_cercanos_caja Pyramid Los vertices del objeto Pyramid más cercanos a las esqui nas de su caja envolvente son: Esquina      Vertice      Distancia 0 (-1,0,-1)   3 (-1,0,-1)   0 1 (1,0,-1)    2 (1,0,-1)   0 2 (1,1,-1)    2 (1,0,-1)   1 3 (-1,1,-1)   3 (-1,0,-1)   1 4 (-1,0,1)    4 (-1,0,1)   0 5 (1,0,1)     1 (1,0,1)    0 6 (1,1,1)     1 (1,0,1)    1 7 (-1,1,1)    4 (-1,0,1)   1 Elapsed time: 0.000160862s Processing finished at Mon Oct 14 20:04:14 2024</pre>
---	---

*Figura 57. Prueba #6. Elaboración propia*

<p>Sexto archivo con vértice cercano sobre él y su caja</p> <p>-&gt;rectangular_prism.txt</p>	<pre>\$ cargar rectangular_prism.txt El objeto Rectangular_Prism ha sido cargado exitosamente desde el archivo rectangular_prism.txt. Elapsed time: 0.00744436s Processing finished at Mon Oct 14 20:05:36 2024  \$ v_cercano 10 10 10 Rectangular_Prism El vertice 6 (10, 1, 1) del objeto Rectangular_Prism es el más cercano al punto (10, 10, 10), a una distancia de 12.7279 Elapsed time: 9.755e-05s Processing finished at Mon Oct 14 20:05:53 2024  \$ envolvente Rectangular_Prism La caja envolvente del objeto Rectangular_Prism se ha ge- nerado con el nombre env_Rectangular_Prism y se ha agreg- ado a los objetos en memoria. Elapsed time: 0.000131443s Processing finished at Mon Oct 14 20:06:03 2024  \$ v_cercanos_caja Rectangular_Prism Los vertices del objeto Rectangular_Prism más cercanos a las esquinas de su caja envolvente son: Esquina      Vertice      Distancia 0 (0,0,0)    0 (0,0,0)    0 1 (10,0,0)   1 (10,0,0)   0 2 (10,1,0)   2 (10,1,0)   0 3 (0,1,0)    3 (0,1,0)    0 4 (0,0,1)    4 (0,0,1)    0 5 (10,0,1)   5 (10,0,1)   0 6 (10,1,1)   6 (10,1,1)   0 7 (0,1,1)    7 (0,1,1)    0 Elapsed time: 0.000170352s Processing finished at Mon Oct 14 20:06:20 2024</pre>
---	--

*Figura 58. Prueba #7. Elaboración propia*

<p>Séptimo archivo con vértice cercano sobre él y su caja</p> <p>-&gt;rotated_rectangular_prism.txt</p>	<pre>\$ cargar rotated_rectangular_prism.txt El objeto Rotated_Rectangular_Prism ha sido cargado exitosamente desde el archivo rotated_rectangular_prism.txt. Elapsed time: 0.00860402s Processing finished at Mon Oct 14 20:07:49 2024  \$ v_cercano 20 20 20 Rotated_Rectangular_Prism El vertice 6 (0.5, 0.1, 0.5) del objeto Rotated_Rectangular_Prism es el más cercano al punto (20, 20, 20), a una distancia de 34.0075 Elapsed time: 0.000103253s Processing finished at Mon Oct 14 20:08:07 2024  \$ envolvente Rotated_Rectangular_Prism La caja envolvente del objeto Rotated_Rectangular_Prism se ha generado con el nombre env_Rotated_Rectangular_Prism y se ha agregado a los objetos en memoria. Elapsed time: 0.000135321s Processing finished at Mon Oct 14 20:08:25 2024  \$ v_cercanos_caja Rotated_Rectangular_Prism Los vertices del objeto Rotated_Rectangular_Prism más cercanos a las esquinas de su caja envolvente son: Esquina      Vertice      Distancia 0 (-8.66,-1.1,-8.66)      0 (-8.66,-1.1,-0.5)      8.1 6 1 (0.5,-1.1,-8.66)      1 (0.5,-1.1,-8.66)      0 2 (0.5,0.1,-8.66)      2 (0.5,0.1,-8.66)      0 3 (-8.66,0.1,-8.66)      3 (-8.66,0.1,-0.5)      8.16 4 (-8.66,-1.1,0.5)      4 (-8.66,-1.1,0.5)      0 5 (0.5,-1.1,0.5)      5 (0.5,-1.1,0.5)      0 6 (0.5,0.1,0.5)      6 (0.5,0.1,0.5)      0 7 (-8.66,0.1,0.5)      7 (-8.66,0.1,0.5)      0 Elapsed time: 0.000237409s Processing finished at Mon Oct 14 20:08:39 2024</pre>
---	--

*Figura 59. Prueba #8. Elaboración propia*

Octavo archivo con vértice cercano sobre él y su caja  ->tetrahedron.txt	<pre>\$ cargar tetrahedron.txt El objeto Tetrahedron ha sido cargado exitosamente desde el archivo tetrahedron.txt. Elapsed time: 0.00766394s Processing finished at Mon Oct 14 20:09:55 2024  \$ v_cercano 10 10 10 Tetrahedron El vertice 0 (1, 1, 1) del objeto Tetrahedron es el más cercano al punto (10, 10, 10), a una distancia de 15.5885 Elapsed time: 0.000110839s Processing finished at Mon Oct 14 20:10:07 2024  \$ envolvente Tetrahedron La caja envolvente del objeto Tetrahedron se ha generado con el nombre env_Tetrahedron y se ha agregado a los objetos en memoria. Elapsed time: 0.000132934s Processing finished at Mon Oct 14 20:10:20 2024  \$ v_cercanos_caja Tetrahedron Los vertices del objeto Tetrahedron más cercanos a las esquinas de su caja envolvente son: Esquina      Vertice      Distancia 0 (-1,-1,-1)    1 (-1,-1,1)    2 1 (1,-1,-1)    3 (1,-1,-1)    0 2 (1,1,-1)     0 (1,1,1)     2 3 (-1,1,-1)    2 (-1,1,-1)    0 4 (-1,-1,1)    1 (-1,-1,1)    0 5 (1,-1,1)     0 (1,1,1)     2 6 (1,1,1)       0 (1,1,1)     0 7 (-1,1,1)     0 (1,1,1)     2 Elapsed time: 0.000163317s Processing finished at Mon Oct 14 20:10:31 2024</pre>
--	---

*Figura 60. Prueba #9. Elaboración propia*

Calcular el vértice cercano general sobre 16 objetos en memoria donde 8 con mallas y 8 son envolventes  ->cube.txt ->dodecahedron.txt ->icosahedron.txt ->octahedron.txt ->pyramid.txt ->rectangular_prism.txt ->rotated_rectangular_prism.txt ->tetrahedron.txt	<pre>\$ listado Hay 16 objetos en memoria: Cube contiene 8 vertices, 18 aristas y 12 caras. env_Cube contiene 8 vertices, 12 aristas y 6 caras. Dodecahedron contiene 20 vertices, 38 aristas y 14 caras . env_Dodecahedron contiene 8 vertices, 12 aristas y 6 caras. Icosahedron contiene 12 vertices, 30 aristas y 21 caras. env_Icosahedron contiene 8 vertices, 12 aristas y 6 caras. Octahedron contiene 6 vertices, 12 aristas y 8 caras. env_Octahedron contiene 8 vertices, 12 aristas y 6 caras . Pyramid contiene 6 vertices, 13 aristas y 9 caras. env_Pyramid contiene 8 vertices, 12 aristas y 6 caras. Rectangular_Prism contiene 8 vertices, 11 aristas y 6 caras. env_Rectangular_Prism contiene 8 vertices, 12 aristas y 6 caras. Rotated_Rectangular_Prism contiene 8 vertices, 11 aristas y 6 caras. env_Rotated_Rectangular_Prism contiene 8 vertices, 12 aristas y 6 caras. Tetrahedron contiene 4 vertices, 5 aristas y 4 caras. env_Tetrahedron contiene 8 vertices, 12 aristas y 6 caras. Elapsed time: 9.2232e-05s Processing finished at Mon Oct 14 20:12:21 2024  \$ v_cercano 30 30 30 El vertice 6 (10, 1, 1) del objeto Rectangular_Prism es el más cercano al punto (30, 30, 30), a una distancia de 45.6289 Elapsed time: 0.000136735s Processing finished at Mon Oct 14 20:12:31 2024</pre>
---	---

Figura 61. Prueba #10. Elaboración propia

Nota: La tabla permite ver la ejecución de los casos de prueba para evaluar el componente 2. Para la elaboración de la columna “comandos ejecutados” se asume que ya se tiene el compilado ejecutado. Elaboración propia.

Es así como concluye nuestro plan de pruebas del componente 2, donde se puede evidenciar visualmente a lo largo de las 10 pruebas como el componente funciona correctamente y con calidad, demostrando que el acople los últimos 3 comandos fue exitoso respecto a lo desarrollado en el primer componente.

### 3.4 Corrección Componente 2

En la sustentación se identificó un problema en las funcionalidades del componente:

1. **El comando vCercanos\_caja:** El componente no permitía calcular los vértices cercanos de ese objeto a su caja envolvente si la caja no se había calculado previamente y la salida no se veía estéticamente atractiva. Esto afectaba directamente el funcionamiento del sistema, ya que la imposibilidad de ejecutar el comando sin hacer una acción previa impedía la prueba de esta funcionalidad además de que, al mostrarse desorganizada su salida no era posible ver con rapidez que efectivamente el comando había sido implementado correctamente.

Para corregir estos problemas, se realizaron las siguientes modificaciones en el código:

1. **Modificación de la función vCercanos\_caja en el TAD: Sistema:** Esta corrección, presentada en la figura 62, se implementó para modificar la salida de la función para que hiciera uso de espacios predefinidos gracias a la función “setw()” de la librería “iomanip”. Su uso se refleja directamente en la salida mostrada al usuario de la funcionalidad de vértices cercanos de un objeto y su caja envolvente, donde, en varias salidas (cout’s) se agregan estos espacios predefinidos para mejorar la estética y lectura de la salida, lo que resuelve la una parte del problema.

```
void Sistema::vCercano_caja(string nombre) { //Recibe nombre de objeto y calcula vertice mas cercanos a las
    Malla m = buscarMalla(&nombre);
    if (m.igual(Malla())) {
        cout << "El objeto " << nombre << " no ha sido cargado en memoria." << endl;
    } else {

        vector<float> valores;
        Malla caja = m.getDirecEnvolvente();
        map<int, Vertice> vCaja = caja->getVertices();
        map<int, Vertice> vertices = m.getVertices();

        cout << "Los vertices del objeto Cube mas cercanos a las esquinas de su caja envolvente son:" << endl;
        cout << setw(6) << "Esquina" << setw(12) << "Vertice" << setw(12) << "Distancia" << endl;
        cout << "-----" << endl;

        for (int i = 0; i < 8; i++) {
            valores = m.vCercano(vCaja[i]);

            // Print "Esquina" with separate 'setw' for ID and coordinates
            cout << setw(6) << vCaja[i].getId();
            << "(" << setw(3) << vCaja[i].getx() << ",";
            << setw(3) << vCaja[i].gety() << ",";
            << setw(3) << vCaja[i].getz() << ")";

            // Print "Vertice" with separate 'setw' for ID and coordinates
            cout << setw(10) << valores[0];
            << "(" << setw(3) << vertices[valores[0]].getx() << ",";
            << setw(3) << vertices[valores[0]].gety() << ",";
            << setw(3) << vertices[valores[0]].getz() << ")";

            // Print "Distancia" with 'setw' for alignment
            cout << setw(10) << valores[1] << endl;

            valores.clear();
        }

        cout << "-----" << endl;
    }
}
```

