3D Computational Geometry and Computer Vision 2023-2024: Path Planning

Ntagkas Alexandros, 1083874

July 15, 2024

Contents

1 Introduction

The goal of this project is to develop a robust pipeline for navigating and path planning through dense indoor environments. Utilizing the dataset from the Stanford Building Parser Dataset, which is based on realistic measurements obtained via LiDAR and depth cameras, this project aims to enhance indoor navigation capabilities.

Numerous steps should be followed in order to achieve that, however a general framework, on which I initially developed the ideas that I am going to showcase is as follows:

1. Detect all walls of the room
2. Cluster the different objects
3. detect the door
4. Perform path planning and avoid the obstacles

2 Plane Detection- 3D Mesh

From the dataset one can obtain either the Point Clouds of each room directly or the 3D Mesh (cropping the respective part of the whole mesh). It is very interesting to view the problem of plane/wall detection from both perspectives. RANSAC (Random Sample Consensus) is an iterative method used for plane

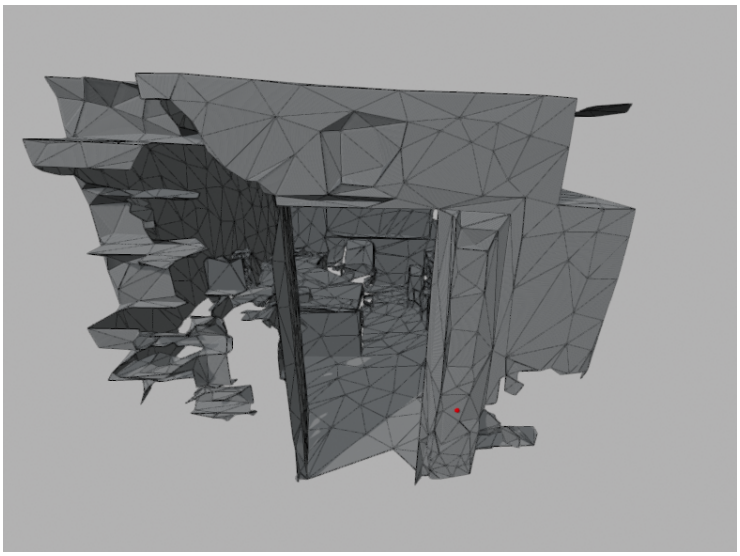


Figure 1: Room 1 Mesh

detection in 3D point clouds by randomly selecting subsets of points to fit a plane model and identifying inliers that conform to the model within a specified tolerance. It is robust against outliers, making it effective in noisy environments for accurately detecting planes amidst significant data irregularities.

I had the idea to implement RANSAC on a 3D mesh, **however** using the number of vertices as a metric of how good a plane is, may fail in cases where a wall is represented with 2 triangles for example (4 vertices). A more sophisticated approach is using the area and not the vertices:

Algorithm 1 RANSAC Mesh

function RANSAC(mesh,threshold,iterations)
Initialize $best\ plane, best\ inliers \leftarrow \emptyset$.
Initialize $best\ area \leftarrow -\infty$.
for iterations **do**
  sample random triangle t from mesh
  compute the plane it creates p
  Calculate which triangles from mesh that belong to p (inliers)
  area=$\sum_{triangle \in inliers} area(t)$
  if $area > best\ area$ **then**
    $best\ plane = p$
    $best\ inliers = inliers$
    $best\ area = area$
  end if
end for

the area of a triangle is $\frac{1}{2}|AB| \cdot |AC| \sin(\theta) = \frac{1}{2} n_t$

to **compute the plane given the triangle** we know that the normal vector of the triangle is the same as the plane's :

$$ax + by + cz + d = 0 \Rightarrow n \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} + d = 0$$

plugging in 1 vertex of the triangle v_1 we solve for d and get the plane equation.

to **check whether a triangle belongs in the plane** :

we need the check if the normal of the triangle n_t and the plane n_p point in the same or opposite directions,so:

$$|n_t \cdot n_p| = 1$$

and if one point of the triangle **satisfies** the plane equation :

$$v_1 \cdot n_p + d = 0$$

RANSAC is only part of plane detection. We need to implement RANSAC iteratively , **removing** the inliers we find in the current step.

After we do that we will be left with N planes (the number of which we control) because some rooms have less than 6 planes and everything that is left is considered an object.

Not all the planes are walls when $N > 6$, therefore some **postprocessing** is done. The planes that are horizontal can be the floor,the ceiling or potentially objects(table ,desk,etc.), therefore I keep only the planes ,whose intersection with the y axis is the minimum and maximum of all planes **(y axis for me is looking up).** To remove them we look where these planes intersect the y axis. Given the plane equations this is $a*0+b*1+c*0+d=0 \Rightarrow -d/b$, so I keep the ones that maximise and minimize this value and classify the rest as objects.

