

Proyecto – Entrega Final

Pontificia Universidad Javeriana



Estructura de datos

Juan Manuel López Vargas

Sebastián Almanza Galvis

David León Velásquez

Johan Sebastián Méndez

Profesor: John Corredor

20 de octubre 2024

Entrega Final:

Evaluación:

1. Estructuras de Datos

- **Vértice:** Representa un punto en el espacio tridimensional con coordenadas (x, y, z). Su método más importante es el cálculo de la **distancia euclidiana** entre dos vértices.
- **Malla:** Almacena la geometría de los objetos 3D mediante vértices y caras. Además, cada malla incluye un **KD-Tree**, lo que facilita la búsqueda rápida de vértices cercanos.

2. KD-Tree: Estructura y Algoritmos

El **KD-Tree** es el núcleo de la optimización en la búsqueda de vértices cercanos:

- **Nodo KD:** Cada nodo contiene un vértice y divide el espacio 3D en subespacios a través de los ejes X, Y, y Z.
- **Inserción:** Los vértices se insertan en el árbol de manera recursiva, dividiendo el espacio en función de un eje rotatorio en cada nivel.
- **Búsqueda del vértice más cercano:** El algoritmo recorre el KD-Tree de manera eficiente, explorando primero el subárbol más prometedor, lo que reduce drásticamente el número de comparaciones necesarias.

3. Funcionalidad Mejorada en la Entrega 2

- **v_cercano:** Este comando utiliza el KD-Tree para encontrar de manera eficiente el vértice más cercano a un punto dado, realizando búsquedas rápidas con una complejidad mucho menor que en una implementación sin optimización.
- **v_cercanos_caja:** Implementación para encontrar los vértices más cercanos a las esquinas de la caja envolvente de una malla.

4. Comparación con Entregas Anteriores

- En **Entrega 0**, solo existía una simulación de comandos sin cálculos reales.
- En **Entrega 1**, se introdujo la funcionalidad básica de carga de mallas y cálculo de la caja envolvente, pero la búsqueda de vértices cercanos no estaba completamente implementada.

- **Entrega 2** añade una implementación completa y eficiente de la búsqueda de vértices mediante KD-Trees, mejorando la velocidad y escalabilidad de la aplicación.

Conclusión

El cambio fundamental en la **Entrega 2** fue la incorporación del **KD-Tree**, que permitió realizar búsquedas espaciales eficientes, optimizando la búsqueda de vértices cercanos y reduciendo la complejidad computacional, lo cual es crucial cuando se manejan grandes volúmenes de datos 3D. Esto marca una clara mejora respecto a las simulaciones y las implementaciones previas.

Descripción General del Programa

Este programa permite cargar, procesar y realizar consultas sobre objetos 3D representados como mallas poligonales. Las mallas están compuestas de vértices (puntos en el espacio tridimensional) y caras (conjuntos de vértices conectados por aristas que forman polígonos).

¿Porque KD-tree?

Para empezar, decidimos usar KD-Tree porque nos permite resolver de manera eficiente el problema que tenemos de buscar vértices cercanos en mallas 3D. Este árbol divide el espacio en secciones según los ejes X, Y y Z, lo que hace que las búsquedas sean mucho más rápidas. En lugar de revisar cada vértice uno por uno, el KD-Tree nos ayuda a encontrar el más cercano reduciendo el número de comparaciones, lo que es ideal cuando trabajamos con grandes cantidades de datos en tres dimensiones.

TADS KD-tree:

Nodo: Representa un nodo en el KD-tree

Atributos:

- **dato:** El valor almacenado en el nodo (de tipo genérico T).
- **hijoIzq y hijoDer:** Punteros a los hijos izquierdo y derecho, respectivamente.
- **tag:** Un entero que indica el eje de comparación (0: X, 1: Y, 2: Z).

Métodos:

- **obtenerDato(), fijarDato():** Obtiene y fija el dato almacenado en el nodo.
- **obtenerHijoIzq(), fijarHijoIzq():** Obtiene y fija el hijo izquierdo.

- **obtenerHijoDer(), fijarHijoDer():** Obtiene y fija el hijo derecho.
- **obtenerTag(), fijarTag():** Obtiene y fija el eje de comparación del nodo.

Árbol: Representa el árbol KD-tree completo

Atributos:

- **raiz:** Un puntero al nodo raíz del árbol.

Métodos:

- **insertar(T dato):** Inserta un nuevo dato en el árbol.
- **encontrarMasCercano(const T& objetivo, double& mejorDistancia):** Encuentra el nodo más cercano a un objetivo dado utilizando una búsqueda recursiva.