Figura 62. Función vCercanos\_caja. Elaboración propia

**2. Modificación del programa principal para agregar otra verificación a partir de los parámetros de entrada:** Esta corrección, presentada en la figura 63, se implementó para realizar una verificación extra a partir de los parámetros de entrada antes de hacer el llamado a la funcionalidad del sistema de los vértices cercanos de un objeto a su envolvente, comprobando a partir del parámetro recibido si ya se ha calculado su caja previamente y de no ser así calcularla en ese instante gracias a la funcionalidad *envolvente* de un objeto, de forma que después al hacer el llamado a la función previamente presentada esta funcionara así el usuario no haya calculado la caja envolvente previamente terminando de resolver el problema presentado.

```

} else if(cin[0] == comandos[0]) {    //vertices cercanos a la caja delimitada por los valores de los ejes
    if (cin.size() == 2) {
        if(sistema.buscarMalla(&cin[1]).direcEnvolvente == nullptr){
            sistema.envolvente(&cin[1]);
        }
        sistema.vCercano_caja(&cin[1]); //Recibe el nombre de la caja delimitadora y encuntra los vertices dentro de ella
    } else {
        cout << "El comando: 'v-cercanos-caja' requiere menos o mas parametros, revise el comando a ejecutar consultando la guia con el comando: 'ayuda'.\n";
    }
}

```

Figura 63. Verificación extra a partir de los parámetros de entrada. Elaboración propia

Para comprobar que los cambios efectuados solucionaron el problema encontrado fue diseñado y ejecutado un nuevo plan de pruebas; por lo que se creó una tabla que presentara el nombre de la prueba, el resultado esperado y los datos de entrada, la justificación de esta y como se va a ejecutar, enfocada en demostrar que la funcionalidad y calidad del componente es la adecuada.

Teniendo en cuenta el problema presentado nos pareció preciso que los casos de prueba consistieran en comprobar que el sistema fuera capaz de ejecutar el comando *vCercanos\_caja* para un objeto previamente cargado sin previamente haber calculado la caja envolvente del mismo y visualizar una salida estética que permita entender su correcto funcionamiento a primera vista, lo que se ve reflejado en la tabla 7.

La estrategia que tenemos la intención de utilizar es: primero descargar 4 archivos de texto (representantes cada uno de un objeto 3D) y cargarlos en el sistema para posteriormente ejecutar el comando ‘vCercanos\_caja’ para cada uno de ellos para así ejecutar 4 casos de prueba diferentes donde se evidencie la correcta resolución del problema.

**Tabla 7**  
*Plan de pruebas*

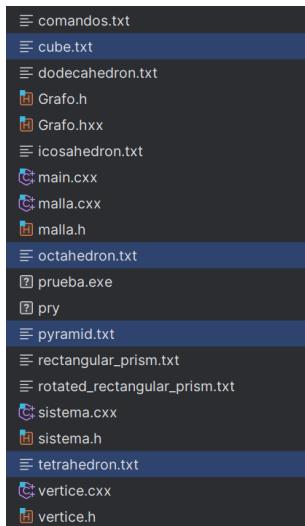
Nombre de la prueba	Resultado esperado y datos de entrada	Justificación	Cómo se ejecutará

Carga del primer archivo y ejecución del comando.	Salida estética y sin problemas ->Archivo de texto del objeto 3D	Evaluar la funcionalidad y calidad del componente al ejecutar el comando que fallaba.	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar el siguiente comando:  \$ cargar archivo.txt \$ v_cercanos_caja obj
Carga del segundo archivo y ejecución del comando.	Salida estética y sin problemas ->Archivo de texto del objeto 3D	Evaluar la funcionalidad y calidad del componente al ejecutar el comando que fallaba.	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar el siguiente comando:  \$ cargar archivo.txt \$ v_cercanos_caja obj
Carga del tercer archivo y ejecución del comando.	Salida estética y sin problemas ->Archivo de texto del objeto 3D	Evaluar la funcionalidad y calidad del componente al ejecutar el comando que fallaba.	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar el siguiente comando:  \$ cargar archivo.txt \$ v_cercanos_caja obj
Carga del cuarto archivo y ejecución del comando.	Salida estética y sin problemas ->Archivo de texto del objeto 3D	Evaluar la funcionalidad y calidad del componente al ejecutar el comando que fallaba.	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar el siguiente comando:  \$ cargar archivo.txt \$ v_cercanos_caja obj

*NOTA: La tabla permite ver la planeación de casos de prueba para evaluar las correcciones del componente 2, elaboración propia.*

### Ejecución de Pruebas

Siguiendo la estrategia presentada en la planeación de pruebas se hizo la respectiva descarga de archivos de forma que quedaron los archivos presentados en la Figura 64 resaltados en azul.



*Figura 64. Descarga de los dos archivos a probar. Elaboración propia*

Ahora, para realizar ejecutar cada prueba se seguirá el orden de comandos presentado en la columna “Como se ejecutará”, y para documentar su correcto funcionamiento se presentará en una tabla el nombre de la prueba y sus datos de entrada, los comandos ejecutados, y el resultado visual en consola, esta se presentará a continuación en la Tabla 8.

**Tabla 8**  
*Ejecución de las pruebas*

Nombre de la prueba y datos de entrada	Comandos ejecutados	Resultado presentado visualmente

Carga del cuarto archivo y ejecución del comando. ->cube.txt	\$ cargar cube.txt \$ v_cercanos_caja Cube	<pre>\$ cargar cube.txt El objeto Cube ha sido cargado exitosamente desde el archivo cube.txt. Elapsed time: 0.0107509s Processing finished at Sat Nov 16 18:54:46 2024  \$ v_cercanos_caja Cube La caja envolvente del objeto Cube se ha generado con el nombre env_Cube y se ha agregado a los objetos en memoria. Los vertices del objeto Cube mas cercanos a las esquinas de su caja envol- ente son: Esquina           Vertice           Distancia ----- 0( -1, -1, -1)   0( -1, -1, -1)   0 1( 1, -1, -1)    1( 1, -1, -1)   0 2( 1, 1, -1)     3( 1, 1, -1)   0 3( -1, 1, -1)    2( -1, 1, -1)   0 4( -1, -1, 1)    4( -1, -1, 1)   0 5( 1, -1, 1)     5( 1, -1, 1)   0 6( 1, 1, 1)      7( 1, 1, 1)   0 7( -1, 1, 1)     6( -1, 1, 1)   0 ----- Elapsed time: 0.000567852s Processing finished at Sat Nov 16 18:54:52 2024</pre>
Carga del cuarto archivo y ejecución del comando. ->octahedron.txt	\$ cargar octahedron.txt \$ v_cercanos_caja Octahedron	<pre>\$ cargar octahedron.txt El objeto Octahedron ha sido cargado exitosamente desde el archivo octahedron.txt. Elapsed time: 0.0213758s Processing finished at Sat Nov 16 18:59:15 2024  \$ v_cercanos_caja Octahedron La caja envolvente del objeto Octahedron se ha generado con el nombre env_Octahedron y se ha agregado a los objetos en memoria. Los vertices del objeto Cube mas cercanos a las esquinas de su caja envolvente son: Esquina           Vertice           Distancia ----- 0( -1, -1, -1)   1( -1, 0, 0)   1.41421 1( 1, -1, -1)    0( 1, 0, 0)   1.41421 2( 1, 1, -1)     0( 1, 0, 0)   1.41421 3( -1, 1, -1)    1( -1, 0, 0)   1.41421 4( -1, -1, 1)    1( -1, 0, 0)   1.41421 5( 1, -1, 1)     0( 1, 0, 0)   1.41421 6( 1, 1, 1)      0( 1, 0, 0)   1.41421 7( -1, 1, 1)     1( -1, 0, 0)   1.41421 ----- Elapsed time: 0.000306395s Processing finished at Sat Nov 16 18:59:29 2024</pre>
Carga del cuarto archivo y ejecución del comando. ->pyramid.txt	\$ cargar pyramid.txt \$ v_cercanos_caja Pyramid	<pre>\$ cargar pyramid.txt El objeto Pyramid ha sido cargado exitosamente desde el archivo pyramid.txt. Elapsed time: 0.0190524s Processing finished at Sat Nov 16 18:59:57 2024  \$ v_cercanos_caja Pyramid La caja envolvente del objeto Pyramid se ha generado con el nombre env_Pyramid y se ha agregado a los objetos en memoria. Los vertices del objeto Cube mas cercanos a las esquinas de su caja envolvente son: Esquina           Vertice           Distancia ----- 0( -1, 0, -1)    3( -1, 0, -1)   0 1( 1, 0, -1)     2( 1, 0, -1)   0 2( 1, 1, -1)     2( 1, 0, -1)   1 3( -1, 1, -1)    3( -1, 0, -1)   1 4( -1, 0, 1)     4( -1, 0, 1)   0 5( 1, 0, 1)      1( 1, 0, 1)   0 6( 1, 1, 1)      1( 1, 0, 1)   1 7( -1, 1, 1)     4( -1, 0, 1)   1 ----- Elapsed time: 0.000307075s Processing finished at Sat Nov 16 19:00:03 2024</pre>

Figura 65. Prueba #1. Elaboración propia

Figura 66. Prueba #2. Elaboración propia

Figura 67. Prueba #3. Elaboración propia

Carga del cuarto archivo y ejecución del comando.  ->te-trahedron.txt	\$ cargar tetrahedron.txt \$ v_cercanos_caja Tetrahedron	<pre>\$ cargar tetrahedron.txt El objeto Tetrahedron ha sido cargado exitosamente desde el archivo tetrahedron.txt. Elapsed time: 0.0214s Processing finished at Sat Nov 16 19:01:40 2024  \$ v_cercanos_caja Tetrahedron La caja envolvente del objeto Tetrahedron se ha generado con el nombre env_Tetrahedron y se ha agregado a los objetos en memoria. Los vértices del objeto Cube mas cercanos a las esquinas de su caja envolvente son: Esquina           Vértice           Distancia ----- 0( -1, -1, -1)   1( -1, -1, 1)   2 1( 1, -1, -1)    3( 1, -1, -1)   0 2( 1, 1, -1)     0( 1, 1, 1)     2 3( -1, 1, -1)    2( -1, 1, -1)   0 4( -1, -1, 1)    1( -1, -1, 1)   0 5( 1, -1, 1)     0( 1, 1, 1)     2 6( 1, 1, 1)       0( 1, 1, 1)     0 7( -1, 1, 1)     0( 1, 1, 1)     2 ----- Elapsed time: 0.000328703s Processing finished at Sat Nov 16 19:01:47 2024</pre>
---	---	---

Figura 68. Prueba #4. Elaboración propia

Nota: La tabla permite ver la ejecución de los casos de prueba para evaluar las correcciones del componente 2. Elaboración propia.