In **room1** of the examples this looks like:

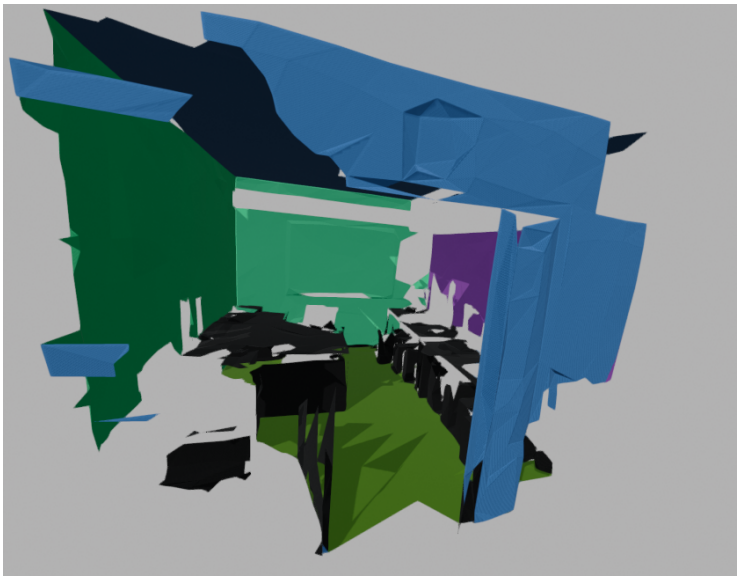


Figure 2: Colorful:Walls , Black: objects

3 Plane Detection- Point Cloud

Before we begin with the plane detection we need to create a Point Cloud from the given mesh. We would like to do that by uniformly sampling the triangles of the mesh.

3.1 Uniform Sampling

We iterate over all the triangles and given the **density (d)** ,which indicates the number of points per unit of surface, and the area of the triangle from

$$a = \frac{1}{2} \|AB \times BC\|$$

we can find the number of point per triangle as : $\lfloor a \cdot d \rfloor$

To sample such points uniformly we will use the equation from [1] :

$$P = (1 - \sqrt{b})A + (\sqrt{b}(1 - a))B + (a\sqrt{b})C$$

where $a, b \sim U[0, 1]$

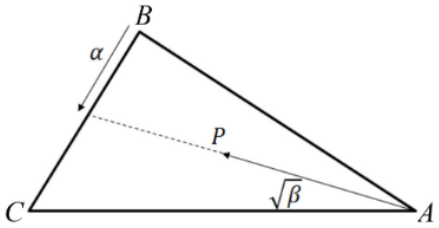
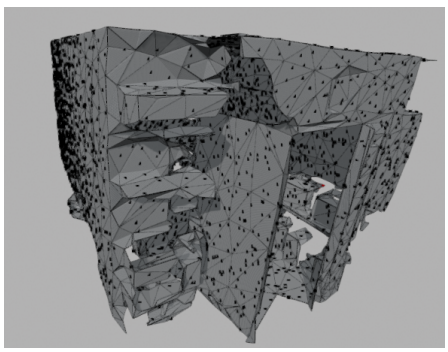


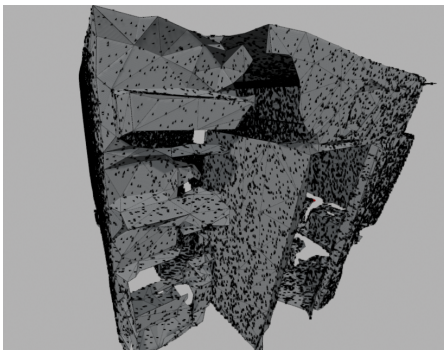
Figure 3: Sampling

a represents the percentage of the edge BC that we want the point to lay and $\sqrt{b}$ represents the percentage from A to the edge BC.

The time complexity is $O(T)$ where T is the number of triangles in the 3D Mesh, however I am using the **multiprocessing library** to utilize multiple cores and the result is almost instant.



(a) Density=100



(b) Density=700

3.2 Plane Detection

Now that we can sample the mesh to get the respective PCD(Point Cloud), we will implement the RANSAC algorithm but now with points instead of triangles.

Algorithm 2 RANSAC Point Cloud

```
function RANSAC(Point Cloud,threshold,iterations)
Initialize best plane, best inliers ← ∅.
Initialize best area ← −∞.
for iterations do
  sample 3 random points v1, v2, v3 from the PCD
  compute the plane they create p
  Calculate which points belong to p (inliers)
  if |inliers| > |best area| then
    best plane = p
    best inliers = inliers
  end if
end for
```

to **compute the plane the points** create we find the plane normal vector as $n_p = (v_3 - v_1) \times (v_2 - v_1)$ and $d = -n_p \cdot v_1$.

to calculate which points **belong to** p we calculate the distance of the point v_0 to p with

$$\frac{|Ax_0 + By_0 + Cz_0 + D|}{\sqrt{A^2 + B^2 + C^2}}$$

and if the distance is smaller than a threshold(0.02-0.03) we consider $v_0 \in p$.

Finally, if the number of current **inliers** is bigger than the number of best inlier points we consider the new plane as the best plane.

RANSAC is only part of plane detection. We need to implement RANSAC

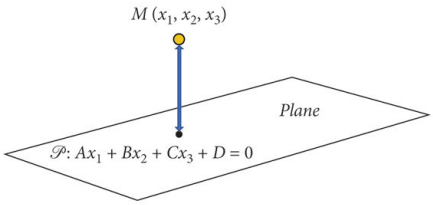


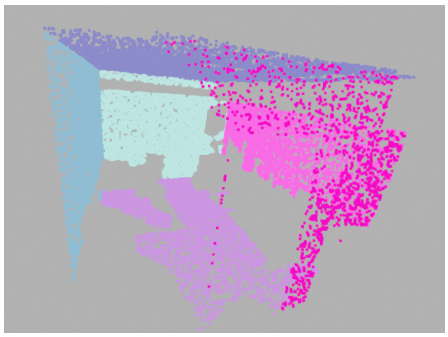
Figure 5: Point to plane distance

iteratively , **removing** the inliers we find in the current step.