Vertice: Es una estructura que representa un punto en el espacio 3D con coordenadas x, y, z

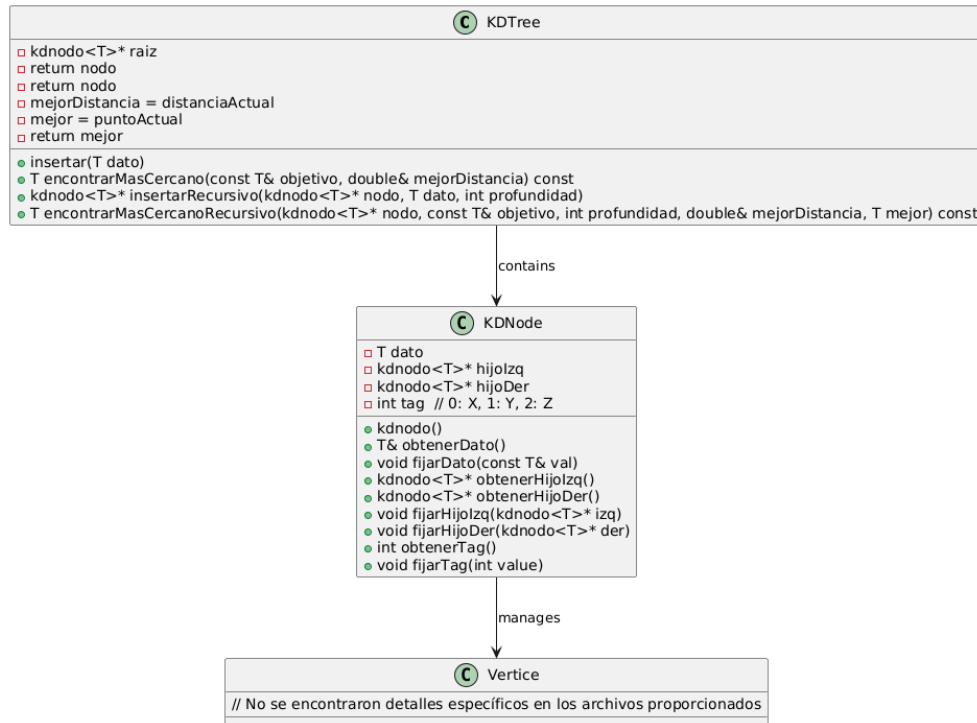
Atributos:

- **px, py, pz:** Coordenadas del vértice en los ejes X, Y y Z.

Métodos:

- **distancia(const Vertice& otro):** Calcula la distancia euclidiana entre dos vértices.
- Sobrecarga del operador << para imprimir los vértices en la salida estándar en el formato (px, py, pz).

Diagrama TADS del árbol:



Métodos:

- **Cargar una malla desde un archivo:** Los archivos de mallas contienen los datos de vértices y caras.
- **Guardar la malla específica en un archivo:** Guarda una malla existente en memoria a un archivo especificado.
- **Calcular la caja:** Calcula la caja envolvente de una malla específica.
- **Calcular caja global:** Calcula la caja envolvente global de todas las mallas cargadas.
- **Buscar el vértice más cercano a un punto dado:** Usando un KD-Tree, el programa encuentra eficientemente el vértice más cercano.
- **Calcular los vértices más cercanos a las esquinas de la caja envolvente de una malla:** La caja envolvente es el rectángulo 3D que rodea completamente a la malla.
- **Listar las mallas cargadas:** Muestra los nombres y detalles de todas las mallas cargadas en memoria.
- **Ruta más corta:** Calcula la ruta más corta desde un vértice dado hasta el centro de la malla.
- **Ruta más corta entre vértices:** Calcula la ruta más corta entre dos vértices específicos de una malla.

- **Proporcionar ayuda:** El programa muestra descripciones de uso para cada comando disponible.
- **Eliminar malla:** Elimina una malla específica de la memoria.
- **Salida:** Cierre de la aplicación.

El programa interactúa con el usuario a través de una interfaz de comandos en una consola.

Descripción de los Archivos

Archivo 1: `vertice.h`

Propósito:

Define la estructura de datos `Vértice`, que representa un punto en el espacio 3D con coordenadas (x, y, z).

Contenido:

- Atributos:

- float px, py, pz: Coordenadas del vértice en el espacio tridimensional.

- Constructores:

- **Vertice()**: Constructor por defecto que inicializa las coordenadas en $((0, 0, 0))$.
- **Vertice(float x, float y, float z)**: Constructor que inicializa las coordenadas del vértice con valores específicos.