Es así como concluye nuestro plan de pruebas para las correcciones del componente 2, donde se puede evidenciar visualmente a lo largo de las 4 pruebas como el componente es funcional y estético, demostrando que las correcciones solo cambiaron detalles del TAD: Sistema (con la inclusión de los espacios setw()) y de la función principal (con la verificación extra) evitando el surgimiento de nuevos problemas, mejorando el componente comprobando que ahora cumple con los requisitos esperados.

#### 4 Componente 3: Rutas cortas y centroide

Este componente tiene como fin el desarrollo de funcionalidades principalmente enfocadas en la identificación de rutas entre los vértices de los objetos, por lo que al hacer uso de tantas características de las mallas poligonales (incluso de sus envolventes) permite evidenciar aún más usos del propio del sistema.

##### Objetivos de los comandos:

1. **Ruta corta que conecte dos vértices de un objeto:** Este comando permite al sistema determinar la ruta más corta desde un vértice a otro (identificados por su índice) dentro de un objeto particular, identificando en el proceso los vértices que conforman la ruta y la longitud a partir de la suma de la distancia euclíadiana entre ellos. Esta funcionalidad es fundamental para optimizar el rendimiento en aplicaciones de gráficos y modelado 3D, permitiendo una navegación eficiente dentro de estructuras complejas y minimizando el tiempo de procesamiento
2. **Ruta corta desde un vértice hacia el centroide de un objeto:** Este comando permite al sistema identificar la ruta más corta desde un vértice (identificado por su índice) hacia el centroide del objeto particular al que pertenece, que corresponde a las coordenadas promedio de todos los vértices del objeto,

identificando en el proceso los vértices que conforman la ruta y la longitud a partir de la suma de la distancia euclídea entre ellos. Esta funcionalidad puede ser crucial para aplicaciones en ingeniería y diseño, donde conocer la ruta más corta al centroide puede ayudar en la distribución de cargas, balanceo de estructuras y mejoras en la manufactura de piezas y componentes.

#### 4.1 Descripción de los Comandos

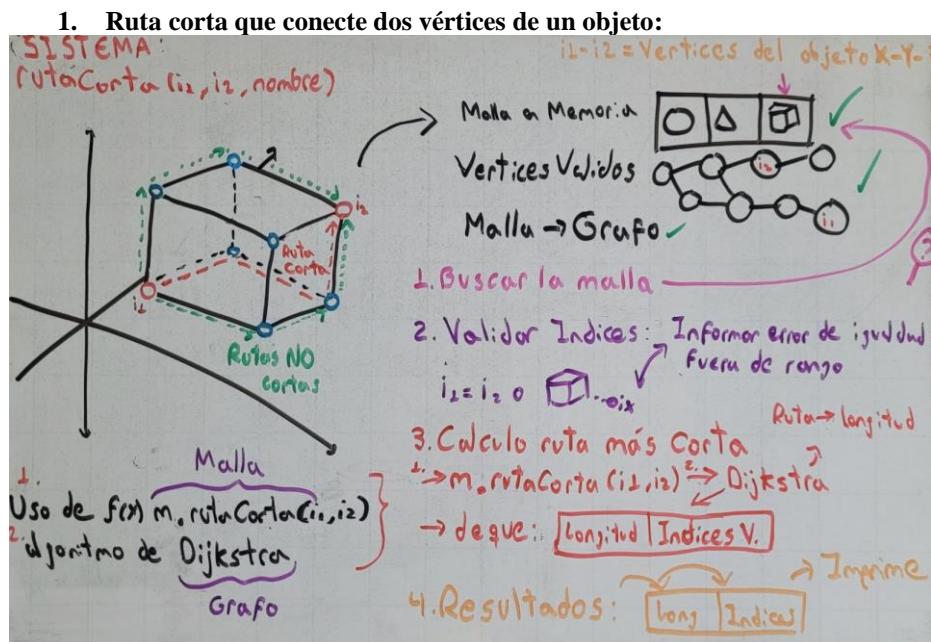


Figura 69. Diseño del comando ruta\_corta i1 i2 Objeto. Elaboración propia

En la Figura 69, podemos observar la idea que se plantea para realizar la acción requerida por el comando *ruta\_corta i1 i2 Objeto*.

**Descripción:** Este comando se implementa en tres partes: En el sistema, en la malla y en el grafo de la malla. En cuanto al sistema, la función a implementar calcula la ruta más corta entre dos vértices ( $i_1$  y  $i_2$ ) de un objeto almacenado en el sistema. Para lograrlo, utiliza el algoritmo de Dijkstra implementado en el TAD Grafo dentro del TAD Malla. La función valida que los vértices y la malla existan, y maneja los casos de error, informando mensajes claros al usuario. Si la ruta se encuentra correctamente, muestra la secuencia de vértices que conforman la ruta y su longitud total.

#### Requisitos previos:

- Malla cargada en memoria: La malla correspondiente al nombre ingresado debe haber sido previamente cargada en el sistema y su nombre debe coincidir exactamente con el que fue utilizado al cargarla en el sistema.

- Índices válidos: Los índices i1 e i2 deben estar dentro del rango de vértices de la malla.
- Malla y grafo configurados correctamente: La malla debe contener un grafo configurado con todas las aristas y distancias calculadas y el grafo de la malla debe implementar el método dijkstra para calcular las rutas
- TAD Malla y TAD Grafo: Deben estar correctamente definidos, implementados y conectados.

**Proceso interno de la función correspondiente a este comando:**

1. Búsqueda de la malla:
  - i. Usa buscarMalla(nombre) para encontrar la malla con el nombre proporcionado.
  - ii. Si no se encuentra la malla, informa al usuario: "El objeto [nombre] no ha sido cargado en memoria."
2. Validación de índices:
  - i. Si i1 e i2 son iguales, informa al usuario: "Los índices de los vértices dados son iguales."
  - ii. Si i1 o i2 están fuera del rango de vértices de la malla, informa al usuario: "Algunos de los índices de vértices están fuera de rango para el objeto [nombre]."
3. Cálculo de la ruta más corta:
  - i. Llama a la función rutaCorta(i1, i2) del TAD: Malla que, a su vez, utiliza la función dijkstra del TAD: Grafo para calcular la ruta más corta y la longitud de la misma.
  - ii. El resultado es un deque de numéricos que contiene:
    1. En la primera posición: la longitud total de la ruta.
    2. En las posiciones siguientes: los índices de los vértices de la ruta en orden.
4. Impresión del resultado:
  - i. Itera sobre el deque para mostrar los índices de los vértices que forman la ruta.
  - ii. Muestra la longitud total de la ruta: "La ruta más corta que conecta los vértices [i1] y [i2] del objeto [nombre] pasa por: [ruta]; con una longitud de [distancia]."

**Resultados e Interpretación:**

- a. (Objeto no encontrado): Si el mensaje "El objeto [nombre] no ha sido cargado en memoria" aparece, esto indica que el objeto con el nombre dado no ha sido cargado en el sistema. Debe verificarse si el nombre es correcto y si la malla ha sido previamente cargada.
- b. (Índices iguales): El mensaje "Los índices de los vértices dados son iguales" implica que no tiene sentido buscar una ruta entre el mismo vértice.

- c. (Índices fuera de rango): El mensaje " Algunos de los índices de vértices están fuera de rango para el objeto [nombre]" implica que uno o ambos índices no existen en la malla por lo que debe asegurarse de que están en el rango.
- d. (Resultado exitoso): El mensaje " La ruta más corta que conecta los vértices [i1] y [i2] del objeto [nombre] pasa por: [i1, v1, v2, ..., i2]; con una longitud de [distancia]" implica que El sistema encontró la ruta más corta entre los vértices, incluyendo la secuencia de vértices intermedios y la distancia total.

#### Verificación de Ejecución Correcta:

1. Mensajes en Consola: La forma más directa de verificar que la función se ha ejecutado correctamente es revisar el mensaje mostrado en la consola. Cada posible escenario tiene un mensaje específico que indica el resultado de la operación.
2. Consistencia en rutas: Se puede comprobar que la ruta generada respete el orden esperado (de i1 a i2) y que la distancia total corresponda a la suma de las distancias de las aristas involucradas.

#### 2. Ruta corta que conecte dos vértices

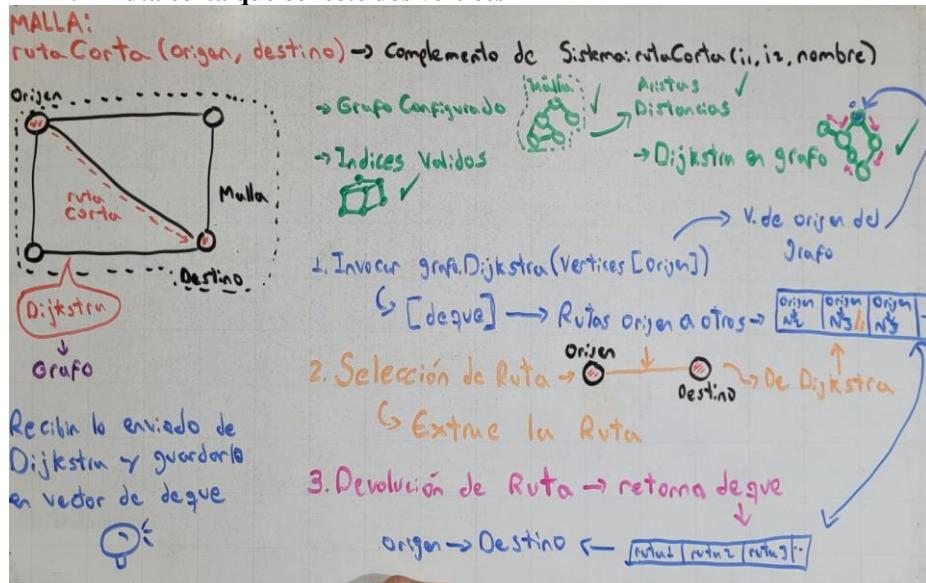


Figura 70. Diseño de la función `ruta_corta(i1, i2)`. Elaboración propia

En la Figura 70, podemos observar la idea que se plantea para realizar la acción requerida por la función `ruta_corta(i1, i2)` la cual es importante precisar pues como se pudo evidenciar en el proceso interno de la función correspondiente al primer comando, es a esta función del TAD: Malla a la que se refiere para encontrar la ruta corta entre dos vértices a través del grafo.

**Descripción:** Esta función es la segunda parte del primer comando (la de la malla) y tiene como propósito calcular la ruta más corta desde un vértice de origen hasta un vértice de destino dentro de una malla. Para ello, utiliza el algoritmo de Dijkstra implementado en el TAD Grafo. El resultado es un deque de valores numéricos que contiene la longitud total de la ruta en su primera posición, seguido de los índices de los vértices que forman la ruta, en orden.

#### Requisitos previos:

- Grafo correctamente configurado: La malla debe tener un grafo interno con todas las aristas y distancias configuradas correctamente.
- Índices válidos:  $i1(\text{origen})$  y  $i2(\text{destino})$  deben estar dentro del rango de índices de vértices de la malla.
- Algoritmo de Dijkstra implementado: La clase Grafo debe tener el método `dijkstra` que permita calcular rutas más cortas desde un vértice dado a los demás.

