Pontificia Universidad
JAVERIANA
Colombia

Entrega Segundo Proyecto Final

Estructuras de Datos

John Corredor Franco

Octubre 11/2024

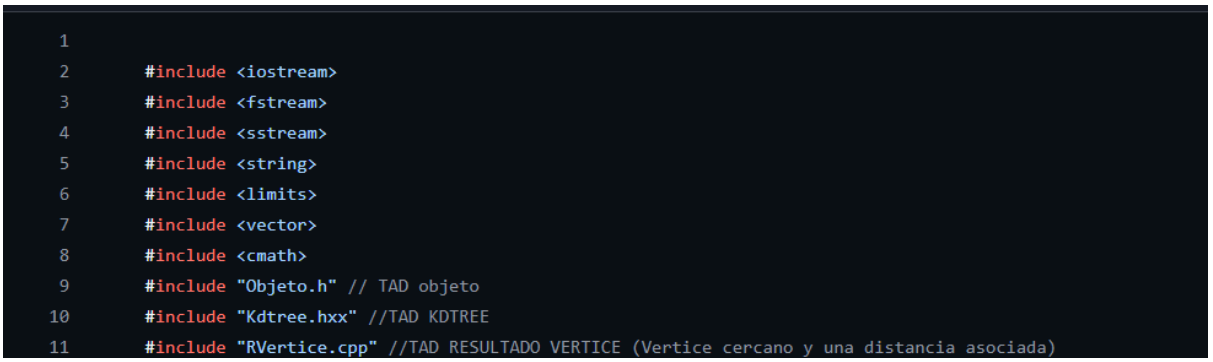
Daniel Rosas
Manuel Rincón
Santiago Camargo
Juan Sebastián Forero Moreno

1. Introducción

Este proyecto está diseñado para gestionar objetos geométricos tridimensionales. Utiliza una interfaz basada en comandos para cargar, manipular y realizar cálculos sobre estos objetos en memoria. Se utilizan varias estructuras de datos como listas, vectores y árboles KD para facilitar las operaciones. El sistema incluye funcionalidades como la creación de cajas envolventes, la búsqueda de vértices cercanos y el cálculo de rutas cortas entre puntos de los objetos. A lo largo del código, se implementan técnicas de manejo de errores, validación de archivos y organización modular, lo que lo hace escalable y fácil de mantener.

2. Explicación de la Estructura del Código

2.1 Bibliotecas Incluidas



```
1
2       #include <iostream>
3       #include <fstream>
4       #include <sstream>
5       #include <string>
6       #include <limits>
7       #include <vector>
8       #include <cmath>
9       #include "Objeto.h" // TAD objeto
10      #include "Kdtree.hxx" //TAD KDTREE
11      #include "RVertice.cpp" //TAD RESULTADO VERTICE (Vertice cercano y una distancia asociada)
```

imagen 1. Contiene las bibliotecas de nuestro main.

¿Por qué se usan estas bibliotecas?

- <iostream>: Permite la interacción con el usuario a través de la consola, utilizando std::cin para entradas y std::cout para salidas. Esto es fundamental para la interfaz de comandos que utiliza el sistema.
- <fstream>: Proporciona funciones para la lectura y escritura de archivos, que es necesario para cargar objetos desde archivos de texto y guardarlos.
- <sstream>: Es útil para el procesamiento de cadenas de texto, como dividir una línea de texto en palabras o convertir datos entre tipos.
- <string>: Para manipular cadenas de caracteres, ya que el nombre de los objetos y comandos son tratados como std::string.

- <limits>: Facilita obtener los valores máximos y mínimos de los tipos de datos numéricos, lo cual es esencial al calcular cajas envolventes (necesitamos comparar coordenadas con límites).
- <vector>: Se utiliza para manejar dinámicamente listas de vértices, aristas y caras, permitiendo que su tamaño varíe según las necesidades del objeto.
- <cmath>: Necesaria para operaciones matemáticas avanzadas como la raíz cuadrada, que es utilizada en los cálculos de distancias entre vértices.