- Métodos:

- float distancia(const Vertice& otro) const: Calcula la distancia euclidiana entre el vértice actual y otro vértice dado.
- friend std::ostream& operator<<(std::ostream& os, const Vertice& v): Sobrecarga del operador `<<` para imprimir un vértice.

Archivo 2: `kdnodo.h`

Propósito

Define la plantilla de clase `kdnodo<T>`, que representa un nodo en un KD-Tree. Los nodos almacenan datos genéricos (en este caso, vértices) y referencian a sus hijos izquierdo y derecho.

Contenido

Atributos:

- **T dat:** Dato almacenado en el nodo (en este caso, un Vértice).
- **kdnodo<T>* hijoIzq, hijoDer:** Punteros a los hijos izquierdo y derecho.
- **int tag:** Indica el eje de partición (0 = eje X, 1 = eje Y, 2 = eje Z).

Métodos:

- **kdnodo():** Constructor por defecto que inicializa los punteros a `nullptr` y el `tag` a 0.
- **T& obtenerDato():** Retorna el dato almacenado en el nodo.
- **void fijarDato(const T& val):** Asigna un valor al dato del nodo.
- **kdnodo<T>* obtenerHijoIzq():** Retorna el puntero al hijo izquierdo.
- **void fijarHijoIzq(kdnodo<T>* izq):** Asigna el hijo izquierdo.
- **kdnodo<T>* obtenerHijoDer():** Retorna el puntero al hijo derecho.
- **void fijarHijoDer(kdnodo<T>* der):** Asigna el hijo derecho.
- **int obtenerTag():** Retorna el valor de tag (el eje de partición).
- **void fijarTag(int value):** Asigna el valor de tag.

Archivo 3: `kdnodo.hxx`

Propósito

Implementa los métodos de la plantilla `kdnodo<T>`, que han sido declarados en `kdnodo.h`.

Métodos implementados

- **kdnodo():** Inicializa los hijos en `nullptr` y el tag en 0.

- **obtenerDato():** Retorna el dato del nodo.
- **fijarDato():** Establece el valor del dato en el nodo.
- **obtenerHijoIzq() y obtenerHijoDer():** Retornan los punteros a los hijos izquierdo y derecho, respectivamente.
- **fijarHijoIzq() y fijarHijoDer():** Asignan valores a los hijos izquierdo y derecho.
- **obtenerTag() y fijarTag():** Obtienen y asignan el valor del eje de partición (tag).

Archivo 4:kdtree.h

Propósito

Define la clase `kdtree<T>`, que representa un KD-Tree para almacenar y buscar puntos en un espacio tridimensional. El KD-Tree permite realizar búsquedas eficientes de puntos cercanos.

Contenido

- Atributos:

- **kdnodo<T>* raiz:** Puntero a la raíz del árbol.

- Métodos:

- **kdtree():** Constructor que inicializa el árbol vacío.
- **void insertar(T dato):** Inserta un nuevo dato en el árbol, posicionándolo según las coordenadas en el eje correspondiente (X, Y, Z).
- **T encontrarMasCercano(const T& objetivo, double& mejorDistancia) const:** Encuentra el nodo más cercano a un punto objetivo usando la distancia euclidiana.

Explicación del Funcionamiento

- **Inserción:** Cada vez que se inserta un nuevo nodo en el árbol, el eje de partición cambia de manera cíclica entre los ejes X, Y, y Z. Dependiendo del valor del punto en el eje actual, el nuevo nodo se inserta en el subárbol izquierdo o derecho.
- **Búsqueda de vecino más cercano:** La búsqueda recursiva compara el punto actual con el objetivo, y explora primero el lado del subárbol que probablemente esté más cerca. Luego, verifica si el otro subárbol puede contener un punto más cercano antes de descartarlo.

Archivo 5: main.cpp

Propósito

Implementa la interfaz de usuario del programa, permitiendo cargar mallas, listar las mallas cargadas, encontrar vértices cercanos, y más, usando una línea de comandos.

Contenido

- Funciones principales:

- **convertirMinusculas(string cadena):** Convierte una cadena de caracteres a minúsculas.
- **dividirComando(const string& linea):** Divide una línea de comando en palabras individuales.
- **Malla LecturaMallas(const string& argumento1):** Carga una malla desde un archivo y la almacena en una estructura `Malla`.
- **void encontrarVerticeCercano(Malla& malla, const Vertice& punto):** Busca el vértice más cercano a un punto dado en una malla.
- **void listarMallas(const vector<Malla>& mallas):** Lista todas las mallas cargadas en memoria.
- **void encontrarVerticesCercanosCaja(Malla& malla):** Calcula los vértices más cercanos a las esquinas de la caja envolvente de una malla.

