Abstract—This paper presents an approach that combines geometry processing with motion planning to enable a robot to efficiently navigate in unstructured environments. The proposed approach relies on a novel oversegmentation method to produce a decomposition of the free space into a set of connected regions. This provides a general and simplified planning layer with navigational routes along which sampling-based motion planning expands a tree of collision-free and dynamically-feasible motions to reach the goal. Experiments using robot models with nonlinear dynamics operating in complex environments show significant speedups over related work.

Index Terms—Motion and Path Planning; Nonholonomic Motion Planning

I. INTRODUCTION

AUTONOMOUS robotics provides a viable venue to increase productivity in transportation in the form of self-driving cars and package-delivery drones, search-and-rescue missions, and exploration, among others [1]. A fundamental ability of autonomous robots is their ability to navigate in complex, obstacle-rich environments. Even when decoupled from the perception problem, motion planning poses significant challenges [1]. The robot often needs to navigate in unstructured environments that contain numerous obstacles and narrow passages. Motions must also obey physical laws which restrict the velocity, acceleration, and turning radius. A successful planner needs to produce collision-free and dynamically-feasible motions that satisfy these constraints.

To address these challenges, the proposed approach leverages advances in two different areas: mesh segmentation from geometry processing and sampling-based motion planning from robotics. In geometry processing, decompositions have been used to partition complex models into convex components. Different approaches have been developed that remove nonconvex features [2], evaluate potential cuts [3], use integer linear programming [4] or Voronoi tessellations [5], [6].

In robotics, decompositions have often been used to facilitate motion planning. Among them, RRT [7] relies on an implicit Voronoi bias, PDST [8] creates a subdivision, KPIECE [9] uses multi-layered grids, and Syclop [10] and GUST [11] use discrete search over a workspace triangulation. Even though significant progress has been made, numerous challenges remain. Grid decompositions contain cells that intersect obstacles. Triangulations or other convex decompositions often have a large number of elements and narrow features, which make intra- and inter-region navigation difficult, especially when the robot dynamics are nonlinear.

To improve the state-of-the-art in sampling-based motion planning, we develop an intermediate representation layer whose goal is to bridge the gap between the raw geometric data describing the obstacles and the semantic information required to facilitate motion planning. The key is to decompose the environment into regions that can be easily navigated. To obtain such a decomposition, we leverage the idea of superfacets, which has significantly improved the speed and quality of decomposition algorithms by drastically reducing the number of regions from potentially millions to just a few hundred [12]–[15]. Superfacet decompositions are often obtained by normalized cuts [16]. This tends to produce good results, but has large runtime and memory requirements. To achieve log-linear time, the work in [17] leverages a bounded Dijkstra expansion, allowing it to scale so that datasets with millions of elements become tractable.

This paper makes several contributions. Drawing from advances in mesh segmentation, it introduces a superfacet decomposition based on a visibility criterion, tailored to facilitate inter- and intra-region navigability. Not only is the proposed superfacet decomposition efficient, but it is also simple to use. In fact, it requires as input only a triangulation of the workspace, freeing the user from the challenging task of determining a priori the number of superfacets or limits on their area. This is made possible by using a visibility flooding
approach to determine an initial partition and then iteratively improving the decomposition by adjusting the superfacet centers, reassigning triangles, and removing superfacet splits.

We use the superfacet decomposition to provide a general and simplified planning layer with navigational routes along which sampling-based motion planning can expand a tree of collision-free and dynamically-feasible motions to reach the goal. We contribute procedures for selecting superfacets based on shortest paths, updating edge costs, and exploring alternative routes to reflect the progress made by the motion-tree expansion. When progress along one route becomes difficult due to constraints imposed by the obstacles and the dynamics, the motion planner feeds back information so that the decomposition can be searched for an alternative route. This interplay enables the approach to rapidly reach the goal.

Experiments are conducted using robot models with nonlinear dynamics operating in complex environments. Comparisons to state-of-the-art related work show speedups of one to two orders of magnitude. Significant speedups are also obtained when using the superfacet decomposition instead of using triangular or other convex decompositions alone.

Given that motion planning with dynamics is undecidable [18], the proposed approach, as other sampling-based planners, offers probabilistic completeness. Our work focused on computational efficiency rather than optimality. Nevertheless, by using shortest paths in the decomposition to guide the exploration, the approach tends to generate short solution trajectories, which can be further optimized by numerical techniques [19]. While some motion planners [20], [21], e.g., RRT*, under certain assumptions, offer probabilistic optimality, their runtime is significantly larger, especially for nonlinear systems due to the rewiring of the motion tree.