Las bibliotecas adicionales como Objeto.h, Kdtree.hxx, y RVertice.cpp son archivos de cabecera específicos del proyecto, que definen los tipos de datos y estructuras para manejar objetos tridimensionales, búsquedas de vértices cercanos y resultados de esas búsquedas.

2.2 Declaración de Funciones

```
void cargarArchivo(std::string nombreArchivo, std::list<Objeto>& listadoObjetos);
bool verificacionObjeto(std::string nombreArchivo, std::list<Objeto>& listadoObjetos);
void listado(std::list<Objeto>& listadoObjetos); void imprimirListado(std::list<Objeto>& listadoObjetos);
void envolvente(std::string nombreObjeto);
void envolvente();
void descargar(std::string nombreObjeto, std::list<Objeto>& listadoObjetos);
void guardar(std::string nombreObjeto, std::string nombreArchivo, std::list<Objeto>& listadoObjetos);
Objeto encontrarObjeto(std::string nombreObjeto, std::list<Objeto>& listadoObjetos);
void salir();
```

imagen 2. Se muestra las funciones que se usaron para nuestro proyecto.

¿Por qué se declararon estas funciones?

El diseño modular del código facilita su mantenibilidad y escalabilidad. Cada función tiene un propósito bien definido, permitiendo que la lógica del programa esté distribuida de manera clara:

- cargarArchivo: Se encarga de leer un archivo y cargar los datos de un objeto en memoria.
- verificacionObjeto: Verifica si el archivo es válido, comprobando su formato y contenido antes de cargarlo.
- listado y imprimirListado: Muestran todos los objetos actualmente cargados en memoria.

- envolvente: Calcula la caja envolvente de un objeto o de todos los objetos cargados, lo que es útil para definir los límites de un objeto tridimensional.
- descargar: Elimina un objeto de la memoria.
- guardar: Guarda los datos de un objeto en un archivo.
- encontrarObjeto: Busca un objeto en la lista de objetos cargados.
- salir: Finaliza la ejecución del programa.

Este enfoque modular ayuda a que el código sea más fácil de depurar y de mantener, ya que cada funcionalidad está encapsulada en su propia función.

3. **Función main y Control del Flujo del Programa**

```
int main() {
  try {
      std::cout << "    ____________________________________________________________________" << std::endl;
      std::cout << "    |                     Bienvenid@ a nuestro proyecto                |" << std::endl;

      std::string comandoUsuario;
      std::vector<std::string> argumentosUsuario;
      std::string argumento;

      while (true) {
          std::cout << "    ____________________________________________________________________"<<std::endl;
          std::cout << "    |                                                                  |" << std::endl;
          std::cout << "    |     Por favor, ingrese el comando ayuda para ver los comandos disponibles    |" << std::endl;
          std::cout << "    |__________________________________________________________________|" << std::endl;
          std::cout << "$";
          std::getline(std::cin, comandoUsuario);

          std::istringstream stream(comandoUsuario);
          stream >> comandoUsuario;

          argumentosUsuario.clear();
          while (stream >> argumento) {
              argumentosUsuario.push_back(argumento);
          }
```

imagen 3. Función main para lo que se va a mostrar o preguntar en pantalla.

¿Por qué se diseñó así?

- El main actúa como el controlador principal que ejecuta el bucle de interacción con el usuario.
- std::getline se utiliza para leer toda la línea de entrada del usuario, lo cual permite que el programa maneje comandos y argumentos ingresados en una sola línea.

- std::istringstream convierte la línea completa en un flujo de texto que luego se divide en el comando principal y sus posibles argumentos. Esto es fundamental para un sistema de comandos flexible.

El uso de un bucle infinito while(true) asegura que el programa siga corriendo hasta que el usuario introduzca el comando "salir". Las estructuras como std::vector<std::string> permiten almacenar múltiples argumentos asociados a un comando (por ejemplo, cargar archivo.txt).