- Función main:

La función principal del programa maneja la interacción del usuario con la aplicación a través de comandos como:

- **cargar:** Carga una malla desde un archivo.
- **v_cercano:** Busca el vértice más cercano a un punto.
- **v_cercanos_caja:** Encuentra los vértices más cercanos a las esquinas de la caja envolvente.
- **listado:** Lista las mallas cargadas.
- **ayuda:** Proporciona información sobre cómo utilizar los comandos.
- **salir:** Termina la ejecución del programa.

Ejemplo de Uso

- Al iniciar el programa, se muestra un menú con los comandos disponibles.
- El usuario puede cargar una malla con el comando cargar <nombre_archivo>.
- Luego, puede buscar vértices cercanos con v_cercano <px> <py> <pz> <nombre_malla>, listar las mallas con listado, y salir del programa con salir.

Explicación de Cada Método

Métodos de **Vertice**

- **Vertice()**: Constructor por defecto que inicializa las coordenadas del vértice a cero.
- **Vertice(float x, float y, float z)**: Constructor parametrizado que inicializa las coordenadas con los valores dados.
- **distancia(const Vertice& otro)**: Calcula la distancia euclidiana entre dos vértices.
- Sobrecarga <<: Permite imprimir un vértice de manera legible.

Métodos de **kdnodo<T>**

- **kdnodo()**: Constructor que inicializa los punteros a nullptr y el tag en 0.
- **T& obtenerDato()**: Retorna el dato almacenado en el nodo.
- **void fijarDato(const T& val)**: Asigna un valor al dato del nodo.
- **kdnodo<T>* obtenerHijoIzq()**: Retorna el puntero al hijo izquierdo.
- **void fijarHijoIzq(kdnodo<T>* izq)**: Asigna el hijo izquierdo.
- kdnodo<T>* obtenerHijoDer()**: Retorna el puntero al hijo derecho.
- void fijarHijoDer(kdnodo<T>* der)**: Asigna el hijo derecho.
- int obtenerTag()**: Retorna el valor de tag.
- void fijarTag(int value)**: Asigna el valor de tag.

Pruebas:

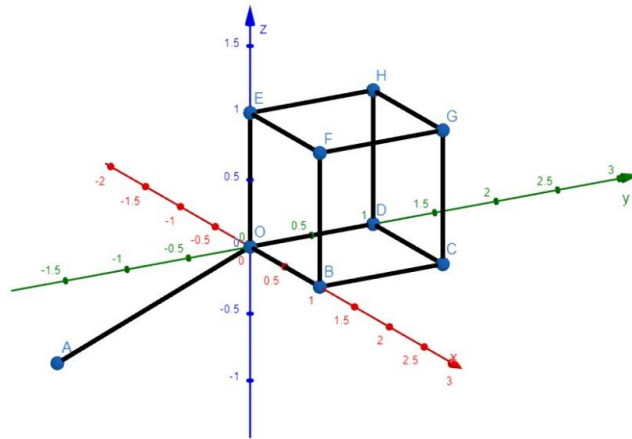
```
sebas@sebas-VirtualBox: ~/Descargas/Entrega-2-2_$ ./main
Bienvenido a la aplicacion de modelado de objetos 3D.
Comandos disponibles:
- cargar <nombre_archivo>
- v_cercano <px> <py> <pz> <nombre_malla>
- v_cercanos_caja <nombre_malla>
- listado
- salir
Para obtener ayuda sobre un comando especifico, ingrese 'ayuda <comando>'.
$ cargar malla_cubo.txt
$
$ listado
Objeto: malla_de_un_cubo, Vertices: 8, Caras: 5
$
$ v_cercano -1 -1 -1 malla_de_un_cubo
El vertice mas cercano es (0, 0, 0) a una distancia de 1.73205
$
$ v_cercano 0.87 2 -1 malla_de_un_cubo
El vertice mas cercano es (1, 1, 0) a una distancia de 1.42018
$
$ v_cercanos_caja malla_de_un_cubo
El vertice mas cercano es (0, 0, 0) a una distancia de 0
El vertice mas cercano es (1, 0, 0) a una distancia de 0
El vertice mas cercano es (0, 1, 0) a una distancia de 0
El vertice mas cercano es (1, 1, 0) a una distancia de 0
El vertice mas cercano es (0, 0, 1) a una distancia de 0
El vertice mas cercano es (1, 0, 1) a una distancia de 0
El vertice mas cercano es (0, 1, 1) a una distancia de 0
El vertice mas cercano es (1, 1, 1) a una distancia de 0
$
```