#### Proceso interno de la función:

1. Llamada al algoritmo de Dijkstra:
  - i. Invoca `grafo.dijkstra(vertices[origen])`, donde `vertices[origen]` representa el vértice de origen en el grafo.
  - ii. Esto devuelve un vector de deque<float> que contiene las rutas desde el vértice de origen hacia todos los demás vértices del grafo.
2. Selección de la ruta:
  - i. Extrae la ruta desde el vértice de origen hasta el vértice de destino del vector devuelto por `dijkstra`.
3. Devolución de la ruta:
  - i. Retorna el deque<float> correspondiente a la ruta entre los vértices origen y destino.

#### Resultados e Interpretación:

- a. (Ruta válida): La función devuelve una deque con dos elementos: La distancia total en la primera posición y Los índices de los vértices de la ruta, en orden, desde el origen hasta el destino, de la cual se puede interpretar que se encontró una ruta entre los vértices, y la distancia total corresponde a la suma de las distancias de las aristas de la ruta.
- b. (Ruta no válida): Si no existe una conexión entre los vértices origen y destino, el deque estará vacío. Esto indicaría que no hay ruta disponible entre los vértices en la topología actual del grafo.

#### Verificación de Ejecución Correcta:

1. Validación del resultado: Asegurarse de que el deque contenga al menos un elemento (la distancia total) y verificar que los índices en el deque formen una secuencia válida de vértices conectados en el grafo.

2. Comprobación en el grafo: Comprobar que la distancia total coincide con la suma de las aristas en la ruta

### 3. Algoritmo Dijkstra desde un vértice de origen

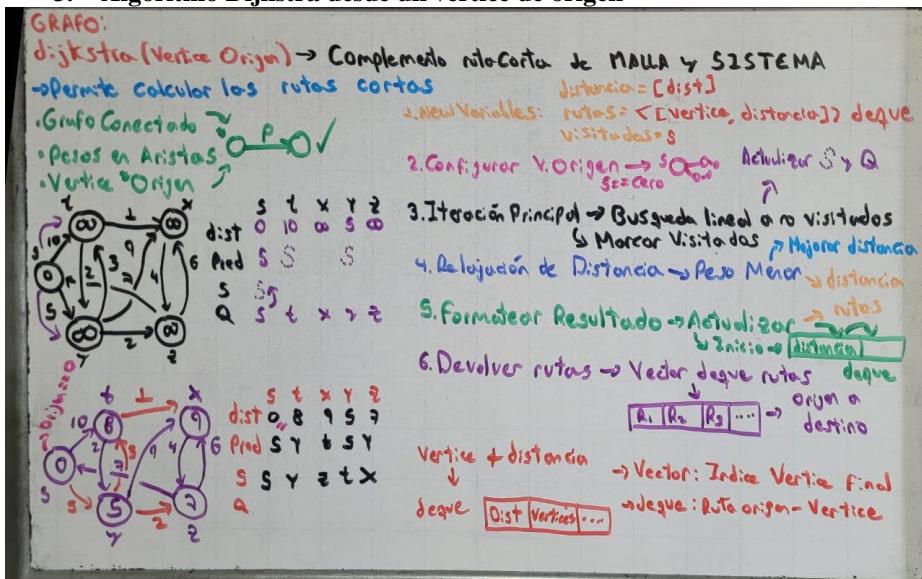


Figura 71. Diseño de la función `dijkstra(verticeOrigen)`. Elaboración propia

En la Figura 71, podemos observar la idea que se plantea para realizar la acción requerida por la función `dijkstra(verticeOrigen)` la cual es importante precisar pues como se pudo evidenciar en el proceso interno de la función correspondiente al primer comando, es a esta función del TAD: Grafo a la que se refiere para encontrar la ruta corta entre dos vértices a través del grafo.

**Descripción:** Esta función es la tercera parte del primer y segundo comando (la del grafo) y tiene como propósito implementar el algoritmo de Dijkstra para calcular las rutas más cortas desde un vértice de origen (origen) hacia todos los demás vértices en el grafo. Devuelve una deque, donde cada posición contiene: La distancia total desde el origen hasta el vértice correspondiente y la secuencia de vértices que conforman la ruta más corta

#### Requisitos previos:

- Grafo conectado: El grafo debe estar correctamente configurado, con todos los vértices y aristas inicializados.
- Pesos válidos en las aristas: Los pesos de las aristas deben ser valores positivos (o representaciones válidas para algoritmos de grafos).
- Vértice de origen válido: El vértice de origen debe existir en el grafo.

**Proceso interno de la función:**

1. Inicialización de variables:
  - i. distancias: Vector con la distancia mínima desde el origen a cada vértice, inicializado con infinito.
  - ii. rutas: Vector de deque de valores numéricos para almacenar las rutas hacia cada vértice.
  - iii. visitados: Vector para marcar qué vértices ya han sido procesados.
2. Configuración del vértice de origen:
  - i. Calcula el índice del vértice de origen y establece su distancia en 0.
3. Iteración principal:
  - i. Encuentra el vértice no visitado con la distancia mínima (utilizando una búsqueda lineal).
  - ii. Marca el vértice como visitado.
4. Relajación de las aristas:
  - i. Para cada vecino del vértice actual, calcula si pasar por este vértice reduce la distancia al vecino.
  - ii. Si la distancia mejora, actualiza el valor en distancias y la ruta en rutas.
5. Formateo del resultado:
  - i. Para cada vértice, agrega la distancia total al inicio del deque correspondiente en rutas.
6. Devolución del resultado:
  - i. Retorna el vector de deque de valores numéricos con las rutas desde el origen hacia todos los demás vértices.

**Resultados e Interpretación:**

- a. (Ruta válida para todos los vértices): La función devuelve una deque con dos elementos: La distancia total en la primera posición y Los índices de los vértices de la ruta, en orden, desde el origen hasta el destino, de la cual se puede interpretar que el algoritmo ha encontrado rutas válidas hacia todos los vértices conectados al origen.
- b. (Vértices inaccesibles): Si un vértice no es accesible desde el origen, el deque correspondiente contendrá infinito en la primera posición y estará vacío después. Esto indicaría No hay conexión entre el origen y este vértice.

**Verificación de Ejecución Correcta:**

1. Validación del vector de rutas: Cada deque debe comenzar con la distancia total y contener una secuencia válida de índices de vértices.
2. Consistencia de distancias: Comprobar que la distancia total coincide con la suma de las aristas en la ruta

**4. Ruta corta desde un vértice hasta el centroide de su objeto:**

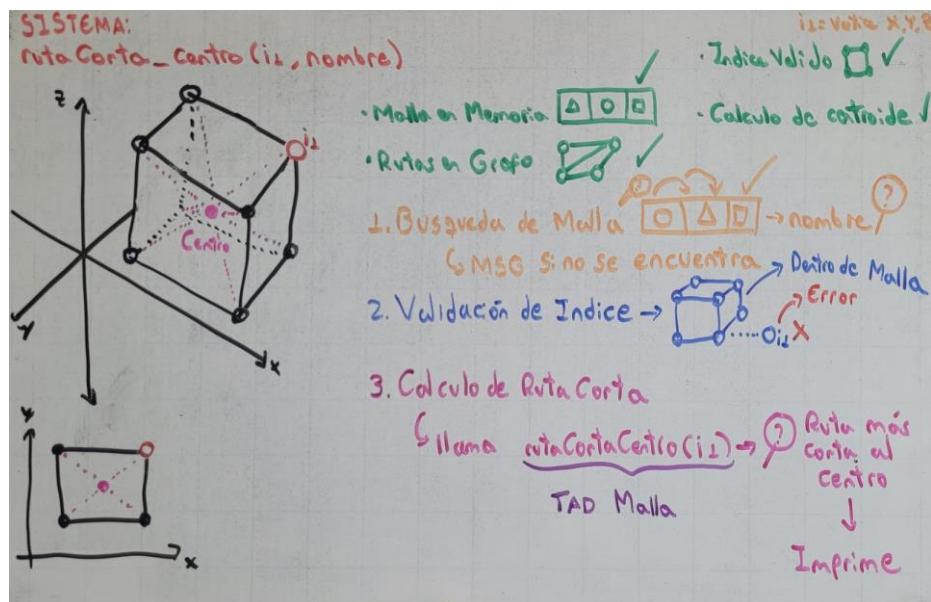


Figura 72. Diseño del comando ruta\_corta\_centro i1 objeto. Elaboración propia

En la Figura 72, podemos observar la idea que se plantea para realizar la acción requerida por el comando *ruta\_corta\_centro i1 objeto*.

**Descripción:** Este comando se implementa en tres partes: En el sistema, en la malla, y en el grafo. El sistema comienza buscando la malla en el sistema y si la encuentra identificando si el vértice se encuentra en ella para posteriormente, calcular y mostrar la ruta más corta desde el vértice (i1) especificado hasta el centro geométrico (centroide) del objeto (nombre). El centroide se calcula como el promedio de las coordenadas de todos los vértices del objeto y si la malla está cargada y el índice del vértice es válido, la función delega la tarea de cálculo al TAD: Malla que a su vez delega la tarea al TAD:Grafo.

#### Requisitos previos:

- Malla cargada en memoria: La malla especificada por el parámetro nombre debe existir en el sistema y haber sido cargada previamente.
- Índice válido: El índice i1 debe estar dentro del rango de índices de vértices de la malla.
- Cálculo correcto del centroide: La clase Malla debe implementar el cálculo del centroide y su conexión al grafo mediante una arista.
- Rutas en el grafo: La malla debe tener un grafo funcional que permita calcular rutas más cortas desde cualquier vértice al centroide.

#### Proceso interno de la función correspondiente a este comando:

1. Búsqueda de la malla:
  - i. Usa buscarMalla(nombre) para encontrar la malla con el nombre proporcionado.
  - ii. Si no se encuentra la malla, muestra: "El objeto [nombre] no ha sido cargado en memoria."
2. Validación del índice:
  - i. Si i1 está fuera del rango de vértices de la malla, muestra: "El índice de vértice está fuera de rango para el objeto [nombre]."
3. Cálculo de la ruta más corta:
  - i. Llama a rutaCortaCentro(i1) del TAD:Malla, que calcula el centroide, lo conecta al vértice más cercano y busca la ruta más corta hasta él.
  - ii. Muestra la información del centroide y la ruta obtenida.

#### **Resultados e Interpretación:**

- a. (Objeto no encontrado): Si se muestra el mensaje " El objeto [nombre] no ha sido cargado en memoria." Esto indica que la malla con el nombre proporcionado no está cargada en el sistema.
- b. (Índice fuera de rango): Si se muestra el mensaje " El índice de vértice está fuera de rango para el objeto [nombre]." Esto indica que El índice i1 no corresponde a ningún vértice de la malla. Verifica que i1 esté en el rango válido.
- c. (Resultado exitoso): Si se muestra el mensaje " La ruta más corta que conecta el vértice [i1] con el centro del objeto [nombre], ubicado en [ctx, cty, ctz], pasa por: [i1, v1, v2, ..., ct]; con una longitud de [valor\_distancia]." Esto indica que La ruta desde el vértice dado hasta el centroide se encontró exitosamente, mostrando los vértices intermedios y la distancia total.