---

4. Manejo de Comandos y Control de Flujo

```
    }
} else if (comandoUsuario == "cargar") {
    if (argumentosUsuario.size() == 1) {
        cargarArchivo(argumentosUsuario[0], objetosPrograma);
    } else {
        std::cout<<"Error"<<std::endl;
        std::cout << "Uso incorrecto, use 'ayuda cargar' para más informacion.\n";
    }
```

imagen 4. Se muestra la condición para el usuario para el uso correcto del comando.

Cada comando ingresado por el usuario se maneja dentro del bucle principal. Aquí se muestra un ejemplo del manejo del comando cargar:

¿Por qué se usa de esta manera?

- Condicional if: Primero verifica si el comando es cargar. Luego, comprueba si se ha pasado un argumento (el archivo a cargar). Si todo está correcto, llama a la función cargarArchivo. Si no, muestra un mensaje de error y ayuda al usuario indicando el uso correcto del comando.
- Este patrón se repite para otros comandos como listado, descargar, envolvente, etc. De esta forma, el programa está preparado para responder adecuadamente a una variedad de entradas del usuario.

5. Función cargarArchivo y Validación de Objetos

```
//enunciado del proyecto 3
void cargarArchivo(std::string nombreArchivo, std::list<Objeto>& listadoObjetos){

  std::ifstream archivo(nombreArchivo);
  bool confirmacionObjeto;
  bool existenciaObjeto=false;

  if(!archivo.is_open()){
    std::cout<<"El archivo "<<nombreArchivo<< " no existe o es ilegible"<<std::endl;
    archivo.close();
  }else{
      confirmacionObjeto = verificacionObjeto(nombreArchivo, listadoObjetos);

      Objeto objetoAux;
      if(confirmacionObjeto == true){

            std::ifstream archivo(nombreArchivo);
            std::string linea;

            //SE OBTIENE EL NOMBRE DEL OBJETO
            if (std::getline(archivo, linea)) {
                objetoAux.fijarNombreObjeto(linea);
                std::cout<<"Se ha obtenido el nombre del objeto: "<<objetoAux.obtenerNombreObjeto()<<std::endl;
            }

              if(confirmacionObjeto == false){
                std::cout<<"El objeto no es valido";

              }
              if(existenciaObjeto == true){
                  std::cout<<"El objeto ya existe en memoria"<<std::endl;
              }

            std::list<Objeto>::iterator iteradorObj;
            iteradorObj = listadoObjetos.begin();

            for(;iteradorObj!=listadoObjetos.end();iteradorObj++){
                if(objetoAux.obtenerNombreObjeto() == iteradorObj->obtenerNombreObjeto()){
```

imagen 5. Función para cargar archivo y validar por medio de condiciones los objetos.

¿Por qué se usa así?

- Apertura del archivo: std::ifstream se usa para leer el archivo que contiene la información del objeto.
- Verificación del archivo: Si el archivo no se puede abrir, se muestra un error al usuario. De lo contrario, se llama a la función verificacionObjeto para asegurarse de que el archivo es válido.
- Proceso de lectura: Se utiliza std::getline para leer la primera línea, que contiene el nombre del objeto, y se asigna este valor al objeto recién creado.

Esta función está diseñada de manera robusta para manejar posibles errores, como archivos corruptos o formatos incorrectos, y garantiza que solo se carguen objetos válidos.