II. PROBLEM FORMULATION

The workspace is defined by its bounding box $W$, goal region $G$, and obstacles $O = \{O_1,\ldots,O_n\}$. The robot is represented by its shape, state space $S$, control space $U$, and motion equations of the form $\dot{s} = f(s,u)$. The function

$$s_{\text{new}} \leftarrow \text{SIMULATE}(s,u,f,dt), \quad (1)$$

computes the new state $s_{\text{new}}$ by applying the control $u$ to the state $s$ and using the fourth-order Runge-Kutta approach to numerically integrate $f$ for one time step $dt$.

Fig. 2 shows the car and snake models used in the experiments. For the car, the state $s = (x,y,\theta,\psi,v)$ defines the position $(x,y)$, orientation $\theta$, steering $\psi$, and velocity $v$. The control $u = (a,\psi)$ defines the acceleration $a$ and the steering turn rate $\omega$. The motion equations $f$ are defined as

$$\dot{x} = v \cos(\theta) \cos(\psi), \quad \dot{y} = v \sin(\theta) \cos(\psi), \quad (2)$$

$$\dot{\theta} = v \sin(\psi)/L, \quad \dot{\psi} = a, \quad \dot{\omega} = \omega, \quad (3)$$

where $L$ is the distance from the back to the front wheels.

The snake is modeled as a car pulling several trailers. The state $s = (x,y,\theta,\psi,v,\theta_1,\ldots,\theta_m)$ is augmented to define the orientation $\theta_i$ of each of the $m$ trailer links. The snake is controlled in the same way as the car. The motion equations are augmented to describe the motions for each trailer as

$$\dot{\theta}_i = \frac{v}{H} (\sin(\theta_{i-1}) - \sin(\theta_0)) \prod_{j=1}^{i-1} \cos(\theta_{j-1} - \theta_j), \quad (4)$$

where $\theta_0 = \theta$ and $H$ is the hitch distance between the links. When $H$ is small ($H = 0.01$ in the experiments), the model resembles a snake, as shown in Fig. 2.

A state $s \in S$ is valid when placing the robot according to the position and orientation specified by $s$ avoids collisions and keeps the robot inside $W$. We use PQP [22] to efficiently implement $\text{COLLISION}: S \rightarrow \{\text{true, false}\}$.

A trajectory $\zeta : \{1,\ldots,\ell\} \rightarrow S$ is defined by starting at a state $s \in S$ and applying a sequence of controls $\langle u_1,\ldots,u_{\ell-1}\rangle$, i.e., $\zeta(1) = s$ and $\forall i \in \{2,\ldots,\ell\}$:

$$\zeta(i) = \text{SIMULATE}(\zeta(i-1),u_{i-1},f,dt). \quad (5)$$

The motion-planning problem is defined as follows: Given

- the world in terms of its bounding box $W$, obstacles $O = \{O_1,\ldots,O_n\}$, and goal region $G$,
- a model of the robot in terms of its geometric shape, state space $S$, control space $U$, motion equations $f$, and
- an initial state $s_{\text{init}} \in S$ compute a sequence of controls $\langle u_1,\ldots,u_{\ell-1}\rangle$ such that the trajectory $\zeta : \{1,\ldots,\ell\} \rightarrow S$ obtained by starting at $s_{\text{init}}$ and applying the controls in succession, as defined in Eqn. 5, avoids collisions and reaches $G$, i.e., $\forall i \in \{1,\ldots,\ell\}$:

$$\text{COLLISION}(\zeta(i)) = \text{false} \quad \text{and} \quad \text{POS}(\zeta(\ell)) \in G, \quad (6)$$

where $\text{POS}(\zeta(\ell))$ denotes the position component of $\zeta(\ell)$.

III. METHOD

The approach has two main components: superfacet decomposition and sampling-based motion planning. The decomposition provides an abstract layer which guides sampling-based motion planning as it expands a tree of collision-free and dynamically-feasible motions toward $G$.

A. Superfacet Decomposition

The superfacet decomposition uses a triangulation of the free area $W_{\text{free}} = W \setminus (O_1 \cup \ldots \cup O_n)$ as the basis for creating superfacets. Fig. 3(a) shows an example. The triangulation, denoted by $\Delta$ and computed by Triangle [23], is represented as a graph with the triangles as vertices. An edge $(\Delta, \Delta')$, with cost defined as $\|\text{CENTROID}(\Delta) - \text{CENTROID}(\Delta')\|$, is added to the graph for each pair of triangles that share a side.