#### **Verificación de Ejecución Correcta:**

1. Mensajes en Consola: Se revisa que los mensajes que se muestran en la consola para determinar si la función ha sido ejecutada correctamente.
2. Cálculo del centroide: Asegurarse de que las coordenadas del centroide sean el promedio de las coordenadas de los vértices del objeto.
3. Validación de la ruta: Comprueba que los índices de los vértices en la ruta sean válidos y conectados

#### **5. Ruta corta al centroide desde un vértice**

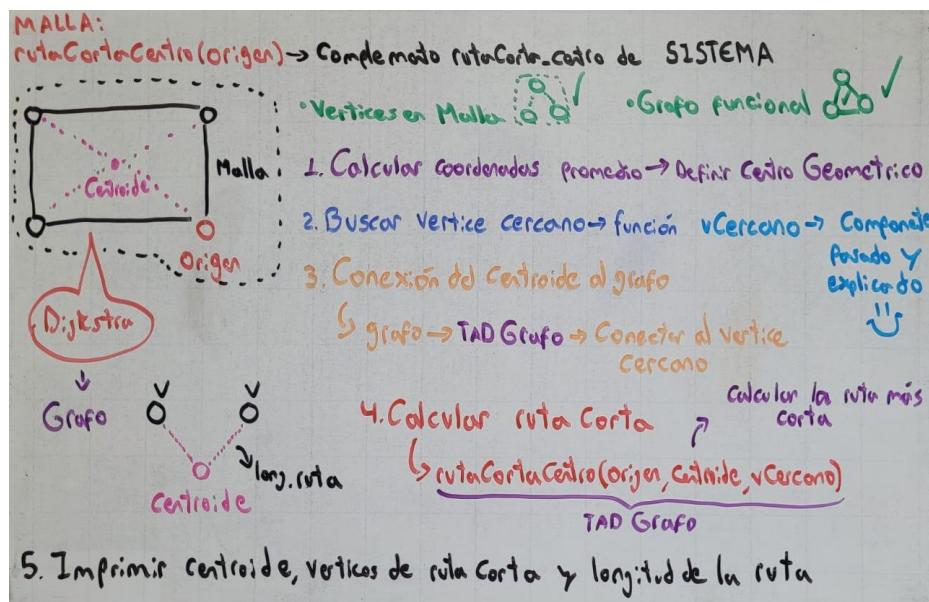


Figura 73. Diseño de la función rutaCortaCentro(verticeOrigen). Elaboración propia

En la Figura 73, podemos observar la idea que se plantea para realizar la acción requerida por la función  $\text{rutaCortaCentro}(\text{verticeOrigen})$  la cual es importante precisar pues como se pudo evidenciar en el proceso interno de la función correspondiente al segundo comando, es a esta función del TAD: Malla a la que se refiere para encontrar la ruta corta entre un vértice y el centroide a través del grafo.

**Descripción:** Esta función es la segunda parte del segundo comando (la de la malla) y tiene como propósito calcular la ruta más corta desde un vértice (origen) hasta el centroide del objeto, definido como el promedio de las coordenadas de todos los vértices. El centroide se conecta al vértice más cercano dentro del grafo mediante una arista. Luego, utiliza el algoritmo de Dijkstra del TAD:Grafo para encontrar la ruta más corta al centroide

#### Requisitos previos:

- Vértices en la malla: La malla debe contener vértices con coordenadas válidas para calcular el centroide.
- Grafo funcional: El grafo debe permitir calcular rutas más cortas entre vértices y soportar la adición temporal del centroide como vértice.

#### Proceso interno de la función:

1. Cálculo del centroide:
  - i. Calcula las coordenadas promedio de todos los vértices ( $x, y, z$ ) para definir el centro geométrico del objeto.
2. Búsqueda del vértice más cercano:

- i. Usa la función vCercano explicada en el pasado componente para identificar el vértice del objeto más cercano al centroide.
- 3. Conexión del centroide al grafo:
  - i. Pasa la información a la función rutaCortaCentro(origen, centroide, vCercano) del TAD: Grafo.
- 4. Cálculo de la ruta más corta:
  - i. Guarda el resultado pues ese es la ruta más corta desde origen hasta el centroide.
- 5. Impresión del resultado:
  - i. Muestra la ubicación del centroide, los vértices que conforman la ruta más corta, y la longitud total de la ruta.

**Resultados e Interpretación:**

- a. (Cálculo exitoso): Si se muestra el mensaje " La ruta más corta que conecta el vértice [origen] con el centro del objeto, ubicado en ([ctx, cty, ctz]), pasa por: [ruta]; con una longitud de [distancia]. " Esto indica el centroide fue calculado, conectado y la ruta se encontró correctamente.
- b. (Error en el cálculo): Si no se encuentra un vértice más cercano o si el grafo está mal configurado, no se podrá calcular la ruta.

**Verificación de Ejecución Correcta:**

1. Validación del centroide: Asegurarse de que las coordenadas calculadas sean efectivamente el promedio de los vértices.
2. Conexión al grafo: Comprueba que el centroide esté conectado al vértice más cercano y que la arista añadida tenga la distancia correcta
3. Ruta generada: Verifica que los vértices en la ruta correspondan al grafo extendido con el centroide.

**6. Ruta corta hacia el centroide**

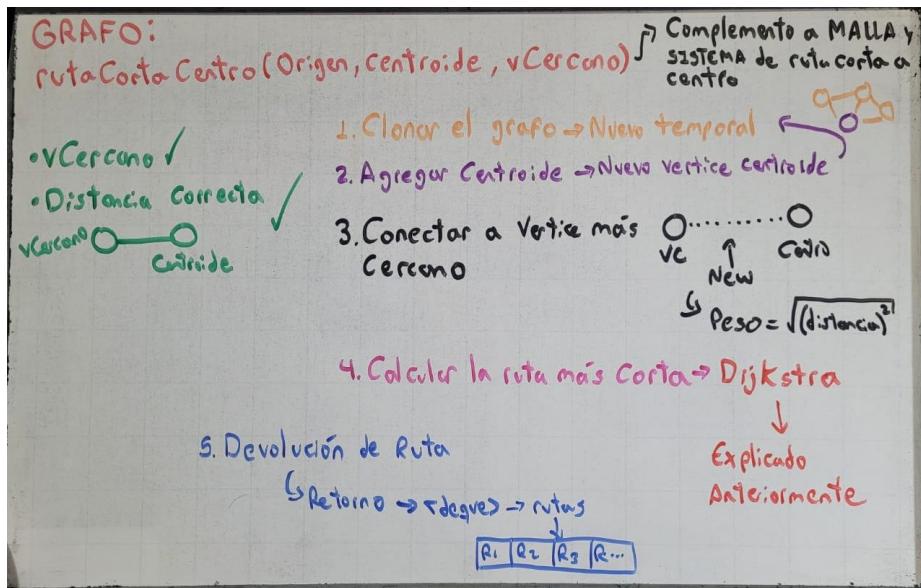


Figura 74. Diseño de la función rutaCortaCentro(verticeOrigen, centroide, verticeCercano). Elaboración propia

En la Figura 74, podemos observar la idea que se plantea para realizar la acción requerida por la función *rutaCortaCentro(verticeOrigen, centroide, verticeCercano)* la cual es importante precisar pues como se pudo evidenciar en el proceso interno de la función correspondiente al segundo comando, es a esta función del TAD: Grafo a la que se refiere para encontrar la ruta corta entre un vértice y el centroide a través del grafo.

**Descripción:** Esta función es la tercera parte del segundo comando (la del grafo) y tiene como propósito calcular la ruta más corta desde un vértice (origen) hasta un punto virtual (centroide) en el grafo, conectándolo temporalmente al vértice más cercano (vCercano). Hace uso del algoritmo de *Dijkstra* (ya explicado) para determinar la ruta hacia el centroide en el grafo modificado y de la función *indiceVertice(vertice)* que busca en el vector de vértices del grafo el índice del vértice solicitado.

#### Requisitos previos:

- Vértice más cercano identificado: El vértice más cercano al centroide (vCercano) debe estar correctamente calculado.
- Cálculo de distancia: La distancia entre el centroide y el vértice más cercano debe calcularse con precisión.

#### Proceso interno de la función:

1. Clonar el grafo:
  - i. Crea un grafo temporal como copia del grafo original.
2. Agregar el centroide:
  - i. Añade el vértice del centroide al grafo temporal.

3. Conectar al vértice más cercano:
  - i. Agrega una arista entre el centroide y el vértice más cercano, con un peso igual a la distancia euclídea.
4. Cálculo de la ruta más corta:
  - i. Usa dijkstra(origen) para calcular la ruta más corta hacia el centroide.
5. Devolución de la ruta:
  - i. Retorna el deque de valores numéricos correspondiente a la ruta más corta.

**Resultados e Interpretación:**

- a. (Ruta válida): La función devuelve una deque con dos elementos: La distancia total al centroide en la primera posición y los índices de los vértices en la ruta, en orden, de la cual se puede interpretar que La ruta más corta al centroide fue calculada correctamente.
- b. (Ruta no válida): Un deque vacío si hay problemas en la configuración del grafo o el cálculo del centroide.

**Verificación de Ejecución Correcta:**

1. Validación de conexiones: Asegúrate de que el centroide esté correctamente agregado y conectado en el grafo.
2. Resultados del algoritmo: Comprueba que la ruta al centroide sea consistente con la topología del grafo extendido
3. Distancia calculada: Verifica que la distancia total corresponda a la suma de las distancias de las aristas de la ruta

## 4.2 Diseño

A continuación, se muestra un bosquejo del diseño inicial (Figura 75) que se planteó para modelar el programa para el tercer componente:

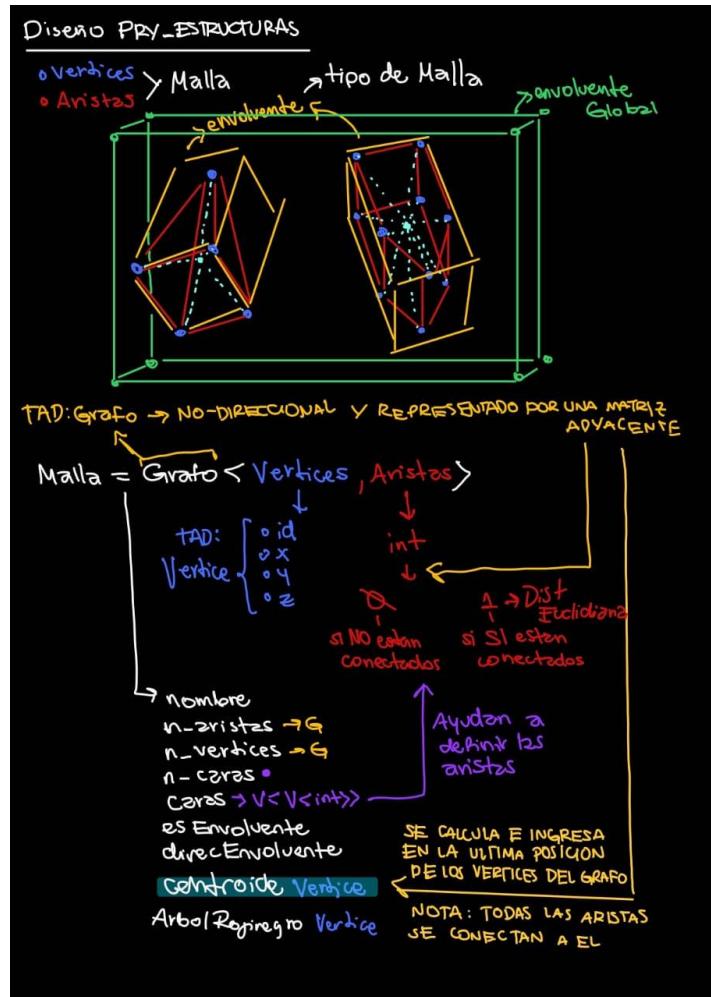


Figura 75. Diseño segundo componente. Elaboración propia

Con esta base planteada se procedió al diseño de TAD's para la correcta ejecución de los comandos.

#### Descripción de los Tipos Abstractos de Datos (TADs):

A continuación, se presenta la especificación de los cuatro TAD's creados para la realización de este proyecto, empezando con la especificación del TAD de Sistema en la Figura 76:

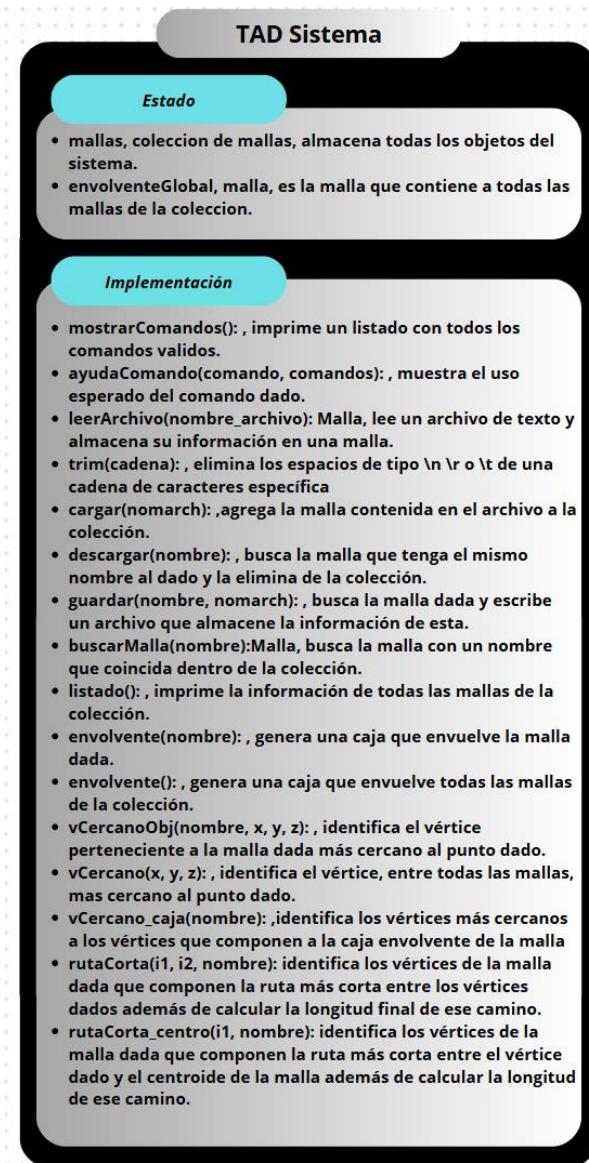


Figura 76. Especificación de sistema. Elaboración propia

Para el sistema se sigue planteando que contenga dos atributos: mallas, siendo una colección de mallas; y envolventeGlobal que contendría una caja que envuelva a todos los objetos de la colección. En cuanto a su implementación, se tendrán las 16 funciones que facilitarán el cumplimiento de las acciones requeridas por los comandos. Ahora se mostrará la especificación del TAD de malla en la Figura 77:

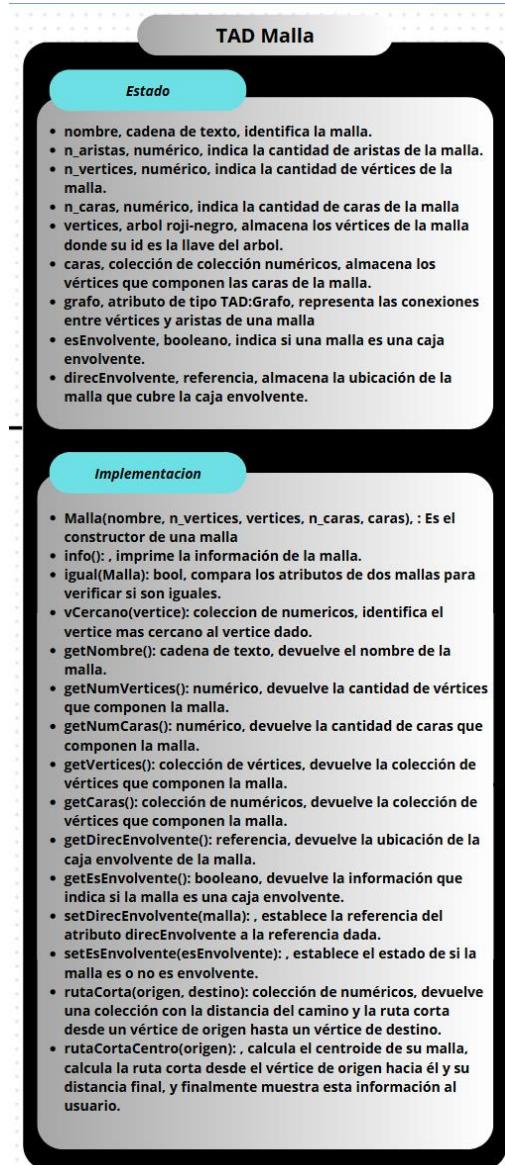


Figura 77. Especificación de malla. Elaboración propia

Para este TAD se considera la información que se recolecta de los archivos, de forma que cuando se lea un archivo de texto se pueda construir de la mejor forma una malla ahora guardando a los vértices en un árbol roji-negro y un grafo que nos permita representar la malla con sus vértices y aristas. La malla cuenta con 15 funciones que son llamadas por el sistema para ejecutar los comandos de las cuales destaca

rutaCortaCentro pues en ella se calcula el centroide de la malla. Ahora se mostrará la especificación del TAD de grafo en la Figura 78:

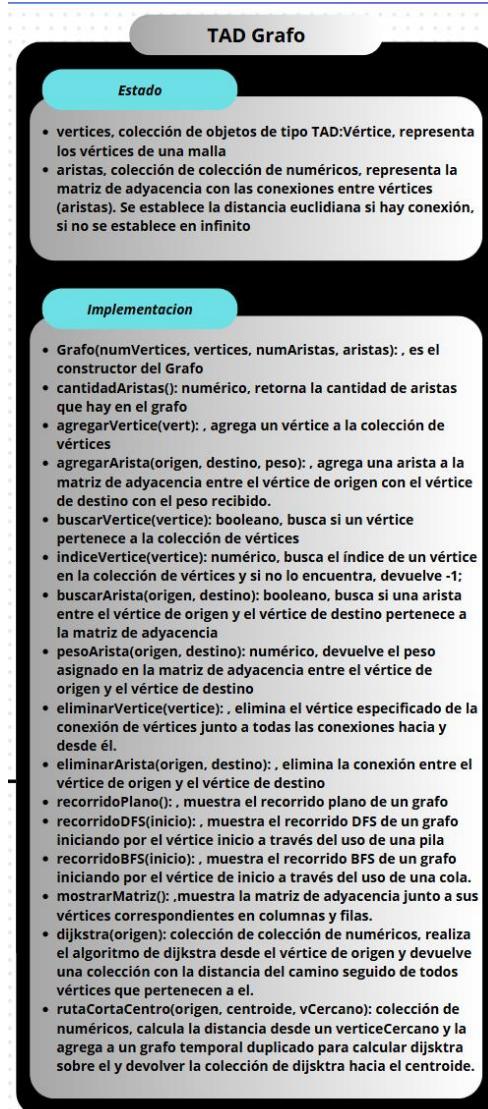


Figura 78. Especificación de grafo. Elaboración propia

Para este TAD se considera la implementación de un grafo a través de una matriz de adyacencia donde si existe conexión el peso de esta es la distancia euclíadiana entre esos dos vértices y si no se pone como infinito. El grafo cuenta con 16 funciones que son llamadas por el sistema para ejecutar los comandos, en especial el algoritmo

dijkstra y rutaCortaCentro para calcular la ruta hacia el centroide. Por último, se muestra el TAD de vértice en la Figura 79:

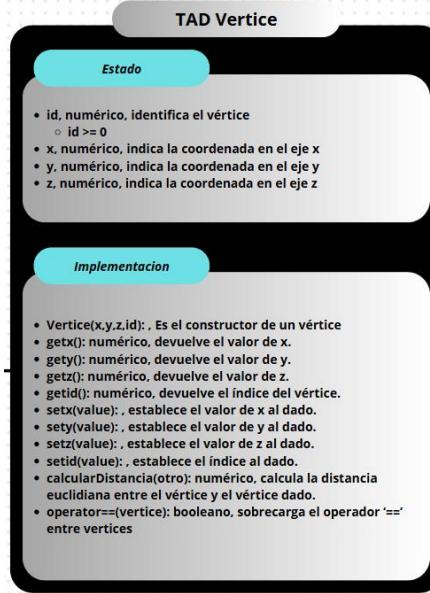


Figura 79. Especificación de vértice. Elaboración propia

Para este TAD se sigue teniendo un id y las coordenadas (x,y,z) que representan un punto del plano 3D. En cuanto a la implementación se tiene se tienen los getters y setters de sus atributos y una función adicional que calcula la distancia euclíadiana entre dos vértices la cual reutilizamos en este componente junto a una que sobrecarga el operador == para que el árbol rojinegro se siga construyendo correctamente.

### Diagrama de Relación de TADs:

A continuación, se muestra el diagrama de relación entre los TAD's utilizados por el programa:

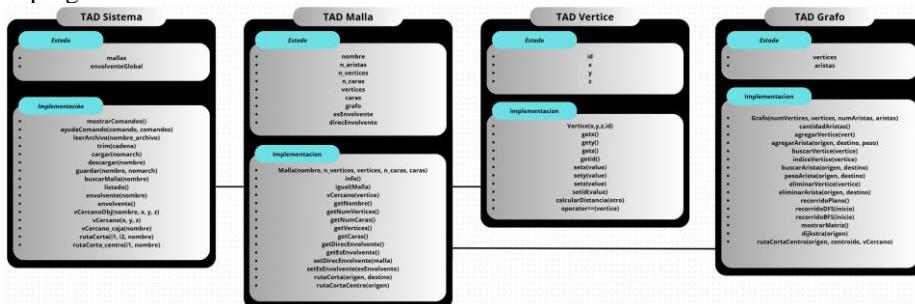


Figura 80. Diagrama de relación de TAD's. Elaboración propia

Es de resaltar que el main sigue funcionando como intermediario para las funcionalidades requeridas. El sistema, sigue gestionando la información de las mallas cargadas por medio de métodos propios del TAD de sistema que redirigen a métodos que forman parte del TAD de Malla que a su vez para este componente en específico redirigen al TAD Grafo. Un ejemplo que demuestra la integración entre los TAD's es cuando se usa cualquier comando de este componente como “*rutaCorta(i1, i2, nombre)*”, pues requiere el uso del TAD: Grafo, que realiza el cálculo del camino y la distancia mediante el algoritmo de Dijkstra, que están dentro de objetos de tipo TAD:Malla que hace uso que a su vez hace llamado el TAD: Sistema en la función correspondiente.