Resultado en pantalla:

```
$ v_cercano -1 -1 -1 malla_de_un_cubo
El vertice mas cercano es (0, 0, 0) a una distancia de 1.73205
$
```

Resultado explicado con geogebra:

●	n = Segmento(O,E)	..
	= 1	
●	p = Segmento(B,F)	..
	= 1	
●	q = Segmento(C,G)	..
	= 1	
●	r = Segmento(D,H)	..
	= 1	
●	A = (-1, -1, -1)	..
	W = Distancia(A,O)	..
	= 1.73	
●	T = Segmento(O,A)	..
	= 1.73	



Explicación:

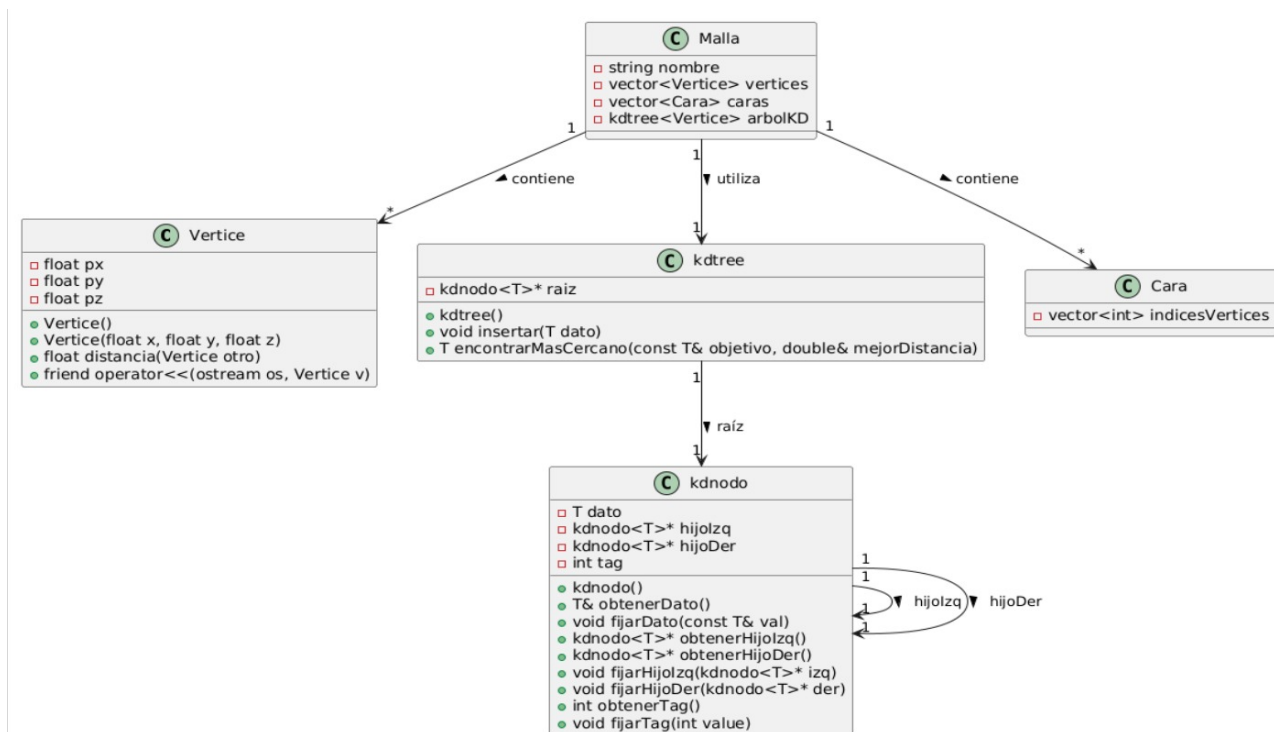
- **Input del usuario:** Cuando ejecutas el comando `v_cercano -1 -1 -1` `mallade_un_cubo`, el programa recibe las coordenadas del punto $(-1, -1, -1)$ y busca dentro del objeto `mallade_un_cubo` el vértice más cercano a ese punto.
- **Distancia Euclidiana:** La distancia que se calcula sigue la fórmula:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

- **Resultado:**

$$d = \sqrt{(0 - (-1))^2 + (0 - (-1))^2 + (0 - (-1))^2} = \sqrt{3} \approx 1.732$$

Diagrama UML del código:



Plan de pruebas entrega 3:

1. cargar <nombre_archivo>

Caso de Prueba	Entrada	Salida Esperada	Salida Obtenida
Archivo válido	cargar malla_cubo.txt	Malla cargada correctamente.	El objeto ha sido cargado exitosamete desde el archivo.
Archivo no existente	cargar archivo_inexistente. txt	El archivo no existe o no es legible.	El archivo no existe o es ilegible.