The superfacet decomposition is defined as a connected partition of $\Delta$, assigning each triangle to exactly one superfacet. The superfacet decomposition is based on a visibility criterion which seeks to facilitate navigation. As described next, the initialization is followed by several refinement steps, as shown schematically in Fig. 4. Pseudocode is shown in Alg. 1.
Adjust Centers Reassign Triangles Remove Superfacet Splits
Workspace Obstacles Triangulate Initialize Superfacets
(a) triangulation (b) initialize (c) update centers (d) iteration 1 (e) iteration 7 (f) iteration 21 (g) splits

Fig. 3. (a–f) Snapshots of the triangulation, superfacet initialization, and superfacet refinement. Superfacets are color coded based on their shortest-path distance to the goal in the decomposition graph (red: close, blue: far). Notice that new superfacets can be created by the refinement. (g) During the triangle reassignment, the visibility criterion can cause one superfacet to be split by another, e.g., area \( A \) is visible only from \( c_1 \), so it will be assigned to \( c_1 \).

Algorithm 1: Superfacet Decomposition \((W, O)\)

**Input:** \( W \): workspace bounding box; \( O \): set of obstacles  
**Output:** decomposition of the free workspace area into superfacets

1: \( A \leftarrow \text{TRIANGULATE}(W, O); \) unmarked \( \leftarrow \Lambda; \) superfacets \( \leftarrow \emptyset \)  
2: \( \text{FILLVISIBILITY}(\text{unmarked}, \text{superfacets}, \Lambda) \) \( \text{§ III-A1} \)  
3: for several iterations and no convergence do  
4: \( \text{ADJUSTCENTERS}(	ext{superfacets}, \Lambda) \) \( \text{§ III-A2a} \)  
5: \( \text{REASSIGN}(	ext{superfacets}, \Lambda) \) \( \text{§ III-A2b} \)  
6: \( \text{REMOVESUPERFACETSPLIT}(	ext{superfacets}, \Lambda) \) \( \text{§ III-A2c} \)  
7: return (superfacets, \( \Lambda \))

function \( \text{FILLVISIBILITY}(\text{unmarked}, \text{superfacets}, \Lambda) \) \( \text{§ III-A1} \)

1: while ISEMPTY(\( \text{unmarked} \)) = false do  
2: \( \Delta \leftarrow \text{select triangle at random from unmarked} \)  
3: \( \Gamma \leftarrow \text{new superfacet with center \( \Delta \); superfacets} \leftarrow \text{superfacets} \cup \{\Gamma\} \)  
4: \( \text{TRIANGLES}(\Gamma) \leftarrow \text{VISDIJKSTRA}(\text{\( \Gamma \)}, \text{\( \Delta \)}, \text{\( \Lambda \)}, \text{labels}) \)  
5: \( \text{unmarked} \leftarrow \text{unmarked} \setminus \text{TRIANGLES}(\Gamma) \)

function \( \text{ADJUSTCENTERS}(	ext{superfacets}, \Lambda) \) \( \text{§ III-A2a} \)

1: for \( \Gamma \in \text{superfacets} \) do  
2: \( c \leftarrow \text{CENTROID}(\Gamma) \sum_{\Delta \in \text{TRIANGLES}(\Gamma)} \text{CENTROID}(\Delta) \)  
3: \( \text{CENTER}(\Gamma) \leftarrow \text{argmin}_{\Delta \in \text{TRIANGLES}(\Gamma)} \|c - \text{CENTROID}(\Delta)\| \)

function \( \text{REASSIGN}(	ext{superfacets}, \Lambda) \) \( \text{§ III-A2b} \)

1: for \( \Delta \in \Lambda \) do  
2: \( d(\Delta) \leftarrow \infty; \) labels(\( \Delta \)) \( \leftarrow \text{null} \)  
3: for \( \Gamma \in \text{superfacets} \) do  
4: \( d(\text{CENTER}(\Gamma)) \leftarrow 0; \) VISDIJKSTRA(\( \Gamma \), \( d \), \( \Lambda \), labels)  
5: while \( \exists \Delta \) with labels(\( \Delta \)) \( = \text{null} \) do  
6: \( \text{superfacets} \leftarrow \text{superfacets} \cup \{\Gamma\}; \) VISDIJKSTRA(\( \Gamma \), \( d \), \( \Lambda \), labels)  
7: for \( \Gamma \in \text{superfacets} \) do  
8: \( \text{TRIANGLES}(\Gamma) \leftarrow \emptyset \)

function \( \text{REMOVESUPERFACETSPLIT}(	ext{superfacets}, \Lambda) \) \( \text{§ III-A2c} \)

