Computer Graphic II
Assignment 1

She Chengying
2024E8018482019

Deadline: Oct 23, 23:59

Task: Implement **any algorithm** for calculating the 3D convex hull, but the time complexity of the algorithm should be less than or equal to $\mathcal{O}(n^2)$. The visualization for the input point set and the calculated 3D convex hull is necessary. Third-party libraries may only be utilized for auxiliary purposes, such as I/O and visualization.

Note: There are no restrictions on programming languages, but plagiarism will score zero. We will compare assignments from the previous courses and the same period.

1 Preparation

Firstly, I prepare some tool functions for data generation and visualization using *numpy* and *matplotlib* libraries. The code is following:

```
def generate_random_points(
    num_points=10, x_range=[0, 10], y_range=[0, 10], z_range=[0, 10]
) -> np.array:
    """ Generate random points within specified ranges """
    x = np.random.uniform(x_range[0], x_range[1], num_points).reshape(-1, 1)
    y = np.random.uniform(y_range[0], y_range[1], num_points).reshape(-1, 1)
    z = np.random.uniform(z_range[0], z_range[1], num_points).reshape(-1, 1)
    points = np.concatenate([x, y, z], axis=1)
    points = [Point(p[0], p[1], p[2]) for p in points.tolist()]
    return points

def plot_points(points, hull_points=None, show_gt=False):
    """ Visualization of 3D points and convex hull """
    if points is None or len(points) == 0:
        raise ValueError("No points to plot")
    fig = plt.figure()
    ax = fig.add_subplot(111, projection="3d")
    ax.scatter(points[:, 0], points[:, 1], points[:, 2],
```

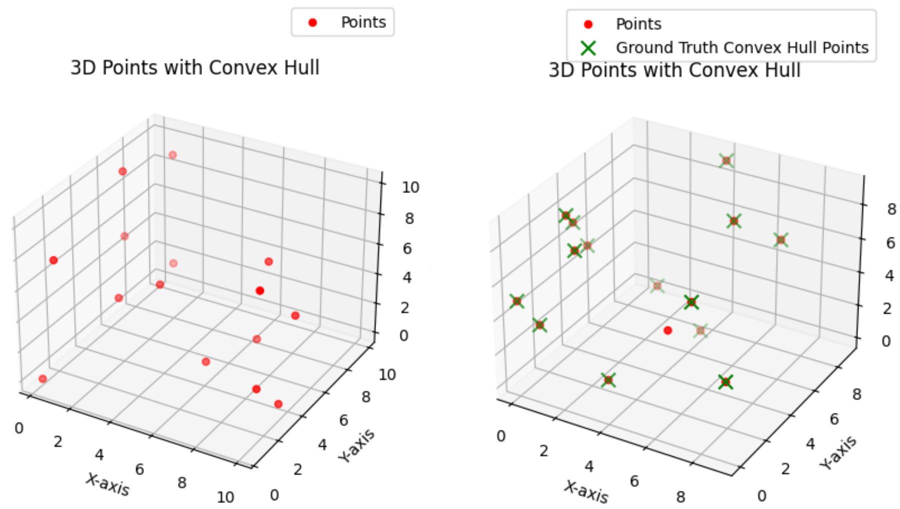
```python
               color="red", label="Points")

    if hull_points is not None and len(hull_points) > 0:
        ax.scatter(
            hull_points[:, 0],
            hull_points[:, 1],
            hull_points[:, 2],
            color="blue",
            label="Computed Convex Hull Points",
            s=60,
            marker="o",
        )

    if show_gt:
        def compute_convex_hull_with_scipy(points):
            """ using scipy library to compute the ground truth convex hull """
            convex_hull = ConvexHull(points)
            hull_points = [points[i] for i in convex_hull.vertices]
            return np.array(hull_points)
        gt_points = compute_convex_hull_with_scipy(points)
        ax.scatter(
            gt_points[:, 0],
            gt_points[:, 1],
            gt_points[:, 2],
            color="green",
            label="Ground Truth Convex Hull Points",
            marker='x',
            s=80)

    ax.set_xlabel("X-axis")
    ax.set_ylabel("Y-axis")
    ax.set_zlabel("Z-axis")
    ax.set_title("3D Points with Convex Hull")
    plt.legend(bbox_to_anchor=(1, 1.2), loc='upper right')
    plt.show()
```

Using the code above, I could quickly generate any list of 3D points within a specified range of coordinates, there is a example in the Fig. 1



(a) 3D points

(b) 3D points and true Convex Hull

Figure 1: Randomly generated 3D points and the true convex hull are labeled with different styles of dots.