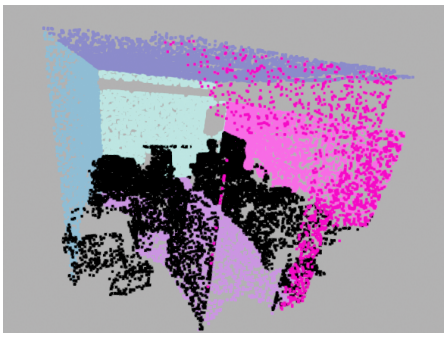
After we do that we will be left with N planes (the number of which we control) because some rooms have less than 6 planes and everything that is left is considered an object.

Not all the planes are walls when $N > 6$, therefore some **postprocessing** is done.

Like in section 2 we remove horizontal planes that are not the ceiling or the floor. We also remove the top **quarter** of the room, which usually has left over lights which cannot be detected with plane detection and we consider noise. Furthermore, we make the **assumption** that the all walls are vertical, so their normal vector is perpendicular to the y axis $\Rightarrow |n_p \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = 0|$. After this post processing, parts of the PCD that are classified as objects may have points that belong to some wall. To resolve that, I iterate all the planes and objects and if some object point belongs to some plane we add it to this plane.



(a) Planes



(b) Objects in black

3.3 Comparing with mesh algorithm

Both methods have the same structure. The time complexity of the mesh method is $O(T * m)$ where m are the iterations at each RANSAC run and T the number of triangles. For the PCD method it is $O(V * m)$ where V are the points of the PCD. Obviously with the sampling I used $V > T$ and therefore PCD method is slower. However, it has the advantage that it can detect details in the geometry of the room (like a thin part of the wall) that the mesh cannot. So we will be continuing the analysis with PCD data.

4 Object Clustering

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is an effective algorithm for object segmentation in PCDs, as it clusters points based on their density, enabling the identification of distinct object regions without needing prior knowledge of the number of objects. This capability allows for robust segmentation of objects with varying shapes and sizes, which is very important in our application so that we can identify the obstacles to path plan around.

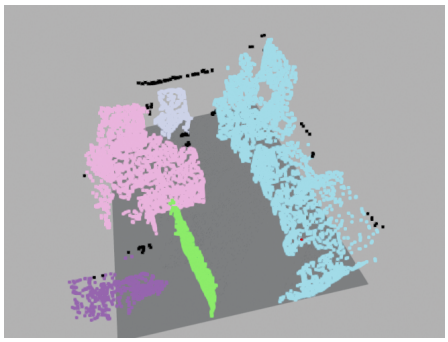
Algorithm 3 DBSCAN

```
function DBSCAN(points,eps, min points)
Initialize clusters = [ ].
Initialize C = 0.
for point in points do
  if point not visited then
    Neighbors ← points in radius eps from point
    if |Neighbors| < min points then
      point is considered noise
    else
      C = C + 1
      initialize Q ← Neighbors
      for neighbor in Neighbors do
        if labeled as noise then
          add to C cluster
        else if not visited then
          P_n ← points in radius eps from neighbor
          if |P_n| < min points then
            Q+ = P_n
          end if
        end if
      end for
    end if
  end if
end for
```

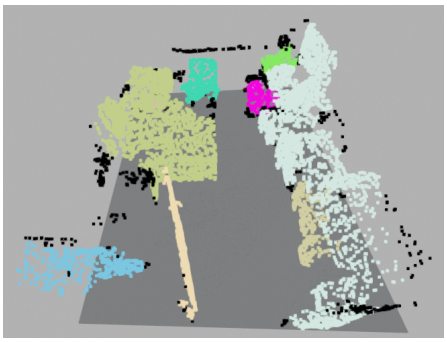
In the algorithm above that I have implemented we can see that we need to query points inside a radius eps of some point p. This has time complexity $O(n)$ if done brute force.

I implement this query with a **KDTree** which has $O(nlogn)$ construction time and $O(logn)$ query time complexity.

We can directly implement the above to the objects PCD:



(a) eps=0.2 , min points =30



(b) eps=0.2 , min points =55

We can see that increasing the minimum points, creates more clusters that are denser and at the same time classifies more points as noise For the purposes of path planning either one will work.

5 Door Detection

Detecting the door is crucial for path planning in order to have the right target/end point for path planning. Each room can have one or more doors.

5.1 Preliminaries

Before I get into the main algorithm we will need to establish 3 techniques:

- Project point on plane
- Sort points on a 3d Line
- Convex Hull Graham Scan

5.1.1 Project point onto plane

Given a point/points we can project them on a plane as follows:

1. take the unit normal vector of the plane as $\bar{n} = \frac{n}{\|n\|}$
2. find the distance of the point to the plane dist=$p \cdot \bar{n} + d$
3. The projected point will be the point that is has distance *dist* in the normal direction : $projected = p - dist * \bar{n}$

5.1.2 Sort points on 3d Line

To sort points on a 3d line in $O(nlogn)$:

1. take 2 points on the line v_1, v_2 and compute the vector $v = v_2 - v_1$(the red vector).
2. calculate the angle between v and all the points in $O(n)$
3. sort the points based on the angle in $O(nlogn)$.