1: \( \text{unmarked} \leftarrow \emptyset \)  
2: for \( \Gamma \in \text{superfacets} \) do  
3: \( \Omega \leftarrow \text{BFS}(	ext{CENTER}(\Gamma), \text{TRIANGLES}(\Gamma), \Lambda) \)  
4: \( \text{unmarked} \leftarrow \text{unmarked} \cup \{\text{TRIANGLES}(\Gamma), \Omega\}; \text{TRIANGLES}(\Gamma) \leftarrow \Omega \)  
5: FILLVISIBILITY(\( \text{unmarked} \), \( \text{superfacets} \), \( \Lambda \))

1) Initialization: Initially, all triangles are unmarked. A superfacet is created by selecting a triangle \( \Delta \) at random from the unmarked triangles and running breadth-first search (BFS) from \( \Delta \). During BFS, a neighboring triangle is added to the queue only if it is both unmarked and visible from \( \Delta \). Two triangles are considered visible from each other if the line segment joining their centers does not intersect an obstacle. The triangles reached by BFS are assigned to the same superfacet as \( \Delta \). When BFS ends, a new superfacet is created by selecting an unmarked triangle and the process is repeated. Initialization ends when all triangles have been assigned to a superfacet. Fig. 3(b) shows an example.

2) Refinement: Each refinement step seeks to improve the decomposition by (a) adjusting the superfacet centers, (b) reassigning triangles to the closest visible center according to the shortest-path distance in the triangulation, and (c) removing any superfacet splits that occurred during the triangle reassignment. The refinement step is repeated until convergence. Fig. 3 shows the evolution of the superfacet decomposition across multiple refinement iterations.

a) Adjusting Superfacet Centers: Let \( \text{TRIANGLES}(\Gamma) \) denote all triangles assigned to superfacet \( \Gamma \). Each refinement step starts by defining the center of each superfacet \( \Gamma \) as the triangle in \( \text{TRIANGLES}(\Gamma) \) that is closest to the centroid, i.e.,

\[
\text{CENTER}(\Gamma) = \arg\min_{\Delta \in \text{TRIANGLES}(\Gamma)} \|c - \text{CENTROID}(\Delta)\|,
\]

where \( c = \frac{1}{|\text{TRIANGLES}(\Gamma)|} \sum_{\Delta \in \text{TRIANGLES}(\Gamma)} \text{CENTROID}(\Delta) \).

b) Reassigning Triangles: This step assigns each triangle to the closest visible center according to the shortest-path distance in the triangulation \( \Lambda \). Let \( c_1, \ldots, c_n \) denote the current superfacet centers. A modified version of Dijkstra’s shortest-path algorithm, here referred to as VISDIJKSTRA, is run from each \( c_i \). The first modification is to use a global array of distances, denoted by \( d \), which is initialized by setting \( d(c_i) = 0 \) for each superfacet center and \( d(\Delta) = \infty \) for every other triangle \( \Delta \in \Lambda \). By making \( d \) global, \( d(\Delta) \) is updated after each run of VISDIJKSTRA. The second modification is that when it removes \( \Delta' \) from the priority queue, a neighbor \( \Delta'' \) of \( \Delta' \) is processed only if \( \Delta'' \) is visible from \( c_i \). If so, VISDIJKSTRA proceeds with the usual distance update. When an update is made, the superfacet label for \( \Delta'' \) is changed to \( i \) to indicate that \( c_i \) is now the closest visible center to \( \Delta'' \).