2 QuickHull Algorithm

The code below is the implementation of the QuickHull Algorithm which has the time complexity of $O(n \log n)$.

First, I define some data structures to store the points, edges and faces:

```

class Edge:
    def __init__(self, p1, p2):
        self.p1 = p1
        self.p2 = p2

    def __hash__(self):
        return hash((self.p1, self.p2))

    def __eq__(self, other):
        return (self.p1 == other.p1 and self.p2 == other.p2) or (
            self.p1 == other.p2 and self.p2 == other.p1
        )


class Point:
    def __init__(self, x=None, y=None, z=None):
```

```
            self.x = x
            self.y = y
            self.z = z

    def __sub__(self, pointX):
        return Point(self.x - pointX.x, self.y - pointX.y, self.z - pointX.z)

    def __add__(self, pointX):
        return Point(self.x + pointX.x, self.y + pointX.y, self.z + pointX.z)

    def length(self):
        return math.sqrt(self.x**2 + self.y**2 + self.z**2)

    def __str__(self):
        return f"Point({self.x}, {self.y}, {self.z})"

    def __hash__(self):
        return hash((self.x, self.y, self.z))

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y and self.z == other.z


class Plane:
    def __init__(self, p1, p2, p3):
        self.p1 = p1
        self.p2 = p2
        self.p3 = p3
        self.normal = None
        self.distance = None
        self.calcNorm()
        self.to_do = set()
        self.edge1 = Edge(p1, p2)
        self.edge2 = Edge(p2, p3)
        self.edge3 = Edge(p3, p1)

    def calcNorm(self):
        v1 = self.p1 - self.p2
        v2 = self.p2 - self.p3
        normal = cross_product(v1, v2)
        length = normal.length()
        normal.x = normal.x / length
        normal.y = normal.y / length
        normal.z = normal.z / length
        self.normal = normal
        self.distance = dot_product(self.normal, self.p1)

```

```
    def dist(self, pointX):
        return dot_product(self.normal, pointX - self.p1)

    def get_edges(self):
        return [self.edge1, self.edge2, self.edge3]

    def calculate_to_do(self, points, temp=None):
        if temp != None:
            for p in temp:
                dist = self.dist(p)
                if dist > EPSOLON:
                    self.to_do.add(p)
        else:
            for p in points:
                dist = self.dist(p)
                if dist > EPSOLON:
                    self.to_do.add(p)

    def __eq__(self, other):
        return (
            self.p1 == other.p1
            and self.p2 == other.p2
            and self.p3 == other.p3
            or self.p1 == other.p1
            and self.p2 == other.p3
            and self.p3 == other.p2
            or self.p1 == other.p2
            and self.p2 == other.p3
            and self.p3 == other.p1
            or self.p1 == other.p2
            and self.p2 == other.p1
            and self.p3 == other.p3
            or self.p1 == other.p3
            and self.p2 == other.p2
            and self.p3 == other.p1
            or self.p1 == other.p3
            and self.p2 == other.p1
            and self.p3 == other.p2
        )

    def __hash__(self):
        return hash((self.p1, self.p2, self.p3))
```