5.1.3 Convex Hull Graham Scan

I will be using the Graham scan algorithm for extracting the convex hull. The algorithm iteratively adds points to the convex hull as long as they form a left turn. If they form a left turn we remove points until the condition is satisfied. We developed this algorithm in detail in the respective Lab exercise.

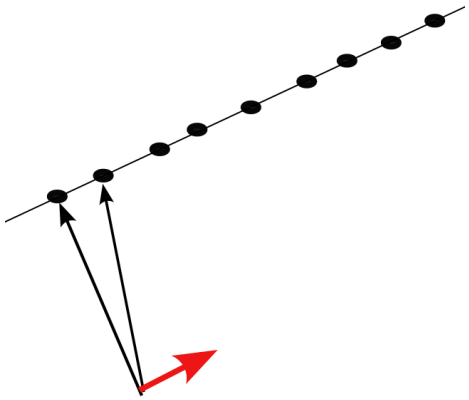
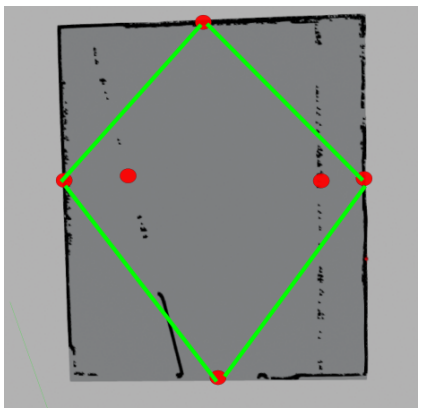


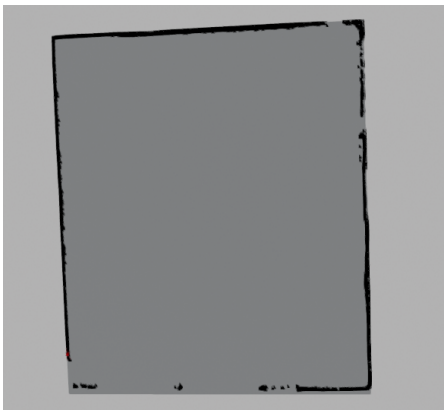
Figure 8: 3d line

5.2 Door Finding

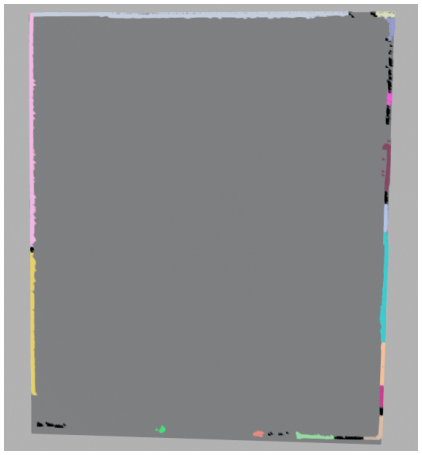
1. We find the outside walls:
   - project all walls on the floor
   - sort their points (supposing they belong in a Line in 3D)
   - find their endpoints and take their midpoints.
   - Create the Convex Hull of the midpoints: **The walls that belong to the CH are outside walls.**
2. Remove the top half/quarter of their points.
3. Project the rest of the outside wall points on the floor
4. Cluster the projected points with DBSCAN
5. Sort the clustered points with 5.1.2.
6. Look for gaps between clusters that are door-sized.



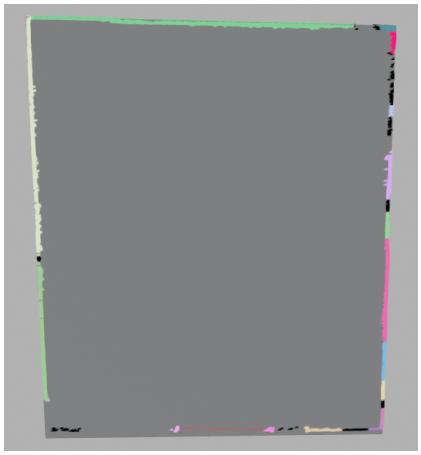
(a) Midpoints of planes in red, CH in green



(b) Outside walls



(a) Clustered outside walls



(b) Door detected (red line)

6 Path Planning

6.1 Introduction

In this section we will discuss how we can use what we achieved before to create a path that connects 2 points. In order for this implementation to be realistic we will suppose that the goal is to create a path for a cylindrical robot of radius r.

To achieve that we will use a visibility graph approach discussed in the book [3, de Berg et all] . For this approach to be implementable we will need to do some preprocessing on our objects to get their outline(a-shapes)[5], [**?**], and use the Minkowski Sum [4]to treat our robot as a point in 2d.

6.2 Obstacle preprocessing

Given a PCD that contains only the objects from 3.2 we will do the following:

1. **Project** all the objects on the floor
2. **Cluster** them with DBSCAN
3. Find their outlines with **a-shapes**
4. Perform **Minkowski sum** on their outlines
5. **Merge** obstacles that have intersecting minkowski sums

The steps 1 and 2 are well discussed in the previous sections.