Note that the adjustment of superfacet centers could cause some triangles to not be visible from any superfacet center, even after running VISDIJKSTRA from each \( c_1, \ldots, c_n \). To ensure that each triangle is assigned to a superfacet with a center from which it is visible, a triangle \( \Delta \) is selected uniformly at random from the unmarked triangles, i.e., those having \( d(\Delta) = \infty \). The triangle \( \Delta \) defines the center of a new superfacet and VISDIJKSTRA is run from \( \Delta \). This is repeated until all triangles have been assigned to a superfacet.

c) Removing Superfacet Splits: Due to the visibility criterion, after the triangle reassignment, some superfacet \( \Gamma \) could be split by another, as shown in Fig. 3(g). For this reason, BFS is run from each superfacet center to determine the triangles in the superfacet that are reachable. During BFS for \( \Gamma \), a neighboring triangle is added to the queue only if it belongs to \( \Gamma \). Triangles that are not reached by BFS are unmarked.
After running BFS from each superfacet, a procedure similar to the decomposition initialization (Section III-A1) is followed to create new superfacets using the unmarked triangles.

3) Visibility Checks: To speed up the visibility checks, VisiLibity [24] is used to compute the visibility polygon from each center Δ, denoted by VisiPOLY(Δ). The visibility test for a triangle Δ′ is conducted by checking whether the center of Δ′ is inside VisiPOLY(Δ). The visibility polygons are maintained in a hashmap. If a center Δ is not found in the hashmap, VisiPOLY(Δ) is computed and added to the map.

B. Superfacet Decomposition as an Abstract Planning Layer

The superfacet decomposition provides an abstract layer to guide motion planning. Each superfacet Γ keeps track of the shortest path in the decomposition from Γ to the goal Γ, denoted by PATH(Γ). The cost of PATH(Γ) serves as a heuristic to estimate the difficulty of reaching Γ from Γ.

To compute these shortest paths, the superfacet decomposition is represented as a graph D = (V_D, E_D, C_D), where V_D, E_D, C_D, and v_Γ represent the superfacets, edges, and cost edges, and v_Γ the goal Γ. An edge (Γ, Γ′) ∈ E_D denotes that Γ and Γ′ are neighbors; i.e., ∃Δ ∈ TRIANGLES(Γ) and Δ′ ∈ TRIANGLES(Γ′) such that Δ and Δ′ share an edge. C_D(Γ, Γ′) is defined as the cost of the shortest path in the triangulation Δ from CENTER(Γ) to CENTER(Γ′). An edge (Γ, v_Γ) with cost ||CENTER(Γ) − CENTROID(Γ)|| is added to E_D for every Γ that intersects Γ, i.e., (U_{Δ∈TRIANGLES(Γ)}) ∩ Γ ≠ ∅. Using v_Γ as the source, a single call to Dijkstra’s algorithm is used to compute PATH(Γ) for every Γ ∈ V_D.

C. Decomposition-Guided Sampling-based Motion Planning

The motion tree is a directed graph T = (V_T, E_T). Each vertex v ∈ V_T is associated with a valid state, denoted by STATE(v). Each edge (v, v′) ∈ E_T is associated with a valid motion from STATE(v) to STATE(v′), obtained by applying a control u to STATE(v) and integrating f for one time step; i.e., STATE(v′) = SIMULATE(STATE(v), u, f, dt). Starting from s_init, T is expanded by adding new vertices and edges. A solution is found when a vertex v that has reached Γ is added to T. The solution corresponds to the trajectory ζ_T(v), which is obtained by concatenating the motions associated with the edges that connect the root of T to v.

Motion planning is driven by three components: (1) using COST(PATH(Γ)) as a heuristic to select a superfacet Γ from which to expand T, (2) expanding T from Γ along PATH(Γ), and (3) updating the edge costs in the superfacet decomposition to reflect the progress made in expanding T. These procedures are invoked iteratively until a solution is found or the limit on the runtime is reached. Fig. 5 provides a schematic illustration. Pseudocode is shown in Alg. 2.

1) Superfacet Selection based on Shortest-Path Costs: The superfacet selection promotes expansions from superfacets that are associated with short paths to Γ. It also takes into account the number of times a superfacet Γ has been previously selected for expansion, denoted by SEL(Γ). These are combined into a weight w(Γ) defined as

\[ w(Γ) = (h(Γ))^{α}β^{SEL(Γ)}, \]

(8)

\[ h(Γ) = 1 + (\epsilon - 1)\frac{\text{COST}(\text{PATH}(Γ))}{\max_{v′∈V_D}\text{COST}(\text{PATH}(Γ′))}, \]

(9)

where \( α ≥ 1, 0 < β < 1, ϵ > 0 \). Note that h(Γ) ∈ [ϵ, 1]. As COST(PATH(Γ)) approaches 0, h(Γ) approaches 1. To ensure that each superfacet will eventually be selected, ϵ rather than zero is used for h(Γ) with the maximum COST(PATH(Γ)). The parameter α tunes the strength of the shortest-path heuristic.