The QuickHull algorithm is conceptually similar to the quicksort algorithm, as it divides the problem recursively and constructs the convex hull steps by step. The following code is my implementation:

```python

def set_correct_normal(possible_internal_points, plane):
    """Set the correct noraml orientation of the plane"""
    for point in possible_internal_points:
        dist = dot_product(plane.normal, point - plane.p1)
        if dist != 0:
            if dist > EPSOLON:
                plane.normal.x = -1 * plane.normal.x
                plane.normal.y = -1 * plane.normal.y
                plane.normal.z = -1 * plane.normal.z
                return

def cross_product(v1, v2):
    """cross product of two vectors"""
    x = (v1.y * v2.z) - (v1.z * v2.y)
    y = (v1.z * v2.x) - (v1.x * v2.z)
    z = (v1.x * v2.y) - (v1.y * v2.x)
    return Point(x, y, z)

def dot_product(v1, v2):
    """dot product of two vectors"""
    return v1.x * v2.x + v1.y * v2.y + v1.z * v2.z

def cal_horizon(list_of_planes, visited_planes, plane, eye_point, edge_list):
    """Calculate the horizon for an eye to make new faces"""
    if plane.dist(eye_point) > EPSOLON:
        visited_planes.append(plane)
        edges = plane.get_edges()
        for edge in edges:
            neighbour = adjacent_plane(list_of_planes, plane, edge)
            if neighbour not in visited_planes:
                result = cal_horizon(
                    list_of_planes, visited_planes,
                    neighbour, eye_point, edge_list
                )
                if result == 0:
                    edge_list.add(edge)

        return 1

    else:
        return 0


def adjacent_plane(list_of_planes, main_plane, edge):
    """Finding the adjacent plane of an edge"""
    for plane in list_of_planes:
```

```
        edges = plane.get_edges()
        if (plane != main_plane) and (edge in edges):
            return plane


def distance_point2line(A, B, P):
    """Calculate the distance of P to line AB"""
    AP = P - A
    AB = B - A
    return cross_product(AP, AB).length() / (AB.length() + EPSOLON)

def max_distance2line_point(points, A, B):
    """Calculate the maximum distant point to the line AB"""
    points = sorted(points, key=lambda p: abs(distance_point2line(A, B, p)))
    return points[-1]

def max_distance2plane_point(points, plane):
    """ Calculate the maximum distant point to the plane """
    points = sorted(points, key=lambda p: abs(plane.dist(p)))
    return points[-1]

def find_eye_point(plane, to_do_list):
    """ Calculate the maximum distant point to the plane """
    to_do_list = sorted(to_do_list, key=lambda p: abs(plane.dist(p)))
    return to_do_list[-1]

def get_extreme_points(points: List[Point]) -> Tuple[Point]:
    """Calculate the extreme points of each axis"""
    x_min = y_min = z_min = float("inf")
    x_max = y_max = z_max = -float("inf")
    num = len(points)
    for i in range(num):
        if points[i].x > x_max:
            x_max = points[i].x
            x_max_p = points[i]
        if points[i].x < x_min:
            x_min = points[i].x
            x_min_p = points[i]
        if points[i].y > y_max:
            y_max = points[i].y
            y_max_p = points[i]
        if points[i].y < y_min:
            y_min = points[i].y
            y_min_p = points[i]
        if points[i].z > z_max:
            z_max = points[i].z
            z_max_p = points[i]
```

```python
        if points[i].z < z_min:
            z_min = points[i].z
            z_min_p = points[i]
    return (x_max_p, x_min_p, y_max_p, y_min_p, z_max_p, z_min_p)

def quickhull_3d(points: List[Point]) -> List[Point]:
    extremes = get_extreme_points(points)
    # find the 2 most distant points
    maxi = -1
    initial_line = []
    for i in range(6):
        for j in range(i + 1, 6):
            dist = math.sqrt(
                (extremes[i].x - extremes[j].x) ** 2
                + (extremes[i].y - extremes[j].y) ** 2
                + (extremes[i].z - extremes[j].z) ** 2
            )
            if dist > maxi:
                initial_line = [extremes[i], extremes[j]]
    third_point = max_distance2line_point(
        points, initial_line[0], initial_line[1])
    first_plane = Plane(initial_line[0], initial_line[1], third_point)
    fourth_point = max_distance2plane_point(points, first_plane)
    possible_internal_points = [
        initial_line[0],
        initial_line[1],
        third_point,
        fourth_point,
    ]  # List that helps in calculating orientation of point
    second_plane = Plane(initial_line[0], initial_line[1], fourth_point)
    third_plane = Plane(initial_line[0], fourth_point, third_point)
    fourth_plane = Plane(initial_line[1], third_point, fourth_point)
    # Setting the orientation of normal correct
    set_correct_normal(possible_internal_points, first_plane)
    set_correct_normal(possible_internal_points, second_plane)
    set_correct_normal(possible_internal_points, third_plane)
    set_correct_normal(possible_internal_points, fourth_plane)

    first_plane.calculate_outside_points(points)
    second_plane.calculate_outside_points(points)
    third_plane.calculate_outside_points(points)
    fourth_plane.calculate_outside_points(points)

    list_of_planes = []
    list_of_planes.append(first_plane)
    list_of_planes.append(second_plane)
    list_of_planes.append(third_plane)
```

```python
    list_of_planes.append(fourth_plane)

    any_left = True

    while any_left:
        any_left = False
        for working_plane in list_of_planes:
            if len(working_plane.to_do) > 0:
                any_left = True
                eye_point = find_eye_point(
                    working_plane, working_plane.to_do
                )

                edge_list = set()
                visited_planes = []

                cal_horizon(
                    list_of_planes, visited_planes,
                    working_plane, eye_point, edge_list
                )

                for internal_plane in visited_planes:
                    # remove the internal planes
                    list_of_planes.remove(internal_plane)

                for edge in edge_list:  # make new planes
                    new_plane = Plane(edge.p1, edge.p2, eye_point)
                    set_correct_normal(possible_internal_points, new_plane)

                    temp_to_do = set()
                    for internal_plane in visited_planes:
                        temp_to_do = temp_to_do.union(internal_plane.to_do)

                    new_plane.calculate_outside_points(points, temp_to_do)

                    list_of_planes.append(new_plane)

    final_vertices = set()

    for plane in list_of_planes:
        final_vertices.add(plane.p1)
        final_vertices.add(plane.p2)
        final_vertices.add(plane.p3)

    return list(final_vertices)
```