6.2.1 alpha-shapes

Algorithm 4 alpha-shapes

function a-shapes(points, alpha)
$T \leftarrow Delaunay(points)$.
Initialize $edges \leftarrow \emptyset$
for $triangle \in T$ **do**
  if circumcircle radius$\leq$ *alpha* **then**
    add the sides of triangle to edges.
  end if
end for
remove duplicate edges
return edges

to perform **Delaunay triangulation** I use the **scipy.spatial.Delaunay**. The circumcircle radius is given from:

$$R = \frac{abc}{4A}$$

,where a, b, c are the side length and A is the area of the triangle

This algorithm works as stated in [5]. The main idea is that we only have to look for the outline in the faces of the Delaunay triangulation. On top of that the smaller the alpha parameter the more detail we have and when a=0 we approach the PCD itself. When $a \to \infty$ we approach the Convex Hull.

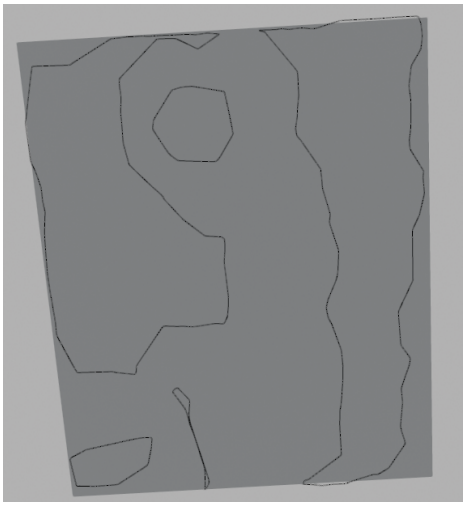


Figure 11: a-shapes on objects

6.2.2 Minkowski Sum

Minkowski sum is really important in path planning. It works as a transform that augments the objects in some sense so that we can work with a point-shaped robot.

$$S = A \oplus B = \{a + b \mid a \in A, b \in B\} \tag{1}$$

in our case A will be the points on the outline of the objects and B is a circle (representing the robot).

s As stated in [3],and many other sources, when both sets are convex the equation 1 holds. However, when A is non-convex, like in our case, we need to use theorem 13.10 from [3]:

$$\mathcal{P} \oplus \mathcal{R} = \bigcup_{i=1}^{n-2} t_i \oplus \mathcal{R}. \tag{2}$$

for non-convex polygon P and a convex polygon R and $t_1, \ldots, t_{n-2}$ is the triangulation of P.

For our application and to improve the speed of this step, I will do the following:

- If an object has less that x points I will be using the convex hull as the outline and use 1 for the minkowski sum.

- Else I will be using a-shapes for the outline and 2 for minkowski.

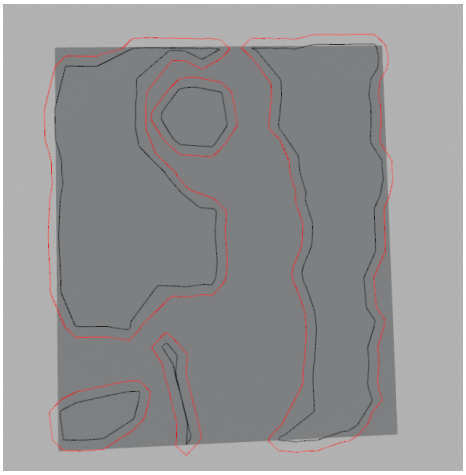


Figure 12: minkowski sum in red

6.2.3 Interval Tree

Since we will need to intersect many edges with other edges from now on, I created a binary search tree called IntervalTree. This tree saves the edges of each polygon(obstacle) with respect to their lower y coordinate. This allows us to query edges that may intersect with a y-parallel line in $O(logn)$ time instead of $O(n)$ (in a brute force manner).

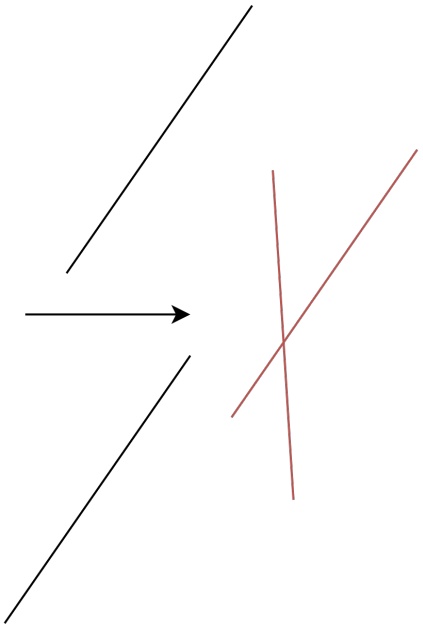


Figure 13: In red the edges the interval tree returns

6.2.4 Merging

A somewhat unusual case is when 2 minkowski sums intersect. Then the visibility graph algorithm we will discuss later will not work properly. To avoid that we will merge intersecting polygons.