2. guardar <nombre_malla> <archivo>

Caso de Prueba	Entrada	Salida Esperada	Salida Obtenida
Malla cargada correctamente	guardar malla_de_un_cubo cubo_guardado.txt	Malla guardada correctamente.	La informacion del objeto ha sido guardada exitosamente en el archivo.
Malla no existente	guardar malla_no_existente salida.txt	La malla no está en memoria.	El objeto no ha sido cargada en memoria.

3. listado

Caso de Prueba	Entrada	Salida Esperada	Salida Obtenida
----------------	---------	-----------------	-----------------

Sin mallas cargadas	listado	La memoria está vacía.	La memoria está vacía.
Con varias mallas cargadas	listado	Lista de mallas: malla_cubo (vértices: 8, caras: 12).	Lista de mallas: malla_cubo (vértices: 8, caras: 12).

4. envolverte <nombre_malla>

Caso de Prueba	Entrada	Salida Esperada	Salida Obtenida
Malla existente	envolverte malla_cubo	Caja envolverte generada con éxito.	La caja envolverte del objeto se ha generado con el nombre..
Malla no existente	envolverte malla_no_existente	La malla no está cargada.	El objeto no ha sido cargado en memoria.

5. envolverte global

Caso de Prueba	Entrada	Salida Esperada	Salida Obtenida
Con varias mallas cargadas	envolverte global	Envolverte global generada con éxito.	La caja envolverte global se ha generado con el nombre.

6. v_cercano <px> <py> <pz> <nombre_malla>

Caso de Prueba	Entrada	Salida Esperada	Salida Obtenida
Punto dentro de la malla	v_cercano 0.5 0.5 0.5 malla_cubo	Vértice más cercano: (0,0,0), distancia: 0.866.	Vértice más cercano: (0, 0,0) a una distancia de 0.866.

7. v_cercano <px> <py> <pz> global

Caso de Prueba	Entrada	Salida Esperada	Salida Obtenida
Con varias mallas cargadas	v_cercano 0.5 0.5 0.5 global	Vértice más cercano globalmente y su distancia.	El vertice mas cercano global es (0,0,0) a una distancia de 0.866025

8. descargar <nombre_malla>

Caso de Prueba	Entrada	Salida Esperada	Salida Obtenida
Malla existente	descargar malla_cubo	La malla se eliminó correctamente.	El objeto ha sido eliminado de la memoria.

Malla no existente	descargar malla_no_existente	La malla no está en memoria.	El objeto no ha sido encontrado en la memoria
--------------------	---------------------------------	------------------------------	---

9. ruta_corta_centro <i1> <nombre_malla>

Caso de Prueba	Entrada	Salida Esperada	Salida Obtenida
Índice válido y malla existente	ruta_corta_centro 0 malla_cubo	Secuencia de vértices desde el índice al centro.	La ruta mas corta desde el vertice cero hasta el centro de la malla es 0.86
Malla no existente	ruta_corta_centro 0 malla_no_existente	La malla no está cargada.	El objeto no ha sido cargado a memoria

10. ruta_corta <i1> <i2> <nombre_malla>

Caso de Prueba	Entrada	Salida Esperada	Salida Obtenida
Índices válidos y malla existente	ruta_corta 0 7 malla_cubo	Secuencia de vértices desde el índice i1 al i2.	La distancia mas corta entre los vertices 0 y 7 es de 1.41
Malla no existente	ruta_corta 0 7 malla_no_existente	La malla no está cargada.	El objeto no ha sido cargado a memoria

11. ayuda <comando>

Caso de Prueba	Entrada	Salida Esperada	Salida Obtenida
Comando válido	ayuda cargar	Descripción del comando cargar.	El comando cargar permite cargar una malla desde un archivo.
Comando no válido	ayuda comando_invalido	El comando no es reconocido.	Comando no reconocido

12. salir

Caso de Prueba	Entrada	Salida Esperada	Salida Obtenida
Comando ejecutado	salir	El programa se cierra correctamente.	El programa se cerró correctamente.

Pruebas:

1. cargar <nombre_archivo>

Carga una malla desde un archivo.