The function *get_extreme_points()* computes 6 extreme points along each axis (X, Y, Z) to find the minimum and maximum points. These points are important as they will help in forming the initial convex boundary.

Find the two most distant points from these extremes as the starting line, then find the point furthest away from this line as the third point to form a plane. After that find the point furthest from this plane as the fourth point to form a tetrahedron.

The function *set_correct_normal()* adjusts the orientation of the planes' normals, ensuring that they point outwards. This is necessary for correctly identifying which points lie outside the current convex hull.

The *calculate_outside_points()* function identifies the points outside each triangular face of the convex hull. These are the points that need to be further considered for expanding the hull.

The algorithm expands the convex hull by recursively traversing the faces of the convex hull, calculating the **eye point** of each face, i.e., the point furthest from this face outside it, and then adding the point to the convex hull. During each iteration, the algorithm removes internal faces (those that are completely hidden by the **eye point**) and adds new external faces formed by the horizon and the eye point.

After all points have been processed, the set of triangular faces forms the convex hull. The vertices of these faces are collected as the final convex hull vertices.

3 Experimental Results

Use the pre-defined function *generate_random_points()* to generate 20 3D points randomly and calculate the convex hull using the function *quickhull_3d()*, then plot these points on the same graph with the convex hull calculated using the algorithm provided in the **scipy.spatial.ConvexHull**. The results is in the Fig. 2

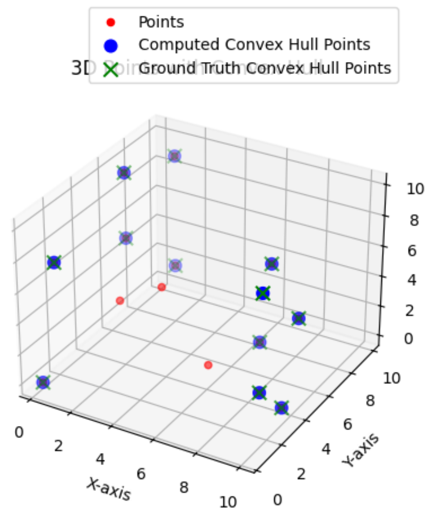


Figure 2: The randomly generated points, the convex hull points calculated using my customized quickhull algorithm and the convex hull points calculated using scipy library as the ground truth.