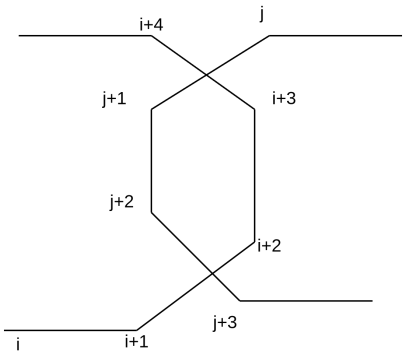


Figure 14: Merging

We first detect the intersecting edges here $x_1 = (i+1, i+2),(j+1, j+2)$ and $x_2 = (i+4, i+3),(j+1, j)$ .This is done in $O(logn)$ with the IntervalTree. Next we detect which points are inside.

For this I implemented a **polygon crossing** algorithm. Basically you look how many times the infinity y line from the given point intersects the polygon. If it is an even number it is outside else it is inside $O(logn)$.

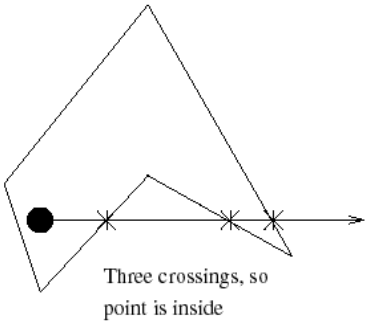


Figure 1 - Crossings Test

Figure 15: Merging

The vertices of the polygon are in order (from a-shapes) so we need to remove the inside points and reorder the rest of the vertices as follows. $i, i+1, x_1, j+3, \ldots, j, x_2, i+4, \ldots, i-1$

6.3 Visibility Graph

The visibility graph method is the **optimal path finding techniques in 2d spaces**. To complete the visibility graph we need to find for each vertex all the other visible vertices.

Algorithm 5 VISIBLE VERTICES(p,Graph)

Sort the obstacle vertices according to the clockwise angle that the halfline from p to each vertex makes with the positive x-axis. In case of ties, vertices closer to p should come before vertices farther from p. Let $w_1, w_2, \ldots, w_n$ be the sorted list.
Let ρ be the half-line parallel to the positive x-axis starting at p. Find the obstacle edges that are properly intersected by ρ, and store them in a balanced search tree T in the order in which they are intersected by ρ.
$W \leftarrow \emptyset$
for $i \leftarrow 1$ to n **do**
  if VISIBLE(w_i) **then**
    Add w_i to W.
  end if
  Insert into T the obstacle edges incident to w_i that lie on the clockwise side of the half-line from p to w_i.
  Delete from T the obstacle edges incident to w_i that lie on the counter-clockwise side of the half-line from p to w_i.
end for
return W

Algorithm 6 VISIBLE(w_i)

```
if pw_i intersects the interior of the obstacle of which w_i is a vertex, locally at
w_i then
  return false
else if i = 1 or w_{i-1} is not on the segment pw_i then
  Search in T for the edge e in the leftmost leaf.
  if e exists and pw_i intersects e then
    return false
  else
    return true
  end if
else
  if w_{i-1} is not visible then
    return false
  else
    Search in T for an edge e that intersects w_{i-1}w_i.
    if e exists then
      return false
    else
      return true
    end if
  end if
end if
```

to check if pw_i intersects internally I take the midpoint of pw_i and then check if that point is inside with **polygon crossing** in $O(logn)$. The search in T takes $O(logn)$ since it's a balanced binary search tree.

The rest are geometric properties with $O(1)$. This is the algorithm given in [3] and is stated as $O(n^2logn)$

the algorithm Visible(w) is computed for all pair of points the total time is actually $O(n^2) * O(logn) = O(n^2logn)$

Figure 16: Visibility Graph

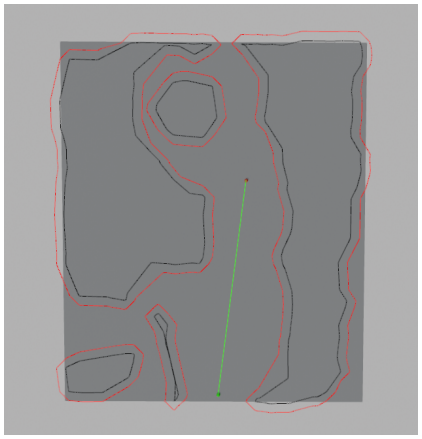
6.4 Path Planning

Now that we have the visibility graph it is very easy to find the shortest path between 2 points with Dijkstra's algorithm.

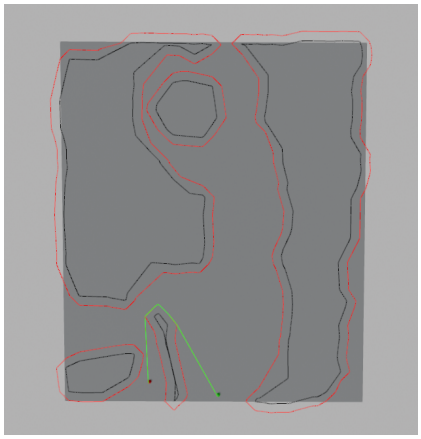
Algorithm 7 Dijkstra's Algorithm

```
1:  function Dijkstra(Graph, source)
2:  for each vertex v in Graph.Vertices do
3:      dist[v] ← ∞
4:      prev[v] ← UNDEFINED
5:      add v to Q
6:  end for
7:  dist[source] ← 0
8:  while Q is not empty do
9:      u ← vertex in Q with minimum dist[u]
10:     remove u from Q
11:     for each neighbor v of u still in Q do
12:         if neighbor exceeds the bound of the room break
13:         alt ← dist[u] + Graph.Edges(u, v)
14:         if alt < dist[v] then
15:             dist[v] ← alt
16:             prev[v] ← u
17:         end if
18:     end for
19: end while
20: return dist[], prev[]
```