- Casos de Prueba

válido con datos correctos:

Entrada: cargar malla_cubo.txt

Salida esperada: Mensaje de éxito indicando que la malla se cargó correctamente.

```
$ cargar malla_cubo.txt
El objeto malla_de_un_cubo ha sido cargado exitosamente desde el archivo malla_cubo.txt.
$
```

Archivo no existente:

Entrada: cargar archivo_inexistente.txt

Salida esperada: Mensaje de error indicando que el archivo no existe o no es legible.

```
$ cargar archivo_inexistente.txt
El archivo archivo_inexistente.txt no existe o es ilegible.
$
```

2. guardar <nombre_malla> <archivo>

Guarda una malla en un archivo.

Casos de Prueba

Malla cargada correctamente:

Entrada: guardar_malla_de_un_cubo cubo_guardado.txt

Salida esperada: Mensaje de éxito indicando que la malla se guardó correctamente.

```
$ guardar malla_de_un_cubo cubo_guardado.txt
La informacion del objeto malla_de_un_cubo ha sido guardada exitosamente en el archivo cubo_guardado.txt.
$
```

Malla no existente:

Entrada: guardar malla_no_existente salida.txt

Salida esperada: Mensaje de error indicando que la malla no está en memoria.

```
$ guardar malla_no_existente salida.txt
El objeto malla_no_existente no ha sido cargado en memoria.
$
```

3. listado

Lista todas las mallas cargadas en memoria.

Casos de Prueba

Sin mallas cargadas:

Entrada: listado

Salida esperada: Mensaje indicando que la memoria está vacía.

```
$ listado
Memoria vacia.
```

Con varias mallas cargadas:

Entrada: listado

Salida esperada: Lista de todas las mallas con su nombre, número de vértices y caras.

```
$ listado
Hay 2 objetos en memoria:
malla_piramide contiene 5 vertices, 5 caras.
malla_de_un_cubo contiene 8 vertices, 5 caras.
```

4. envolvente <nombre_malla>

Calcula la caja envolvente de una malla específica.

Casos de Prueba

Malla existente:

Entrada: envolverte malla_cubo

Salida esperada: Mensaje indicando que la caja envolverte se generó con éxito.

```
$ envolverte malla_de_un_cubo  
La caja envolverte del objeto malla_de_un_cubo se ha generado con el nombre env_malla_de_un_cubo.
```

Malla no existente:

Entrada: envolverte malla_no_existente

Salida esperada: Mensaje de error indicando que la malla no está cargada.

```
$ envolverte malla_no_existente  
El objeto malla_no_existente no ha sido cargado en memoria.
```

5. envolverte global

Calcula la caja envolverte global de todas las mallas cargadas.

Caso de Prueba

Con varias mallas cargadas:

Entrada: envolverte global

Salida esperada: Mensaje indicando que la envolverte global se generó con éxito.

```
$ envolverte  
La caja envolverte global se ha generado con el nombre env_global.
```

6. v_cercano <px> <py> <pz> <nombre_malla>

Encuentra el vértice más cercano a un punto dado en una malla específica.

Casos de Prueba

Punto dentro de la malla:

Entrada: v_cercano 0.5 0.5 0.5 malla_cubo

Salida esperada: El vértice más cercano y su distancia al punto.

```
$ v_cercano 0.5 0.5 0.5 malla_de_un_cubo  
El vertice mas cercano es (0, 0, 0) a una distancia de 0.866025.
```

Malla no existente:

Entrada: v_cercano 1 1 1 malla_no_existente

Salida esperada: Mensaje de error indicando que la malla no está cargada.

```
$ v_cercano 1 1 1 malla_no_existente  
El objeto malla_no_existente no ha sido cargado en memoria.
```

7. v_cercano <px> <py> <pz> global

Encuentra el vértice más cercano a un punto de todas las mallas cargadas.

Caso de Prueba

Con varias mallas cargadas:

Entrada: v_cercano 0.5 0.5 0.5 global

```
$ v_cercano 0.5 0.5 0.5 global  
El vertice mas cercano global es (0, 0, 0) a una distancia de 0.866025.
```

Salida esperada: El vértice más cercano globalmente y su distancia.

8. descargar <nombre_malla>

Elimina una malla cargada del sistema.