To counteract the greediness of h(Γ), w(Γ) is multiplied by β (0 < β < 1) after each selection of Γ. The weight reduction ensures that eventually some other superfacet will have the maximum weight and be selected for expansion. This is necessary to avoid becoming stuck when expansions from Γ are infeasible due to the geometric and differential constraints imposed by the obstacles and the dynamics.

Each Γ also keeps track of the motion-tree vertices that have reached one of the triangles associated with Γ; i.e.,

\[ \text{VERTICES}(Γ) = \{v : v ∈ V_T \} \] and

\[ \text{POS}(\text{STATE}(v)) ∈ \bigcup_{Δ∈\text{TRIANGLES}(Γ)} Δ \].

(10)

Among the superfacets that T has reached, the superfacet with the maximum weight is selected for expansion; i.e.,

\[ \text{SELECTSUPERFACET}(D) = \arg\max_{Γ ∈ V_D, |\text{VERTICES}(Γ)| > 0} w(Γ). \]

(11)
2) **Motion-Tree Expansion along Shortest Path:** After selecting a superfacet $\Gamma$, the objective is to expand $T$ from $\text{VERTICES}(\Gamma)$ along $\text{PATH}(\Gamma)$. The expansion is driven by procedures to select a vertex $v$, select a target $p$, and expand $T$ from $v$ toward $p$. Recall that $T$ reaches a triangle $\Delta$ when $\exists v \in V_T$ such that $\text{POS}(\text{STATE}(v)) \in \Delta$. Similarly, $T$ reaches $\Gamma$ when it reaches some triangle in $\text{TRIANGLES}(\Gamma)$. The notation $\text{PATH}_i(\Gamma)$ denotes the $i$-th element, where $\text{PATH}_1(\Gamma) = \Gamma$ and $\text{PATH}_k(\Gamma)$ with $k = |\text{PATH}(\Gamma)|$ corresponds to the goal $G$.

The vertex selection has two steps: (i) select $\Delta$ from $\text{TRIANGLES}(\Gamma)$ that $T$ has reached, and (ii) select $v$ from the vertices that have reached $\Delta$. This promotes expansions from different areas, making it easier to find alternative routes when expansion along the current route becomes difficult.

To promote expansions that follow $\text{PATH}(\Gamma)$, the target $p$ is selected by sampling a point at random inside the superfacet $\text{PATH}_2(\Gamma)$, referred to as $\Gamma'$ (this is the superfacet next to $\Gamma$ in $\text{PATH}(\Gamma)$). To achieve this, a triangle $\Delta'$ is selected first from $\text{TRIANGLES}(\Gamma')$ with probability proportional to its area. A point is then generated at random inside $\Delta'$.

After selecting the vertex $v$ and the target $p$, the planner seeks to add a trajectory that starts at $\text{STATE}(v)$ and gets near $p$ (within some distance $d_{\text{near}} > 0$). Such a trajectory is generated by applying controls and integrating the motion equations for several steps until a collision occurs or the new state $s_{\text{new}}$ obtained after each integration step is near $p$, i.e., $\|\text{POS}(s_{\text{new}}) - \text{POS}(p)\| \leq d_{\text{near}}$. A proportional-integral-derivative (PID) controller [25] is used to steer the expansion from $v$ toward $p$. For a car or a snake model, this controller selects inputs that turn the wheels toward $p$ and then moves forward.

3) **Updating Shortest Paths:** When the expansion from $\Gamma$ to $\Gamma'$ encounters a collision, $\text{COST}(\Gamma, \Gamma')$ is multiplied by $\gamma$ (set to 1.1 in the experiments). The shortest paths for each superfacet are also updated. The cost increase reduces the likelihood of the shortest paths using the edge $(\Gamma, \Gamma')$, and, thus, promotes the exploration along alternative routes.

### D. Runtime Analysis

The triangulation requires $O(|\Lambda| \log |\Lambda|)$ time, where $|\Lambda|$ denotes the number of triangles [23]. The initial stage of the superfacet decomposition relies on BFS, which runs in $O(|\Lambda|)$ time since each triangle has at most three neighbors.