#### 4.3 Plan de Pruebas

##### Planificación de Pruebas

Previo a la codificación, fue realizado un proceso de análisis respecto al funcionamiento del comando *rutaCorta* y *rutaCortaCentro* para de esa forma crear una tabla que presentara el nombre de la prueba, el resultado esperado y los datos de entrada, la justificación de esta y como se va a ejecutar; lo anterior, de forma que fuera posible evaluar la mayor cantidad de casos de prueba posibles que demostrarán la funcionalidad y calidad de los comandos, que a su vez abogaran por el componente mismo.

Como bien se mencionó en la descripción de los comandos, el comando *rutaCorta* y *rutaCortaCentro* calculan: el camino y su distancia entre dos vértices de un objeto, y el camino y su distancia del vértice de un objeto a su centroide; ahora bien, como bien mencionamos en nuestro primer componente en el espacio tridimensional en el que nos encontramos un objeto 3D puede ubicarse en cualquiera de los 8 octantes formados a partir de una combinación diferente de signos que se ve así:

1. Primer octante:  $(x > 0), (y > 0), (z > 0)$
2. Segundo octante:  $(x < 0), (y > 0), (z > 0)$
3. Tercer octante:  $(x < 0), (y < 0), (z > 0)$
4. Cuarto octante:  $(x > 0), (y < 0), (z > 0)$
5. Quinto octante:  $(x > 0), (y > 0), (z < 0)$
6. Sexto octante:  $(x < 0), (y > 0), (z < 0)$
7. Séptimo octante:  $(x < 0), (y < 0), (z < 0)$
8. Octavo octante:  $(x > 0), (y < 0), (z < 0)$

Por lo que, nos pareció preciso que los casos de prueba consistieran en comprobar que el cálculo de esa *rutaCorta* y *rutaCortaCentro* funcionara sin importar los octantes en los que se encontraran los objetos 3D o incluso si se encontraban en medio de ellas, lo que se ve reflejado en la tabla 9.

La estrategia que tenemos la intención de utilizar es: Usar los 10 archivos que usamos en el componente uno, que nos permitieron no solo probar la efectividad de componente en su totalidad, logrando así un total de 10 casos de prueba.

**Tabla 9**  
*Plan de pruebas*

Nombre de la prueba	Resultado esperado y datos de entrada	Justificación	Cómo se ejecutará
Objeto 3D en el primer octante	rutaCorta para vertices 0 y 1, ruta-CortaCentro desde 2 ->Archivo de texto del objeto 3D en el primer octante	Evaluar la funcionalidad y calidad de los comandos con un objeto 3D en el primer octante	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar los siguientes comandos:  \$ cargar objOct1.txt \$ ruta_corta 0 1 Obj1 \$ ruta_corta_centro 2 Obj1
Objeto 3D en el segundo octante	rutaCorta para vertices 0 y 1, ruta-CortaCentro desde 2 ->Archivo de texto del objeto 3D en el segundo octante	Evaluar la funcionalidad y calidad de los comandos con un objeto 3D en el segundo octante	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar los siguientes comandos:  \$ cargar objOct2.txt \$ ruta_corta 0 1 Obj2 \$ ruta_corta_centro 2 Obj2
Objeto 3D en el tercer octante	rutaCorta para vertices 0 y 1, ruta-CortaCentro desde 2 ->Archivo de texto del objeto 3D en el tercer octante	Evaluar la funcionalidad y calidad de los comandos con un objeto 3D en el tercer octante	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar los siguientes comandos:  \$ cargar objOct3.txt \$ ruta_corta 0 1 Obj3 \$ ruta_corta_centro 2 Obj3

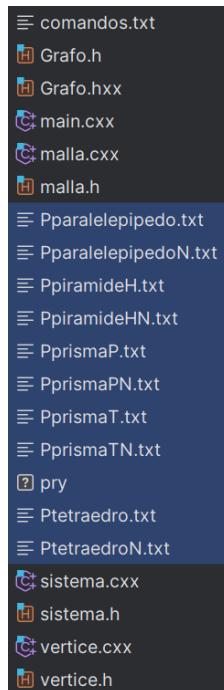
Objeto 3D en el cuarto octante	rutaCorta para vertices 0 y 1, rutaCortaCentro desde 2  ->Archivo de texto del objeto 3D en el cuarto octante	Evaluar la funcionalidad y calidad de los comandos con un objeto 3D en el cuarto octante	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar los siguientes comandos:  \$ cargar objOct4.txt \$ ruta_corta 0 1 Obj4 \$ ruta_corta_centro 2 Obj4
Objeto 3D en el quinto octante	rutaCorta para vertices 0 y 1, rutaCortaCentro desde 2  ->Archivo de texto del objeto 3D en el quinto octante	Evaluar la funcionalidad y calidad de los comandos con un objeto 3D en el quinto octante	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar los siguientes comandos:  \$ cargar objOct5.txt \$ ruta_corta 0 1 Obj5 \$ ruta_corta_centro 2 Obj5
Objeto 3D en el sexto octante	rutaCorta para vertices 0 y 1, rutaCortaCentro desde 2  ->Archivo de texto del objeto 3D en el sexto octante	Evaluar la funcionalidad y calidad de los comandos con un objeto 3D en el sexto octante	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar los siguientes comandos:  \$ cargar objOct6.txt \$ ruta_corta 0 1 Obj6 \$ ruta_corta_centro 2 Obj6
Objeto 3D en el séptimo octante	rutaCorta para vertices 0 y 1, rutaCortaCentro desde 2  ->Archivo de texto del objeto 3D en el séptimo octante	Evaluar la funcionalidad y calidad de los comandos con un objeto 3D en el séptimo octante	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar los siguientes comandos:  \$ cargar objOct7.txt \$ ruta_corta 0 1 Obj7

			\$ ruta_corta_centro 2 Obj7
Objeto 3D en el octavo octante	rutaCorta para vertices 0 y 1, rutaCortaCentro desde 2  ->Archivo de texto del objeto 3D en el octavo octante	Evaluar la funcionalidad y calidad de los comandos con un objeto 3D en el octavo octante	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar los siguientes comandos:  \$ cargar objOct8.txt \$ ruta_corta 0 1 Obj8 \$ ruta_corta_centro 2 Obj8
Objeto 3D en el primer y segundo octante	rutaCorta para vertices 0 y 1, rutaCortaCentro desde 2  ->Archivo de texto del objeto 3D en el primer y segundo octante	Evaluar la funcionalidad y calidad de los comandos con un objeto 3D entre el primer y segundo octante	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar los siguientes comandos:  \$ cargar objOct1y2.txt \$ ruta_corta 0 1 Obj9 \$ ruta_corta_centro 2 Obj9
Objeto 3D espejo en el primer y segundo octante	rutaCorta para vertices 0 y 1, rutaCortaCentro desde 2  ->Archivo de texto del objeto 3D espejo en el primer y segundo octante	Evaluar la funcionalidad y calidad de los comandos con un objeto 3D espejo entre el primer y segundo octante	1 Agregar el archivo de entrada a la carpeta del sistema 2 Compilar y ejecutar el sistema 3 Ejecutar los siguientes comandos:  \$ cargar objOct1y2.txt \$ ruta_corta 0 1 Obj10 \$ ruta_corta_centro 2 Obj10

*Nota: La tabla permite ver la planeación de casos de prueba para evaluar componente 3, elaboración propia.*

### Ejecución de Pruebas

Siguiendo la estrategia presentada en la planeación de pruebas se hizo la respectiva carga de archivos usados en el componente 1 de forma que quedaron los archivos presentados en la Figura 81, siguen la dinámica de ser objetos 3D que son espejos.



*Figura 81. Carga de los 10 archivos. Elaboración propia*

Ahora, para realizar ejecutar cada prueba se seguirá el orden de comandos presentado en la columna “Como se ejecutará”, y para documentar su correcto funcionamiento se presentará en una tabla el nombre de la prueba y sus datos de entrada, la imagen de los comandos ejecutados, y el resultado visual graficado con ayuda de Geogebra que nos permite una identificación eficiente de posibles errores, esta se presentará a continuación en la Tabla 10.

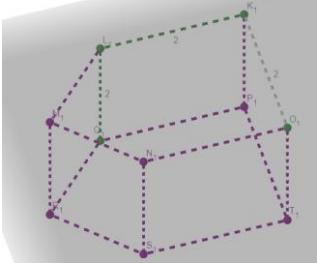
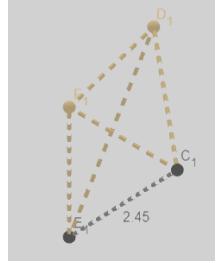
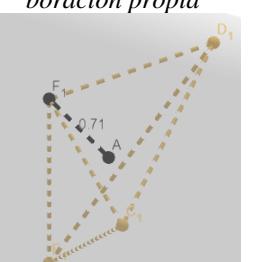
**Tabla 10**  
*Ejecución de las pruebas*

Nombre de la prueba y datos de entrada	Comandos ejecutados	Resultado presentado visualmente

<p>Objeto 3D en el primer octante -&gt;PprismaT.txt</p> <pre>\$ cargar PprismaT.txt El objeto PrismaTriangular ha sido cargado exitosamente desde el archivo PprismaT.txt . Elapsed time: 0.000962s Processing finished at Sun Nov 17 13:21:36 2024  \$ ruta_corta 0 4 PrismaTriangular La ruta mas corta que conecta los vertices 0 y 4 del objeto PrismaTriangular pasa por: 0, 1, 4; con una longitud de 4 Elapsed time: 0.002008s Processing finished at Sun Nov 17 13:21:40 2024  \$ ruta_corta_centro 3 PrismaTriangular La ruta mas corta que conecta el vertice 3 con el centro del objeto PrismaTriangular, ubicado en centro(3, 2.33333, 3), pasa por: 3, 5, 2, centro; con una longitud de 4.61606. Elapsed time: 0.002543s Processing finished at Sun Nov 17 13:21:45 2024</pre> <p><i>Figura 82. Prueba #1. Elaboración propia</i></p>	<p><i>Figura 83. Prueba #1: ruta_corta. Elaboración propia</i></p>
<p>Objeto 3D en el segundo octante -&gt;Pparalelepipedo.txt</p> <pre>\$ cargar Pparalelepipedo.txt El objeto Paralelepipedo ha sido cargado exitosamente desde el archivo Pparalelepipedo.txt. Elapsed time: 0.001994s Processing finished at Sun Nov 17 15:01:02 2024  \$ ruta_corta 1 7 Paralelepipedo La ruta mas corta que conecta los vertices 1 y 7 del objeto Paralelepipedo pasa por: 1, 0, 3, 7; con una longitud de 6.47214 Elapsed time: 0.002509s Processing finished at Sun Nov 17 15:01:06 2024  \$ ruta_corta_centro 4 Paralelepipedo La ruta mas corta que conecta el vertice 4 con el centro del objeto Paralelepipedo, ubicado en centro(-5, 3, 3), pasa por: 4, 0, 3, centro; con una longitud de 5.88635. Elapsed time: 0.003071s Processing finished at Sun Nov 17 15:01:12 2024</pre> <p><i>Figura 85. Prueba #2. Elaboración propia</i></p>	<p><i>Figura 86. Prueba #2: ruta_corta. Elaboración propia</i></p>