Casos de Prueba

Malla existente:

Entrada: descargar malla_cubo

Salida esperada: Mensaje indicando que la malla se eliminó correctamente.

```
$ descargar malla_de_un_cubo  
El objeto malla_de_un_cubo ha sido eliminado de la memoria.
```

Malla no existente:

Entrada: descargar malla_no_existente

Salida esperada: Mensaje de error indicando que la malla no está en memoria.

```
$ descargar malla_no_existente  
El objeto malla_no_existente no ha sido encontrado en memoria.
```

9. ruta_corta_centro <i1> <nombre_malla>

Calcula la ruta más corta desde un vértice al centro geométrico de la malla.

Casos de Prueba

Índice válido y malla existente:

Entrada: ruta_corta_centro 0 malla_cubo

Salida esperada: Secuencia de vértices desde el índice hasta el centro.

```
$ ruta_corta_centro 0 malla_cubo  
La ruta más corta desde el vértice 0 hasta el centro de la malla es a través del vértice 0 con una distancia de 0.866025.  
$
```

Malla no existente:

Entrada: ruta_corta_centro 0 malla_no_existente

Salida esperada: Mensaje de error indicando que la malla no está cargada.

```
$ ruta_corta_centro 0 malla_no_existente  
El objeto malla_no_existente no ha sido cargado en memoria.
```

10. ruta_corta <i1> <i2> <nombre_malla>

Calcula la ruta más corta entre dos vértices en una malla.

Casos de Prueba

Índices válidos y malla existente:

Entrada: ruta_corta 0 7 malla_cubo

Salida esperada: Secuencia de vértices desde el índice i1 al i2.

```
La distancia mas corta entre los vertices 0 y 7 es 1.41421.  
El camino más corto es: 0 7
```

Malla no existente:

Entrada: ruta_corta 0 7 malla_no_existente

Salida esperada: Mensaje de error indicando que la malla no está cargada.

```
$ ruta_corta 0 7 malla_no_Existente  
El objeto malla_no_Existente no ha sido cargado en memoria.
```

11. ayuda <comando>

Proporciona una descripción de un comando específico.

Casos de Prueba

Comando válido:

Entrada: ayuda cargar

Salida esperada: Descripción del comando cargar.

```
$ ayuda cargar  
Comando 'cargar':  
Carga una malla desde un archivo especificado. El archivo debe contener los vertices y las caras de la malla.  
Uso: cargar <nombre_archivo>
```

Comando no válido:

Entrada: ayuda comando_invalido

Salida esperada: Mensaje indicando que el comando no es reconocido.

```
$ ayuda comando_invalido  
Comando no reconocido. Usa 'ayuda' para ver la lista de comandos disponibles.
```

12. salir

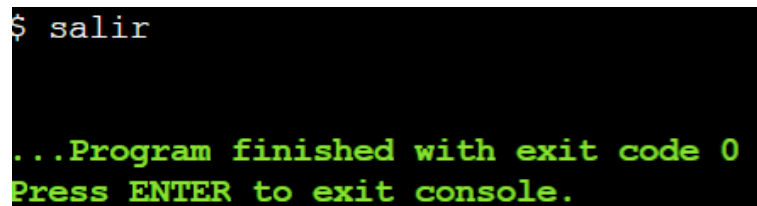
Termina la aplicación.

Casos de Prueba

Comando ejecutado:

Entrada: salir

Salida esperada: El programa se cierra correctamente.



```
$ salir

...Program finished with exit code 0
Press ENTER to exit console.
```

Conclusiones:

1. **Gestión eficiente de los datos:** El uso del Árbol KD (KDTree) para organizar y buscar vértices demuestra que el proyecto está diseñado para manejar datos espaciales de manera eficiente. Este enfoque permite realizar búsquedas rápidas, como encontrar el vértice más cercano, lo que es esencial para aplicaciones 3D.
2. **Modularidad:** La estructura del código, que separa funcionalidades en diferentes clases como kdtree, vertice, y malla, promueve la modularidad. Esto hace que el proyecto sea extensible, permitiendo agregar nuevas funcionalidades sin modificar significativamente el núcleo del programa.
3. **Aplicación de modelaje 3D:** Este proyecto permite manejar objetos 3D, con lo cual puede tener diferentes aplicaciones como para el modelado en video juegos, para la impresión en 3D o para la creación de objetos de cualquier tipo de industria.
4. **Interfaz del usuario:** El proyecto tiene una interfaz amigable con el usuario para su entendimiento, incluso teniendo una opción de ayuda la cual muestra el correcto funcionamiento del comando.
5. **Expansión:** Este proyecto puede ser una base para uno más grande, el cual con más recursos puede llegar a un mayor flujo de datos permitiendo estructuras más grandes y complejas. Expandiendo los horizontes de las aplicaciones de este.