Adjusting the superfacet centers during the decomposition refinement takes $O(|\Lambda|)$ time. Dijkstra’s algorithm requires $O(|\Lambda| \log |\Lambda|)$ time since there are at most three edges for each triangle. Removing the superfacet splits requires running BFS, which takes $O(|\Lambda|)$ time. Therefore, each iteration of the refinement stage takes $O(|\Lambda| \log |\Lambda|)$ time.

Visibility checks are done from each superfacet center. The time to compute the visibility polygon from a center is $O(n_{\text{centers}} |O| \log |O|)$, where $n_{\text{centers}}$ is the total number of vertices of the polygons representing the obstacles and $|O|$ is the number of obstacles [26]. Let $n_{\text{centers}}$ denote the total number of centers created during the initialization and refinement stages. Note that $n_{\text{centers}} \leq |\Lambda|$, and, in practice, $n_{\text{centers}} \ll |\Lambda|$. Putting it all together, the time for the superfacet decomposition when invoking the refinement stage $k_{\text{refine}}$ times is

$$O(k_{\text{refine}} |\Lambda| \log |\Lambda| + n_{\text{centers}} (n_O + |O| \log |O|)).$$

(12)

Each iteration in motion planning (Alg. 2) is dominated by $\text{EXPANDMOTIONTREE}$ and $\text{UPDATESHORTTESTPATHS}$. Let $n_i$ denote the number of vertices added to $T$ during the $i$-th iteration. This means that $\text{SIMULATE}$ and $\text{COLLISION}$ were called at most $n_i + 1$ times (since the last call could result in collision and no vertex would be added). Therefore, if $\text{EXPANDMOTIONTREE}$ is called $k_{\text{MP}}$ times, then $\text{SIMULATE}$ and $\text{COLLISION}$ are called at most $k_{\text{MP}} n_i + \ldots + n_{k_{\text{MP}}} = k_{\text{MP}} + |T|$. $\text{SIMULATE}$ uses Runge-Kutta, so its runtime is $O(|S|)$. $\text{COLLISION}$ uses sweep-and-prune algorithms [22], [27], so its run time is $O(n_P \log n_P)$, where $n_P$ is the total number of vertices in the set of polygons representing the obstacles and the robot. $\text{UPDATESHORTTESTPATHS}$ uses Dijkstra’s algorithm. Since $|V_D| \leq |\Lambda|$ and $|E_D| \leq 3|\Lambda|$, its runtime is $O(|\Lambda| \log |\Lambda|)$. Putting it all together, if the motion planner terminates after $k_{\text{MP}}$ iterations, then its runtime is

$$O(k_{\text{MP}} |\Lambda| \log |\Lambda| + (k_{\text{MP}} + |T|)(|S| + n_{\text{MP}} \log n_P)).$$

(13)

Placing a bound on the number of iterations while guaranteeing convergence is an open problem. The probabilistic completeness for our approach follows from the analysis, applicable to any workspace decomposition, found in [11].

### IV. Experiments and Results

Experimental validation is provided in simulation using the car and snake models (Section II) and four different scenes, as shown in Fig. 1 and 6. These provide challenging scenarios where the robot has to navigate among numerous obstacles and pass through several narrow passages.

The approach is compared to $\text{RRT}$ [7] and $\text{KPIECE}$ [9] – two popular motion planners. To show the impact of the superfacet decomposition, it is also compared to work that uses only the triangulation without superfacets [11]. The notation $\text{M[Superfacets]} / \text{M[NoSuperfacets]}$ denotes the approach when using / not using the superfacet decomposition. Experiments were also conducted using a convex decomposition instead of the superfacet decomposition. We refer to
this variant as $M_{[CD]}$. The convex decompositions, shown in Fig. 6(d), were obtained by GEOMPACK++ [28].

Experiments also evaluate the impact of the triangulation granularity, decomposition refinement, and visibility criteria.

A. Problem Instances

To measure the performance, 60 different problem instances are generated for each scene. A problem instance is obtained by placing the robot and the goal at random locations. In scenes 1, 3, and 4, the robot is placed near the bottom and the goal is placed near the top. In scene 2, the goal is placed inside the innermost ring while the robot is placed outside the rings. In this way, the robot has to wiggle its way through several narrow passages to reach the goal.

The performance of a method is measured by running it for each robot model and scene on each of the 60 problem instances. The mean runtime and solution length are computed after dropping the four best and worst runs to avoid the influence of outliers. The solution length measures the distance traveled by the robot. The experiments were run on an Intel Core i7 using Ubuntu 15.10 and g++-4.8.4.