<p>Objeto 3D en el tercer octante</p> <p>-&gt; PprismaP.txt</p>	<pre>\$ cargar PprismaP.txt El objeto PrismaPentagonal ha sido cargado exitosamente desde el archivo PprismaP.txt.  Elapsed time: 0.002002s Processing finished at Sun Nov 17 17:45:49 2024  \$ ruta_corta 2 9 PrismaPentagonal La ruta mas corta que conecta los vertices 2 y 9 del objeto PrismaPentagonal pasa por: 2, 3, 4, 9; con una longitud de 5.41421 Elapsed time: 0.001999s Processing finished at Sun Nov 17 17:46:26 2024  \$ ruta_corta_centro 8 PrismaPentagonal La ruta mas corta que conecta el vertice 8 con el centro del objeto PrismaPentagonal , ubicado en centro(-1.6, -2, 4), pasa por: 8, 7, 6, 1, centro; con una longitud de 6.3 6466 Elapsed time: 0.003417s Processing finished at Sun Nov 17 17:46:41 2024</pre> <p><i>Figura 88. Prueba #3. Elaboración propia</i></p>	<p><i>Figura 89. Prueba #3: ruta_corta. Elaboración propia</i></p>
<p>Objeto 3D en el cuarto octante</p> <p>-&gt; Pte-trae-dro.txt</p>	<pre>\$ cargar PteTraedro.txt El objeto Tetraedro ha sido cargado exitosamente desde el archivo PteTraedro.txt. Elapsed time: 0.00151s Processing finished at Sun Nov 17 17:48:24 2024  \$ ruta_corta 3 1 Tetraedro La ruta mas corta que conecta los vertices 3 y 1 del objeto Tetraedro pasa por: 3, 1; con una longitud de 1.41421 Elapsed time: 0.002519s Processing finished at Sun Nov 17 17:48:43 2024  \$ ruta_corta_centro 2 Tetraedro La ruta mas corta que conecta el vertice 2 con el centro del objeto Tetraedro , ubicado en centro(2, -2.5, 2.5), pasa por: 2, 3, centro; con una longitud de 2.70711. Elapsed time: 0.002515s Processing finished at Sun Nov 17 17:49:04 2024</pre> <p><i>Figura 91. Prueba #4. Elaboración propia</i></p>	<p><i>Figura 92. Prueba #4: ruta_corta. Elaboración propia</i></p>

<p>Objeto 3D en el quinto octante -&gt; PprismaTN.txt</p> <pre>\$ cargar PprismaTN.txt El objeto PrismaTriangular_Negativo ha sido cargado exitosamente desde el archivo PprismaTN.txt. Elapsed time: 0.001013s Processing finished at Sun Nov 17 17:50:31 2024  \$ ruta_corta 5 0 Prismatriangular_Negativo La ruta mas corta que conecta los vertices 5 y 0 del objeto Prismatriangular_Negativo pasa por: 5, 3, 0; con una longitud de 3.41421 Elapsed time: 0.001998s Processing finished at Sun Nov 17 17:51:33 2024  \$ ruta_corta_centro 4 Prismatriangular_Negativo La ruta mas corta que conecta el vertice 4 con el centro del objeto Prismatriangular_Negativo , ubicado en centro(3, 2.33333, -3), pasa por: 4, 5, 2, centro; con una longitud de 4.61606. Elapsed time: 0.001507s Processing finished at Sun Nov 17 17:51:45 2024</pre>	
<p>Objeto 3D en el sexto octante -&gt; PparallelepipedoN.txt</p> <pre>\$ cargar PparallelepipedoN.txt El objeto Paralelepipedo_Negativo ha sido cargado exitosamente desde el archivo PparallelepipedoN.txt. Elapsed time: 0.001549s Processing finished at Sun Nov 17 17:56:34 2024  \$ ruta_corta 4 2 Paralelepipedo_Negativo La ruta mas corta que conecta los vertices 4 y 2 del objeto Paralelepipedo_Negativo pasa por: 4, 5, 1, 2; con una longitud de 6.47214 Elapsed time: 0.002996s Processing finished at Sun Nov 17 17:56:57 2024  \$ ruta_corta_centro 6 Paralelepipedo_Negativo La ruta mas corta que conecta el vertice 6 con el centro del objeto Paralelepipedo_Negativo , ubicado en centro(-5, 3, -3), pasa por: 6, 7, 3, centro; con una longitud de 5.65028. Elapsed time: 0.002999s Processing finished at Sun Nov 17 17:57:10 2024</pre>	

<p>Objeto 3D en el séptimo octante -&gt; PprismaPN.txt</p>	<pre>\$ cargar PprismaPN.txt El objeto PrismaPentagonal_Negativo ha sido cargado exitosamente desde el archivo PprismaPN.txt. Elapsed time: 0.000998s Processing finished at Sun Nov 17 17:59:50 2024  \$ ruta_corta 6 4 PrismaPentagonal_Negativo La ruta mas corta que conecta los vertices 6 y 4 del objeto PrismaPentagonal_Negativo pasa por: 6, 1, 0, 4; con una longitud de 6 Elapsed time: 0.001997s Processing finished at Sun Nov 17 18:00:09 2024  \$ ruta_corta_centro 9 PrismaPentagonal_Negativo La ruta mas corta que conecta el vertice 9 con el centro del objeto PrismaPentagonal_Negativo , ubicado en centro(-1.6, -2, -4), pasa por: 9, 4, 0, 1, centro; con una longitud de 7.53623; Elapsed time: 0.004024s Processing finished at Sun Nov 17 18:00:21 2024</pre>	 <p>Figura 101. Prueba #7: ruta_corta. Elaboración propia</p>
<p>Objeto 3D en el octavo octante -&gt; PtetraedroN.txt</p>	<pre>\$ cargar PtetraedroN.txt El objeto Tetraedro_Negativo ha sido cargado exitosamente desde el archivo PtetraedroN.txt. Elapsed time: 0.001004s Processing finished at Sun Nov 17 18:01:44 2024  \$ ruta_corta 0 2 Tetraedro_Negativo La ruta mas corta que conecta los vertices 0 y 2 del objeto Tetraedro_Negativo pasa por: 0, 2; con una longitud de 2.44949 Elapsed time: 0.001001s Processing finished at Sun Nov 17 18:02:05 2024  \$ ruta_corta_centro 3 Tetraedro.Negativo La ruta mas corta que conecta el vertice 3 con el centro del objeto Tetraedro_Negativo , ubicado en centro(2, -2.5, -2.5), pasa por: 3, centro; con una longitud de 0.707107 Elapsed time: 0.007003s Processing finished at Sun Nov 17 18:02:14 2024</pre>	 <p>Figura 104. Prueba #8: ruta_corta. Elaboración propia</p>  <p>Figura 105. Prueba #8: ruta_corta_centro. Elaboración propia</p>

<p>Objeto 3D en el primer y segundo octante</p> <p>-&gt; PpiramideH.txt</p>	<pre>\$ cargar PpiramideH.txt El objeto PiramideHexagonal ha sido cargado exitosamente desde el archivo PpiramideH.txt. Elapsed time: 0.002522s Processing finished at Sun Nov 17 18:03:15 2024  \$ ruta_corta 1 4 PiramideHexagonal La ruta mas corta que conecta los vertices 1 y 4 del objeto PiramideHexagonal pasa por: 1, 2, 3, 4; con una longitud de 4.82843 Elapsed time: 0.001996s Processing finished at Sun Nov 17 18:03:28 2024  \$ ruta_corta_centro 2 PiramideHexagonal La ruta mas corta que conecta el vertice 2 con el centro del objeto PiramideHexagonal, ubicado en centro(0, 4, -3.42857), pasa por: 2, 1, 0, centro; con una longitud de 4.89194. Elapsed time: 0.001999s Processing finished at Sun Nov 17 18:03:37 2024</pre>	<p>Figura 107. Prueba #9: ruta_corta. Elaboración propia</p>
<p>Objeto 3D espejo en el primer y segundo octante</p> <p>-&gt;PpiramideHN.txt</p>	<pre>\$ cargar PpiramideHN.txt El objeto PiramideHexagonal_Negativo ha sido cargado exitosamente desde el archivo PpiramideHN.txt. Elapsed time: 0.001006s Processing finished at Sun Nov 17 18:17:29 2024  \$ ruta_corta 5 3 PiramideHexagonal_Negativo La ruta mas corta que conecta los vertices 5 y 3 del objeto PiramideHexagonal_Negativo pasa por: 5, 4, 3; con una longitud de 3.41421 Elapsed time: 0.001006s Processing finished at Sun Nov 17 18:17:43 2024  \$ ruta_corta_centro 6 PiramideHexagonal_Negativo La ruta mas corta que conecta el vertice 6 con el centro del objeto PiramideHexagonal_Negativo, ubicado en centro(0, 4, -3.42857), pasa por: 6, 0, centro; con una longitud de 4.79435. Elapsed time: 0.002023s Processing finished at Sun Nov 17 18:17:55 2024</pre>	<p>Figura 108. Prueba #9: ruta_corta_centro. Elaboración propia</p>

*Nota: La tabla permite ver la ejecución de los casos de prueba para evaluar los comandos rutaCorta y rutaCortaCentro. Para la elaboración de la columna “comandos*

*ejecutados” se asume que ya se tiene el compilado ejecutado. La grafica puede verse directamente desde Geogebra a través del siguiente enlace: <https://www.geogebra.org/3d/cjyp8ejb>. Elaboración propia.*

Es así como concluye nuestro plan de pruebas del componente 3, donde se puede evidenciar visualmente a lo largo de las 10 pruebas como no solo los comandos *rutaCorta* y *rutaCortaCentro* son perfectamente funcionales, sino que el componente mismo funciona de maravilla y con calidad, sea al usar el comando cargar o cualquier otro lo que demuestra un desarrollo exitoso del proyecto en general.

## Conclusión

El desarrollo e implementación de este sistema para la manipulación de mallas poligonales tridimensionales resalta el papel fundamental de las estructuras de datos en el avance de las aplicaciones de gráficos por computadora y modelado 3D. Al habilitar funcionalidades como la carga, organización, análisis y optimización de mallas poligonales, este proyecto proporciona un marco sólido adecuado para diversas aplicaciones, incluyendo videojuegos, simulación y animación.

A través del diseño cuidadoso de comandos y protocolos de prueba, el sistema ha demostrado fiabilidad y precisión en operaciones como la generación de cajas envolventes, identificación de vértices más cercanos y cálculo de caminos más cortos. Además, las correcciones y mejoras aplicadas durante la fase de desarrollo incrementaron la usabilidad y la resiliencia del sistema, resolviendo desafíos como la validación de parámetros y los problemas de formato en los archivos.

Este proyecto no solo cumple con sus objetivos técnicos, sino que también sienta las bases para futuras exploraciones y mejoras. El trabajo futuro podría enfocarse en extender las capacidades del sistema para soportar tipos de mallas más complejas, optimizar el rendimiento computacional e integrarse con herramientas de renderizado avanzadas. En definitiva, este proyecto destaca el potencial transformador de las estructuras de datos para resolver problemas del mundo real en el ámbito de los gráficos por computadora.

## Referencias

1. Tiiimägi, S. (2024). *¿Qué es una malla poligonal y cómo editarla?* 3D Studio. Recuperado de <https://3dstudio.co/es/polygon-mesh/>
2. 3DCadPortal. (2024). *Mesh - Malla Poligonal*. Recuperado de <https://www.3dcadportal.com/mesh-malla-poligonal.html>
3. PNGWing. (2024). *Descarga gratuita de imágenes PNG*. Recuperado de <https://www.pngwing.com/es/free-png-xqvno>