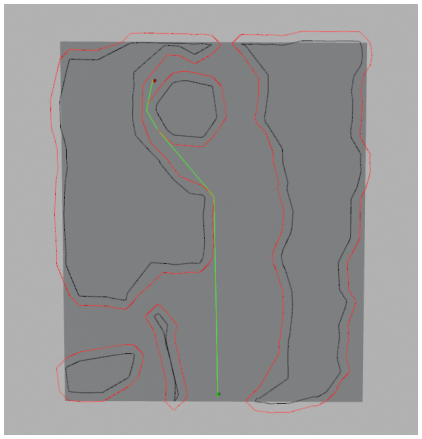
To find the path we just traverse the *prev* list in the reverse order. The total time complexity of Dijkstra is $O(ElogV)$ where E are the edges and V the vertices.



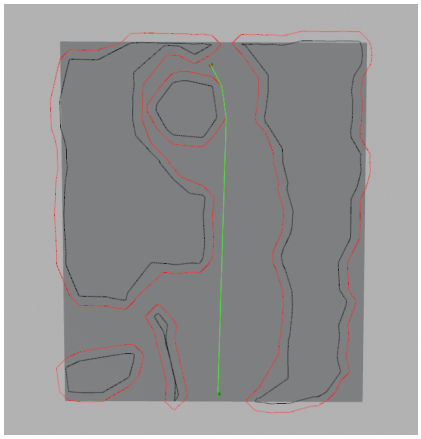
(a) Path example



(b) Path example with bounds in action



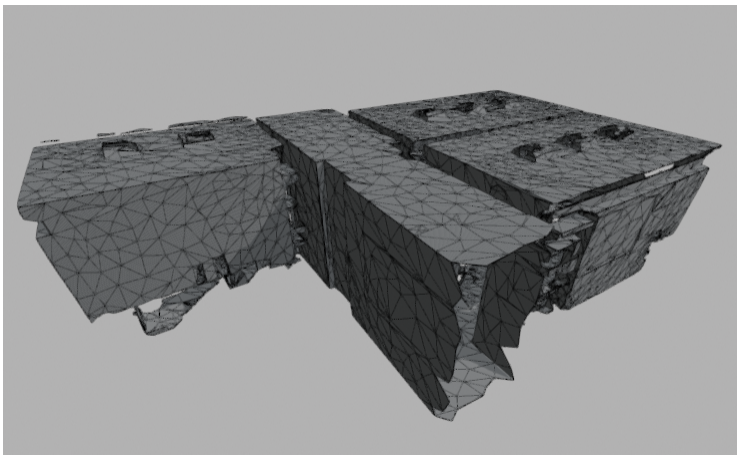
(a) Path example



(b) Path example

6.5 Multiple Room Path Planning

When having multiple rooms calculating the complete visibility is very inefficient.



What we will do instead is:

1. Calculate the visibility graphs for each room.
2. Calculate the connection of rooms using the door coordinates.
3. BFS to find the shortest path.

To find potential room connections we check if 2 doors are close to each other and that establishes a connection.

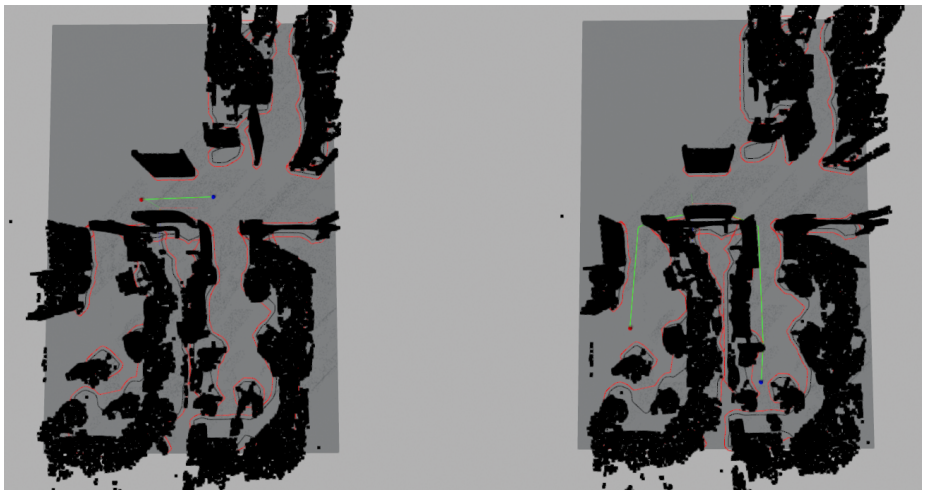
Now we can represent each room as a **node** and the **edges** of our graph represent the **connections**. It is an unweighted graph with edges where there is a connection between 2 nodes.