B. Comparisons to Other Motion Planners

The results in Fig. 7 show that $M_{[Superfacets]}$ is faster by orders of magnitude when compared to the state-of-the-art
methods RRT, KPIECE, M[CD], and M[NoSuperfacets]. The nearest-neighbor heuristic used by RRT often leads the motion-tree expansion towards areas blocked by obstacles. KPIECE lacks direction towards the goal, which slows down the exploration. M[CD] generates regions with different shapes and sizes, making it difficult to balance the motion-tree expansion from one region to the next. As expected, the runtime of M[NoSuperfacets] increases as the triangulation becomes more finely grained since more time is needed for the graph search. It also becomes more difficult to follow the shortest path since the triangles along the path have small areas. At the other end, coarse-grained triangulations create many narrow triangles, which make it harder for M[NoSuperfacets] to follow the shortest path. By grouping the triangles into superfacets, M[Superfacets] considerably reduces the graph search time. Moreover, the visibility criteria makes it easier for M[Superfacets] to follow the shortest paths.

Fig. 7 also shows the results on solution length. Since M[Superfacets] and M[NoSuperfacets] use shortest paths in the decomposition to guide sampling-based motion planning, both tend to generate short solutions.

C. Impact of the Triangulation Granularity

Fig. 8 shows the results of M[Superfacets] when varying the granularity of the triangulation (by changing the parameter TriMaxArea that defines the maximum triangle area, expressed as a percentage over the workspace area), which forms the basis for the superfacet decomposition. As shown, M[Superfacets] works well with a wide range of triangulations. A coarse triangulation should not be used since it makes it difficult to create superfacets that can effectively guide the motion-tree expansion. At the other end, a fine-grained triangulation increases the time to create the superfacet decomposition but tends to lead to shorter solutions. For new problem instances, we recommend starting with M[Superfacets,TriMaxArea=1/64%].

D. Impact of the Decomposition Refinement

Fig. 9 shows the results when varying the maximum number of iterations for the decomposition refinement. The runtime of M[Superfacets] increases considerably when there is no refinement. This is expected since the initialization, as shown in Fig. 3, tends to create superfacets that extend on opposite sides of an obstacle, making the navigation more difficult. In
contrast, after the refinement, each superfacet tends to be only on one side of an obstacle, which considerably improves the navigation. Fig 10 shows that the decomposition refinement converges after a few iterations, which considerably reduces the preprocessing overhead.

E. Impact of the Visibility Criterion

The superfacet decomposition uses visibility checks when assigning triangles to superfacets. Fig. 11 shows the results when disabling the visibility checks. This version, denoted \( M[k,\text{NoVisCheck}] \), initializes the decomposition by selecting \( k \) centers at random. During the refinement, each triangle is assigned to the closest center according to the shortest-path distance in the decomposition without checking if the triangle is visible from the center.

Results in Fig. 11 show that \( M[k,\text{NoVisCheck}] \) is considerably faster than \( M[k,\text{VisCheck}] \). When visibility checks are disabled, it is likely that triangles in a superfacet will be on opposite sides of an obstacle, making the navigation more difficult. This is more prevalent when the number of centers is small. When the number of centers is large, it increases the runtime for the graph search, and, consequently, for the whole approach. In contrast, \( M[k,\text{VisCheck}] \) ensures that each triangle in a superfacet is visible from its center, which makes it less likely for the robot to collide with the obstacles.

V. DISCUSSION

We have presented an approach that combines a novel, visibility-guided superfacet decomposition with sampling-based motion planning in order to enable a robot to efficiently navigate complex environments. Experimental results demonstrate significant speedups with respect to the state of the art. This work opens up several avenues for research. One direction is to consider further superfacet refinement through merging or splitting. Another direction is to extend the superfacet algorithm to 3D decompositions by grouping volumetric elements, e.g., tetrahedra, representing navigable space based on 3D visibility. Our intuition is that superetras would reduce the geometric complexity while effectively guiding motion planning. As with other sampling-based approaches, one could consider improving the quality of the first solution, for example, by rewiring the branches in the motion tree similarly to RRT* [20]. In addition, shape descriptors could potentially be introduced to provide information about narrow passages, hard turns, or terrain difficulty, which the motion planner can take into account during the motion-tree expansion. The long-term goal is to use geometry processing techniques to analyze a complex environment and extract high-level features carrying semantic information that facilitates motion planning.

REFERENCES