---

6. Función verificacionObjeto para Validación de Archivos

```
bool verificacionObjeto(std::string nombreArchivo, std::list<Objeto>& listadoObjetos){
    std::ifstream archivo(nombreArchivo);

    if (!archivo.is_open()) {
        std::cout<< "El archivo "<<nombreArchivo <<"no se pudo abrir (Desde verificacion objeto)."<<std::endl;
        return false;
    }

    std::string linea;

    //Se valida que el titulo del objeto no tenga
    //espacios en el primer renglón
    if (!std::getline(archivo, linea)) {
        std::cout<<"El archivo esta vacio (como mi corazon)"<<std::endl;
        return false;
    }

    if (linea.find(' ') != std::string::npos) {
        std::cout<<"El nombre del objeto tiene espacios"<<std::endl;
        return false;
    }

    //Revisar que está el número de vertices
    if (!std::getline(archivo, linea)) {
        std::cout<<"No existe el numero de vertices"<<std::endl;
        return false;
    }

    //Se revisa que realmente se esté ingrensando un número de vertices
    //que sea un entero positivo y que sea un número
    int numVertices;
    std::istringstream entrada(linea);
    if (!(entrada>>numVertices) || numVertices<=0) {
        std::cout<<"El numero de vertices no es un numero o no es una entrada valida"<<std::endl;
        return false;
```

imagen 6. Se muestra el contenido de la función y condiciones para la verificación para objeto.

¿Por qué es importante esta función?

- Manejo de errores de apertura: Asegura que el archivo puede abrirse antes de continuar.
- Validación del formato del archivo: Verifica que el nombre del objeto no tenga espacios, que el archivo contenga el número correcto de vértices y que este sea un número positivo. Estas validaciones protegen al programa de intentar cargar datos incorrectos o dañados.

7. Cálculo de la Caja Envolvente

```
void envolvente(std::string nombreObjeto) {
    // Buscar el objeto en la lista
    Objeto objetoEnMemoria;
    bool objetoEncontrado = false;
    // Recorrer la lista de objetos
    for (std::list<Objeto>::iterator it = objetosPrograma.begin(); it != objetosPrograma.end(); ++it) {
        if (it->obtenerNombreObjeto() == nombreObjeto) {
            objetoEnMemoria = *it;
            objetoEncontrado = true;
            break;
        }
    }
    //No encontro el objeto en la lista
    if (!objetoEncontrado) {
        std::cout << "(Objeto no existe) El objeto " << nombreObjeto << " no ha sido cargado en memoria." << std::endl;
        return;
    }

    //Se inicializa en un valor muy grande para que se pueda comparar con los demás
    float xmin = std::numeric_limits<float>::max();
    float xmax = -std::numeric_limits<float>::max();
    float ymin = std::numeric_limits<float>::max();
    float ymax = -std::numeric_limits<float>::max();
    float zmin = std::numeric_limits<float>::max();
    float zmax = -std::numeric_limits<float>::max();
```

imagen 7. Se muestra la función que se usó para caja envolvente.

¿Por qué se usa de esta manera?

- Búsqueda del objeto: Se recorre la lista de objetos cargados en memoria para encontrar el que tiene el nombre especificado por el usuario.
- Inicialización de límites: Los valores iniciales de xmin, xmax, ymin, etc., se establecen en valores extremos utilizando std::numeric_limits<float>. Esto permite que cualquier coordenada encontrada se compare correctamente y actualice los valores mínimos y máximos.

La función recorre todos los vértices del objeto para identificar los límites de la caja envolvente en los tres ejes (x, y, z).

8. Búsqueda de Vértices Cercanos con KD-Tree

```
        KdTree<double> kdTree(3);

        // Obtener las caras del objeto encontrado
        std::list<Cara> caras = itObj->obtenerCaras();
        for (std::list<Cara>::iterator itCara = caras.begin(); itCara != caras.end(); ++itCara) {
            std::list<Arista> aristas = itCara->obtenerListaAristas();
            for (std::list<Arista>::iterator itArista = aristas.begin(); itArista != aristas.end(); ++itArista) {
                std::list<Vertice> vertices = itArista->obtenerListaVertices();
                for (std::list<Vertice>::iterator itVertice = vertices.begin(); itVertice != vertices.end(); ++itVertice) {
                    // Convertir las coordenadas del vértice a un vector de doubles
                    std::vector<double> punto = {
                        static_cast<double>(itVertice->obtenerX()),
                        static_cast<double>(itVertice->obtenerY()),
                        static_cast<double>(itVertice->obtenerZ())
                    };
                    // Insertar el vértice en el Kd-Tree
                    kdTree.insertar(punto, nombre_objeto);
                }
            }
        }
```

imagen 8. Contiene las bibliotecas de nuestro main.

8.1. Comando: v_cercano px py pz nombre_objeto

```
}
////////////////////////////////COMPONENTE 2////////////////////////////////
void v_cercano(int px, int py, int pz, std::string nombre_objeto) {
    // Verificar si el objeto existe en memoria recorriendo la lista manualmente
    std::list<Objeto>::iterator itObj;
    bool objetoEncontrado = false;

    // Buscar el objeto en la lista de objetos cargados en memoria
    for (itObj = objetosPrograma.begin(); itObj != objetosPrograma.end(); ++itObj) {
        if (itObj->obtenerNombreObjeto() == nombre_objeto) {
            objetoEncontrado = true;
            break; // Salir del bucle al encontrar el objeto
        }
    }

    if (!objetoEncontrado) {
        // El objeto no fue encontrado
        std::cerr << "El objeto " << nombre_objeto << " no ha sido cargado en memoria." << std::endl;
        return;
    }

    // Crear un Kd-Tree para almacenar los vértices del objeto en 3D
    KdTree<double> kdTree(3);

    // Obtener las caras del objeto encontrado
    std::list<Cara> caras = itObj->obtenerCaras();
    for (std::list<Cara>::iterator itCara = caras.begin(); itCara != caras.end(); ++itCara) {
        std::list<Arista> aristas = itCara->obtenerListaAristas();
        for (std::list<Arista>::iterator itArista = aristas.begin(); itArista != aristas.end(); ++itArista) {
            std::list<Vertice> vertices = itArista->obtenerListaVertices();
            for (std::list<Vertice>::iterator itVertice = vertices.begin(); itVertice != vertices.end(); ++itVertice) {
                // Convertir las coordenadas del vértice a un vector de doubles
                std::vector<double> punto = {
                    static_cast<double>(itVertice->obtenerX()),
                    static_cast<double>(itVertice->obtenerY()),
                    static_cast<double>(itVertice->obtenerZ())
                };
```

Imagen 9. Se muestran la función que se usó para objetos encontrar objetos cercanos

En el proyecto se usó unos comandos están diseñados para interactuar con los objetos geométricos en memoria. A continuación, se mostrará en detalle el funcionamiento de tres comandos clave:

- **Entrada**:
    - px, py, pz: Coordenadas del punto al que se desea encontrar el vértice más cercano.
    - nombre_objeto: El nombre del objeto en el que se va a buscar el vértice.
- **Proceso**:
    - El programa primero verifica si el objeto nombre_objeto está cargado en memoria.
    - Si el objeto se encuentra, se recorren todas las caras y aristas del objeto para obtener una lista de todos sus vértices.
    - Se utiliza un árbol KD (KdTree), que es una estructura de datos eficiente para búsquedas en espacios multidimensionales, en este caso tridimensionales. Todos los vértices del objeto se almacenan en este árbol.
    - El árbol KD se utiliza para buscar el vértice que esté más cercano al punto (px, py, pz) utilizando la distancia euclidiana como métrica.
    - Se calcula la distancia entre el punto (px, py, pz) y el vértice más cercano.
- **Salida**:
    - El vértice más cercano y la distancia desde el punto (px, py, pz) hasta este vértice se muestran en la consola.

¿Por qué usar este comando?

Este comando es útil cuando necesitas identificar el vértice más cercano a un punto específico dentro de un objeto en particular. Esto puede ser útil para operaciones de modelado geométrico, análisis espacial o simplemente para comprender mejor la geometría del objeto.

8.2. Comando: v_cercano px py pz

Este comando también busca el vértice más cercano al punto tridimensional (px, py, pz), pero a diferencia del comando anterior, la búsqueda no se limita a un objeto específico. En lugar de ello, busca el vértice más cercano entre todos los objetos cargados en memoria.

Funcionamiento:

- **Entrada**:
    - px, py, pz: Coordenadas del punto al que se desea encontrar el vértice más cercano.
- **Proceso**:
    - El programa verifica si hay objetos cargados en memoria.
    - Recorre todos los objetos en memoria, y para cada objeto, extrae las caras, aristas y vértices.
    - Utiliza un árbol KD (KdTree) para almacenar todos los vértices de todos los objetos en memoria.
    - El árbol KD se emplea para buscar el vértice más cercano al punto (px, py, pz) en todo el conjunto de objetos.
    - Se calcula la distancia euclidiana entre el punto dado y el vértice más cercano.

- **Salida**:
    - El vértice más cercano y la distancia desde el punto (px, py, pz) hasta este vértice, considerando todos los objetos cargados en memoria, se muestran en la consola.

¿Por qué usar este comando?

Este comando es útil cuando no te interesa un objeto específico, sino que deseas buscar el vértice más cercano en todo el conjunto de objetos cargados. Puede ser de utilidad para análisis globales de un conjunto de objetos tridimensionales o para casos donde el contexto del objeto no es relevante para la búsqueda.

---

3. Comando: v_cercanos_caja nombre_objeto

Este comando identifica los vértices más cercanos a las esquinas de la caja envolvente (bounding box) del objeto especificado. Una caja envolvente es el menor cubo que puede contener completamente al objeto en cuestión.

Funcionamiento:

- **Entrada**:
    - nombre_objeto: El nombre del objeto cuya caja envolvente se va a utilizar para buscar vértices cercanos.
- **Proceso**:
    - El programa verifica si el objeto ”nombre_objeto” está cargado en memoria.
    - Si el objeto existe, se calcula la caja envolvente de este objeto. Esto implica identificar las coordenadas mínimas y máximas de los vértices del objeto en los tres ejes (x, y, z).
    - Una vez identificados los límites, se determinan las 8 esquinas de la caja envolvente.
    - Para cada una de las esquinas, se ejecuta el proceso de búsqueda del vértice más cercano usando el árbol KD (KdTree), similar a los comandos anteriores.
    - Se calcula la distancia entre cada esquina de la caja envolvente y su vértice más cercano.
- **Salida**:
    - Para cada esquina de la caja envolvente, se muestra el vértice más cercano y la distancia entre la esquina y dicho vértice.

¿Por qué usar este comando?

Este comando es útil cuando deseas analizar la geometría externa del objeto y cómo se relacionan los vértices con las esquinas de su caja envolvente. Puede ser útil para entender cómo un objeto tridimensional se ajusta dentro de un espacio o para determinar si la geometría del objeto está concentrada en ciertas áreas.

Funcionamiento del codigo:

1. **Directorio donde se ubica el proyecto**

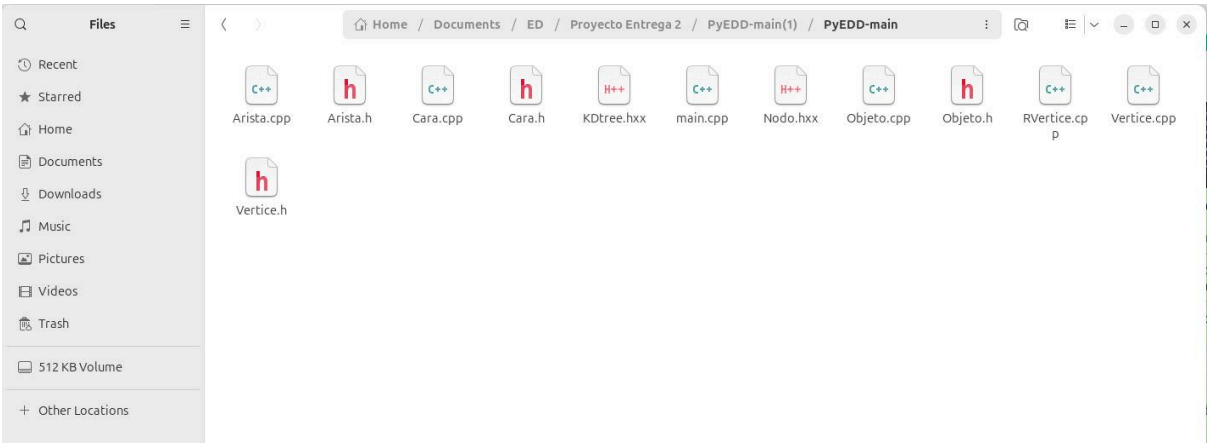


Imagen 10. Directorio

Este es el directorio dentro de la máquina con sistema operativo Linux Ubuntu donde se encuentra el proyecto y todos sus archivos de código necesarios, aquí también se encuentran los documentos con la información de los objetos que funcionan dentro del proyecto.

2. **compilación del código en Ubuntu:**

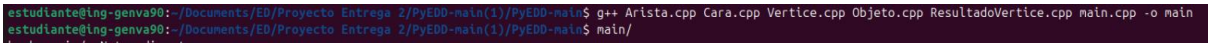


Imagen 11. Comando Buscar

El código que se escribe en la terminal de control de ubuntu donde primero se busca el repertorio donde están los archivos del proyecto.

3. **Ejecución del código en Ubuntu:**

Imagen 12. Terminal para ejecutar

El código que se escribe en la terminal de control de Ubuntu para realizar la ejecución de los archivos compilados en el paso anterior.

4. **Menú de ayuda:**

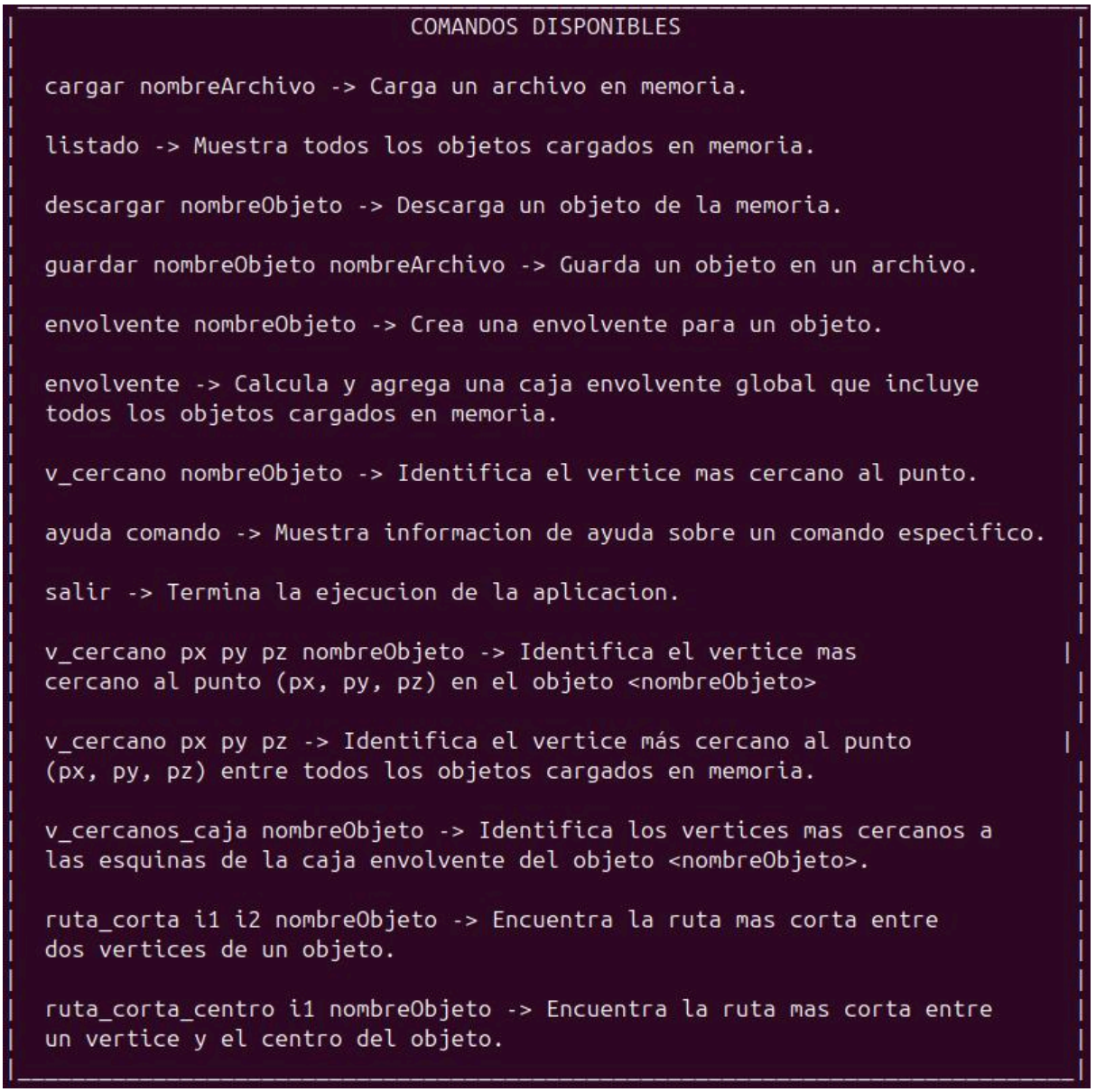


Imagen 13. Menú

Menú de ayuda donde se muestran todos los diferentes comandos del proyecto junto con una leve descripción de estos.

5. **v_cercano px py pz:**

```
$v_cercano 4 5 9
El vértice más cercano es: (0 0 0 ) del objeto: Mesh_0 con una distancia de: 11.0454
```

Imagen 14. Comando v_cercano

Lo que hace este comando es evaluar entre todos los objetos cargados en memoria y encontrar el vértice más cercano al punto x, y, y z indicados.

6. **v_cercano px py pz nombreObjeto:**

```
$v_cercano 1 2 3 Mesh_2
El vértice más cercano es: (0, 0, 0) del objeto Mesh_2 con una distancia de: 3.74166
```

Imagen 15. Comando v_cercano mesh_2

Lo que hace este comando es evaluar el objeto indicado dentro del comando y encontrar cual es su vértice más cercano al punto x, y, z indicados.

7. **v_cercanos_caja nombreObjeto:**

```
$v_cercanos_caja Mesh_1
(Resultado exitoso) Los vértices del objeto Mesh_1 más cercanos a las esquinas de su caja envolvente son:
Esquina     Vertice     Distancia
1 (0, 0, 0) i1 (0, 0, 0) 4.94066e-324
2 (20, 0, 0) i2 (20, 0, 0) 4.94066e-324
3 (20, 20, 0) i3 (20, 20, 0) 4.94066e-324
4 (0, 20, 0) i4 (0, 20, 0) 4.94066e-324
5 (0, 0, 0) i5 (0, 0, 0) 4.94066e-324
6 (20, 0, 0) i6 (20, 0, 0) 4.94066e-324
7 (20, 20, 0) i7 (20, 20, 0) 4.94066e-324
8 (0, 20, 0) i8 (0, 20, 0) 4.94066e-324
```

Imagen 16. Comando v_cercano mesh_1

Lo que hace este comando es evaluar el objeto indicado dentro del comando y encontrar cuales son los vértices de dicho objeto más cercanos a las esquinas de su caja envolvente.

Conclusión del Diseño

Este proyecto está bien estructurado, con una clara separación de responsabilidades a través de funciones modulares. Se utilizan estructuras de datos avanzadas, como los árboles KD, para hacer búsquedas eficientes y se implementan mecanismos de validación y manejo de errores para asegurar que el sistema solo procese datos válidos. La interfaz de usuario es simple pero efectiva, y permite a los usuarios realizar una amplia gama de operaciones sobre objetos tridimensionales, incluyendo cargar, guardar, manipular y realizar cálculos geométricos.