Given a point p we can find in which room it belongs:

$$min_x \leq p_x \leq max_x and min_y \leq p_y \leq max_y$$

We perform BFS to find the shortest path in an unweighted graph. We start with a queue that has only the start point. If the same room is the end room we find the shortest path. Else we traverse the neighbors that have **not been visited** and calculate the shortest paths to the current point(start point or some intermediate point) to the **average point of the 2 rooms' doors** and add the neighbors to the queue. When we reach the end room we add all the paths that brought us there and we have the shortest path.

This approach is much faster that calculating the visibility graph of all the rooms combined. However, now our solution is **suboptimal** but very close to optimal.



(a) Path example (b) Path example

7 Project Details

The project has dependencies on **vvrpywork library and open3d**.

The folders are organized as follows:

- src folder: contains all the source code
- roomX folders: contains a sub folder called **Planes** containing all the plane PCDs, a mesh.ply file that contains the room mesh and am objects.ply file that contains the objects.

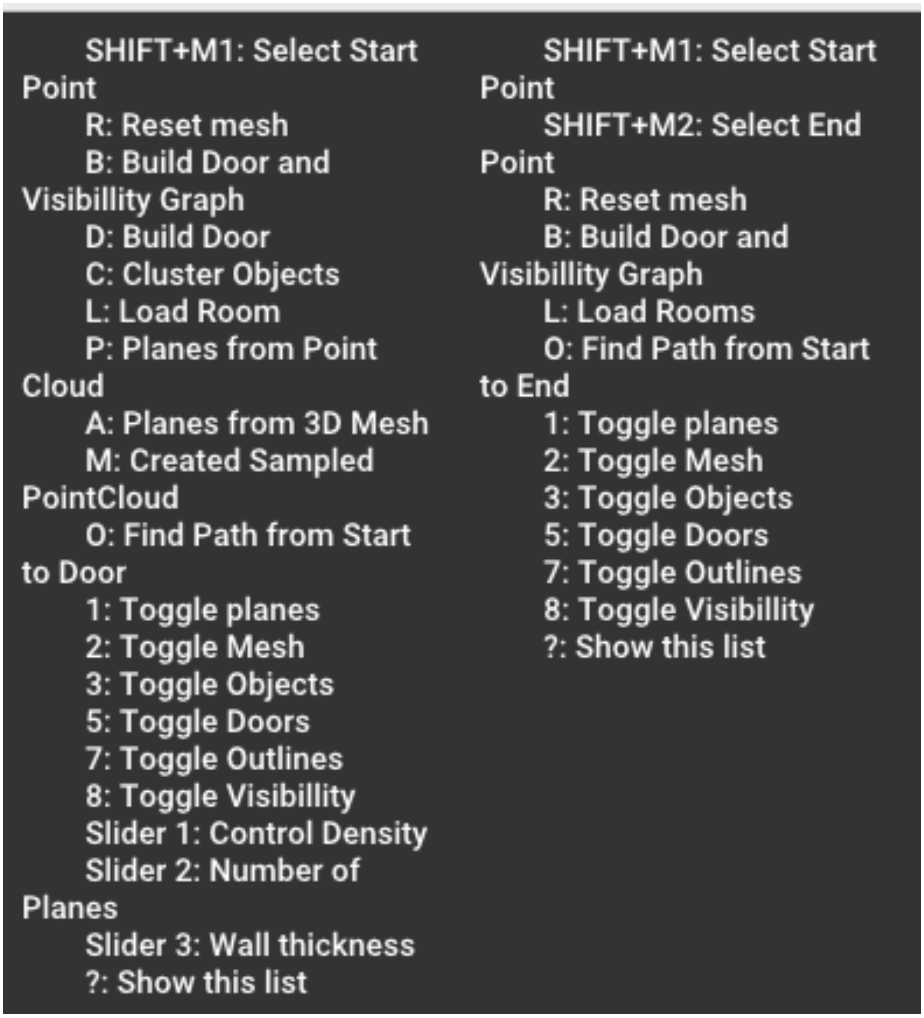
To run the single room example:

```
python singleRoom.py roomname
```

where roomname can be room1,room2,etc.
To run the multiple room example:

```
python multiRoom.py
```

Each application has its own clear instructions.



(a) Single room Instructions

(b) Multiple room Instructions

8 Bibliography

References

[1] Funkhouser, Thomas A., Peter Shilane, *Partial matching of 3D shapes with priority-driven search*, Proceedings of the tenth international conference on Artificial Intelligence and Statistics, 2002.

[2] Armeni, Iro, Ozan Sener, Amir R. Zamir, Helen Jiang, Ioannis Brilakis, Martin Fischer, Silvio Savarese, *3D Semantic Parsing of Large-Scale Indoor Spaces*, Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 1534–1543.

[3] de Berg, Mark, Marc van Kreveld, Mark Overmars, Otfried Schwarzkopf, *Computational Geometry: Algorithms and Applications*, 2nd ed., Springer, 2000.

[4] Edmond S. L. Cohen and Julien Tierny. Precise Surface Quasi-developability for Interactive Design. Technical Report LIRIS-3766, LIRIS CNRS, 2006. `https://liris.cnrs.fr/Documents/Liris-3766.pdf`.

[5] John Fisher. Alpha Shapes and Their Applications. Stanford University, 2011. `https://graphics.stanford.edu/courses/cs268-11-spring/handouts/AlphaShapes/as_fisher.pdf`.

[6] Wikipedia Dijkstra's Algorithm `https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm`