Worcester Polytechnic Institute
Digital WPI

Major Qualifying Projects (All Years) Major Qualifying Projects

April 2016

Automatic Contact Surface Detection

John William Pryor
Worcester Polytechnic Institute

Follow this and additional works at: https://digitalcommons.wpi.edu/mqp-all

Repository Citation
Pryor, J. W. (2016). *Automatic Contact Surface Detection*. Retrieved from https://digitalcommons.wpi.edu/mqp-all/1365

This Unrestricted is brought to you for free and open access by the Major Qualifying Projects at Digital WPI. It has been accepted for inclusion in Major Qualifying Projects (All Years) by an authorized administrator of Digital WPI. For more information, please contact digitalwpi@wpi.edu.

Automatic Contact Surface Detection

Will Pryor
Worcester Polytechnic Institute
Email: jwpryor@wpi.edu

***Abstract*—Motion planners for humanoid robots search the space of all possible contacts the robot can make with its environment. In order to define this space, it is necessary to define all available surfaces in the environment with which the robot may make contact. We introduce a method to automatically detect these surfaces in real time, enabling humanoid robot motion planners to operate in any environment without requiring costly manual pre-processing. The system maintains a set of known planar surfaces, expressed as two dimensional polygons, and detects new surfaces and new areas of existing surfaces as the robot moves through the environment. The only required inputs are data from commonly available sensors and a small set parameters which depend only on the characteristics of the robot and sensor. No advance knowledge of the environment is necessary. In real-world environments, Surface Detection detects an initial set of surfaces within 1.5 seconds, and the average time to detect a newly-visible surface is less than two seconds. The system is capable of detecting all planar surfaces in a real-world environment and providing them to a motion planner in real time.**

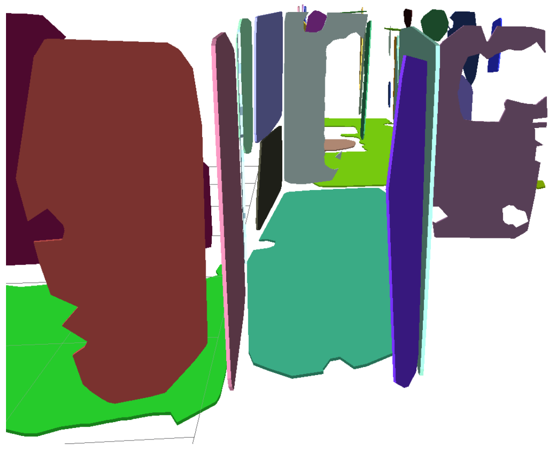
Fig. 1. The result of applying Surface Detection to a complex real-world environment. The viewpoint shows a hallway with adjacent rooms visible at the sides.

I. Introduction

Legged robots, including humanoid robots, move by making, maintaining, and breaking contact with the environment. Traditional planning methods for wheeled or fixed robots do not account for this need to make and break contacts, and actively avoid contact with the environment as a part of collision avoidance. Planners for legged robots use contact-before-motion planning, which plans a sequence of contacts before or simultaneous with planning for the motions which create those contacts.

Contact-before-motion planners operate in the contact space of the robot. Conceptually, a contact includes an area on the environment which the robot is making contact with and an area on the robot which is used to make contact. The former is referred to as an "environment contact surface" and the latter as a "robot contact surface". We refer to environment contact surfaces simply as "contact surfaces". Current contact-before-motion planners vary on how the surfaces are represented. In [1], [2], and [3], contact surfaces are represented with a 2D convex polygon and the pose of that polygon in the 3D environment. In [4], each environment contact surface is represented as a set of overlapping circles, each of which is represented as a 3D pose and a radius. We chose to represent each environment contact surface as a non-convex polygon and its pose in the 3D environment, which can be further processed into either of the above representations.

This report represents the work of WPI undergraduate students submitted to faculty as evidence of completion of a degree requirement. WPI routinely publishes these reports on its website with editorial or peer review. For more information about the projects program, please see http://www.wpi.edu/academics/ugradstudies/project-learning.htm

The environment contact surfaces, in whichever representation, may be defined by a human, as in [1] and [2], or extracted from another representation of the scene, as in [4]. Requiring human pre-processing of the environment severely limits the environments in which such planners can be run, and eliminates applications such as disaster recovery where the environment is completely unknown. Extracting the information from a point cloud, a common representation of 3D sensor data, allows these planners to be used without requiring human pre-processing of the environment. [4] and [5] use this approach, but both assume the extraction will run once, at the beginning of planning, with a point cloud that contains all the necessary information for the plan. In many applications, including disaster recovery, the only available sources of data are the sensors mounted on the robot itself. In order to work in these environments, it is necessary to process the environment incrementally, as it is seen by the sensor. Our process aims to perform surface detection in real time, which requires consideration of the sensor data over time.

The two most common types of 3D sensors used for robot perception are laser scanners and depth cameras. Both of these technologies can provide data in point cloud format. A point cloud contains a (typically unordered) list of three-dimensional points, each of which may also contain additional information such as color or intensity. Each point in the point cloud corresponds to a point on a physical surface in the environment. Depth cameras produce very dense point clouds which are typically organized, meaning the 2D pixel location of the point from the camera sensor is preserved and can be used to speed up certain algorithms, and have color information, but are limited to a low field of vision and often a low range. Laser scanners have high range and field of vision, but produce unorganized point clouds with many fewer points. We aimed to develop a surface detection process which works on any input sensor, so chose algorithms which do not rely on point clouds being dense or organized, having color information, or having a large field of view.

The Point Cloud Library (PCL) contains implementations of many standard algorithms used to process and extract information from point clouds [6]. In our process we make use of PCL data structures and algorithms for preprocessing and segmentation of the input point clouds.

II. Problem Statement

This project aims to develop a method to automatically detect all planar surfaces in the environment and provide them in a format suitable for a humanoid robot motion planner. The output should consist of a minimal set of 2D polygons, which need not be convex, where each planar surface is represented by one polygon. The process should be capable of taking input incrementally, as the robot explores the space, and should run in real time. Finally, the only required input should be the data available from sensors that are commonly found on humanoid robots and a small set of input parameters which should be environment-independent.

We use four parameters to control the behavior of surface detection. The values of these parameters depend only on the characteristics of the robot and the sensor, and not on the characteristics of the environment. We refer to these parameters as h, d, r, and n.

The first parameter, h, controls the maximum size of hole or concavity that will be included as a part of the surface. It is expressed as the maximum distance between two points on the boundary of the hole. Because the point cloud input is discontinuous, it is necessary to specify a threshold which allows us to distinguish between discontinuities caused by the representation and discontinuities that accurately represent the underlying surface. The value of h is bounded below by the expected resolution of the point cloud sensor. If it is lower than the expected resolution, discontinuities caused by the sensor may be interpreted as holes which should be represented in the detected surface. h can be bounded above by the maximum size of hole that the robot's contact surfaces can cover without affecting the stability of the contact, and is therefore determined by the robot geometry.

The second parameter, d, controls the distance at which two points can be considered identical. This value is used as the discretization resolution of the point cloud, and so higher values will lead to better performance as the preprocessing steps can discard more redundant points. d can be specified by choosing the largest value which will not cause inaccuracies in the representation by moving points across the threshold defined by h. The maximum distance the discretization process can create between two points is defined by the farthest distance between two adjacent cells in an 8-connected 3D grid. Therefore, $d \leq \frac{h}{2\sqrt{3}}$.

The third parameter, r, controls the neighborhood of points which must be considered to acquire an accurate estimate of local surface geometry. This value is determined by the properties of the sensor. r should chosen set such that the probability

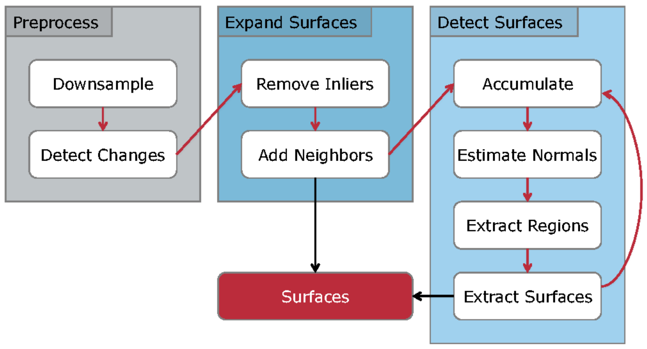

Fig. 2. The three phases of surface detection and the components of each phase. The flow of points between components is depicted with red arrows, and the flow of surfaces is depicted with black arrows.

of significant errors in the estimation of local geometry based on neighbors within radius r is low. This calculation depends on the expected density of points, determined by the sensor's properties and by d, and on the properties of the noise in the sensor readings.

Finally, n controls the minimum number of points which a surface must contain in order to be considered valid. This parameter should be set to either the minimum number of points required to derive an accurate model or the number of points corresponding to a desired minimum surface area, whichever is higher. The number of points required to derive an accurate model is determined by the noise characteristics of the sensor. The number of points required to guarantee a minimum surface area is determined by the expected density of points, which determined by the sensor's properties and by d.

III. Surface Detection

Surface detection is implemented in three phases: a preprocessing phase, which reduces the number of points to be considered without discarding information; an expansion phase, during which the incoming point cloud is searched for points which correspond to the sensor seeing additional area from an already-detected surface; and a detection phase, which detects new surfaces in the environment.

A. *Preprocessing*

Preprocessing of the input serves two purposes. First, it allows the system to discard many of the input points without losing information. In testing, preprocessing was able to reduce the number of input points by factors of 3 to 5. Because the performance of many of the algorithms used in surface detection scale with the number of input points, this reduction is important to allow the system to run in real time. Second, it compensates for the difference in input density between depth camera sensors and laser scanner sensors, allowing the system to be less sensitive to the technology used. Preprocessing involves two steps: downsampling and change detection.

1) Downsampling: We use PCL's VoxelGrid to limit the density of the incoming point cloud. A voxel grid discretization groups the input points into cells in a three-dimensional grid, then outputs one point for each occupied grid cell whose coordinates are the centroid of every point in that cell. This effectively combines nearly-identical points into a single representative point while preserving location information. The discretization resolution is d, which is chosen to be small enough that discretization will not introduce holes.

2) Change Detection: Downsampling removes redundancy in space, but since this is an incremental algorithm we must also consider redundancy in time. The entire Surface Detection process is designed to never discard information, so once a point enters the system it is never lost. Because of this, adding the same point a second time has no effect other than increasing processing time. Change Detection uses PCL's OctreePointCloudChangeDetector to remove points which are nearly identical to points which have been perceived before. It uses the same voxel grid discretization as the downsampling step to determine which points are nearly identical. After the first few seconds of input, Change Detection often discards 80% of the input points, greatly reducing the processing time of the algorithms that follow.

B. Surface Expansion

The second component of the process is Surface Expansion, which searches incoming point clouds for points which are members of existing surfaces. This is occurs before Surface Detection because the sensor will often detect a sufficiently large additional area within an existing surface for Surface Detection to report it as a new plane. This would result in two different planes representing a single real-world contact surface, which is a less accurate representation of the environment. Surface Expansion also runs faster than Surface Detection, so it is beneficial to allow Surface Expansion to run on every point in a preprocessed input cloud and then run New Surface Detection on the remaining points. Surface Expansion involves two steps: adding inliers and expanding surfaces.

Algorithm 1: Expand Surfaces

```
input : Existing surfaces $surfaces$, ordered by size
        Point cloud of new points $cloud$
output: Potentially expanded surfaces $surfaces$
        Uncategorized points $cloud$
foreach $surface \in surfaces$ do
    $inliers \leftarrow surfaceInliers(surface, cloud)$
    $surface \leftarrow surface \cup inliers$
    $cloud \leftarrow cloud \setminus inliers$
foreach $surface \in surfaces$ do
    $boundary \leftarrow$ boundary of $surface$
    while $boundary$ has neighbors in $cloud$ do
        $neighbors \leftarrow$
          $pointNeighbors(boundary, cloud)$
        $expansion \leftarrow$
          $planeNeighbors(surface, neighbors)$
        $boundary \leftarrow boundary \cup expansion$
        $cloud \leftarrow cloud \setminus expansion$
    $surface \leftarrow surface \cup boundary$
```

1) Add Inliers: Often a sensor will perceive a surface more than once, and subsequent scans can provide more detail on the surface. Add Inliers identifies points which are contained in an existing surface, referred to as "inliers", and adds them to the list of inliers belonging to that surface. The updated list of inliers can then be used to refine the estimate of the polygon's pose within the environment, although that is not currently implemented. Inliers are found by iterating over the current list of known surfaces, beginning with the surface which contains the most points. The number of existing inliers serves as a very simple heuristic to estimate the probability of finding new inliers. Each newly-discovered inlier reduces the number of points that have to be checked for the remaining surfaces, so iterating in order of probability of discovering new inliers offers a speed benefit.

On each iteration, the point cloud is first filtered to only those points within a given distance s to the surface's plane, then the remaining points are filtered to only those points within the surface's polygon using PCL's CropHull. The distance s represents the maximum distance between the location of the real-world surface and the location of the detected point. This distance is determined by the sum of the expected sensor noise and the maximum error caused by downsampling, which is $d\sqrt{3}$. Points which pass both these filters are the inliers, and are added to the surface's list of inliers and removed from the input point cloud. After the last iteration, all points remaining from the input point cloud are passed on to surface expansion. When no surface have been detected yet, Add Inliers passes its input along unmodified.

2) Expand Surfaces: This step handles the common scenario of detecting additional area within an already-detected surface. Similarly to III-B1, the input is processed by iterating over surfaces with the largest surfaces first to maximize the probability of removing points from consideration quickly. Each surface maintains a list of boundary points, which are points along the boundary of the surface polygon and are never more than h distance away. On each iteration, the we attempt to grow the surface outward by finding all points within h distance of any boundary point, and filtering the results to only those points within s distance of the surface's plane. This two-step search allows Expand Surfaces to search a larger distance in the directions parallel to the plane than in the direction perpendicular to the plane. If any points are found, those points are added to the list of boundary points and the process is repeated. This allows large areas of the input cloud to be added to their surface quickly. Once no new points can be found, the newly discovered boundary points are added as inliers to the surface and its polygon and list of boundary points are recomputed. Points which are not used to expand any surface are passed on to New Surface Detection. When no surface have been detected yet, Expand Surfaces passes its input along unmodified.

C. New Surface Detection

The other important step in the process is the New Surface Detection step. This step is responsible for the initial identification of surfaces within the point cloud. This is by far the most time-consuming step, which is why every step that could potentially remove points from consideration is placed before it. New Surface Detection consists of four parts: point accumulation, surface normal estimation, region segmentation, and plane segmentation.

Algorithm 2: Detect Surfaces

```
input : New points
output: New surfaces
        Uncategorized points
$surfaces \leftarrow \emptyset$
$cloud \leftarrow$ new points
$normals \leftarrow estimateNormals(cloud)$
$regions \leftarrow segmentRegions(cloud, normals)$
foreach $region \in regions$ do
    while $|region| > n$ do
        $model, inliers \leftarrow fitModel(region)$
        $region \leftarrow region \setminus inliers$
        $segments \leftarrow segmentDistance(inliers)$
        foreach $segment \in segments$ do
            if $|segment| > n$ then
                add $(model, segment)$ to $surfaces$
```

1) Accumulate Points: The first step in New Surface Detection exists to account for a scenario which occurs fairly commonly with a laser scanner, but also may occur when using a depth camera. Rotating laser scanners, if sampled quickly enough, will often see a surface one thin strip at a time. Because strips are close together and points on each strip are nearly collinear, points from only one or two strips often have too low of an area for their surface to be detected. If each strip of a surface occurs in a different scan, and points which are not detected as planes are discarded, it is possible for a surface to never be detected. To solve that problem, points for which a surface is not detected using New Surface Detection return to the accumulator and are combined with input from the Surface Expansion step before being output to New Surface Detection again.

Every time Accumulate Points receives a new set of points from Surface Expansion, it applies a radius outlier filter to all accumulated points. The filter, which is implemented using PCL's RadiusOutlierRemoval, removes every point which does not have at least two neighbors within a distance of r. The number two is significant, because the step that follows Accumulate Points fits a plane to every point and its neighbors, and the minimum number of points needed to fit a plane is three. If, after filtering out the outliers, there are more than n points remaining, the filtered point cloud is sent to the next step. The plane's outliers then form the first points of the next accumulated cloud, because discarding them would violate the principle that information does not leave the system.

2) Surface Normal Estimation: This step estimates the surface normals of each point. Although a point cloud itself is not a surface, so cannot have a surface normal, the points in a point cloud represent a real-world surface. Surface Normal Estimation is the process of estimating the surface normal of the real-world surface at every point in the point cloud. This is a common operation on point clouds, and the Moving Least Squares algorithm [7] is one of the standard surface normal estimation methods.

Moving Least Squares (MLS) considers every point in a point cloud separately. It finds each point's neighbors within a given radius, then performs a least-squares fit of a 3D polynomial equation to those neighbors, where each neighbor is weighted by its distance to the point of interest. The normal vector to the equation at the projection of the point of interest onto the equation's surface is then an estimation of the real-world surface normal. The radius, polynomial degree, and weighting factor are defined by the user. MLS is also capable of using its estimated equation to resample the surrounding points, using one of a number of strategies, but we do not make use of this ability.

For our application, we use a first-degree polynomial, i.e. a plane. We set the search radius to r, and the weighting factor to the square of the search radius (that is, r^2) as recommended by PCL. PCL's implementation of MLS also stores the final error of the least-squares fit as the "curvature" of the point. This value is correlated to how far the point's neighbors deviated from the plane, which can be used to identify points that lie on the boundary between two surfaces.

3) Region Segmentation: Once the surface normal information is available, it is used to segment the input point cloud into a set of smoothly connected regions. Smoothly connected regions are groups of neighboring points where the difference between the surface normals of adjacent points is low, and none of the points has high curvature. This effectively separates points which are neighbors, but belong to different surfaces, by detecting corners and edges. Note that region segmentation does not guarantee that every region will be planar. Any surface with a gradual curve will be segmented as a single region, including cases where multiple planar surfaces are connected by a curve. This means that an extra step is required to separate planar surfaces from curves, but it has the advantage of allowing easy extensibility. Because this step does not separate curved surfaces, it would also apply to an expanded version of Surface Detection which considers 3D primitives other than planes. Cylinders, which are common in man-made environments as pillars, poles, and railings, are also extracted as smoothly connected regions, and to add support for detecting cylinders would not require significant modification of this step or any prior step.

In order to detect smoothly connected regions, we use Region Growing Segmentation [8]. This algorithm segments the environment by repeatedly picking a seed point for a new cluster from the set of uncategorized points and growing that cluster to include every point which is smoothly connected to the seed. It does this by finding the neighbors within a given radius, comparing their surface normals and curvatures, adding those with similar surface normals and low curvatures to the region, and repeating the search with their neighbors. The radius, surface normal difference threshold, and curvature threshold are defined by the user.

For this application, the radius is r and we use $30°$ for the surface normal threshold and 0.05 for the curvature threshold. PCL's implementation, RegionGrowing, also accepts a minimum size parameter, which determines the minimum number of points required in a region, which we set to n.

4) Surface Extraction: Once the environment has been segmented into smoothly connected regions, it undergoes one final segmentation to extract planar surfaces. This step also estimates the plane equation of the planar surface, which is necessary in order to define the surface polygon. To extract a plane equation (also called a plane model) and set of inliers from the region, we use the Random Sample Consensus (RANSAC) model fitting algorithm, implemented in PCL as SACSegmentation.

RANSAC robustly fits a model to a set of observations in the presence of outliers [9]. It achieves this by randomly choosing the lowest number of points necessary to fit a model (three, in the case of a plane) and building a candidate model. The rest of the input points are checked for membership in the model. If the model contains more inliers than the previous best model, then it is saved as the best model. After a number of iterations, the best model is returned. Though RANSAC is not designed for cases in which more than one instance of the model appears in the data, its robustness to outliers means it works well in that scenario. The model, maximum distance to the model, and number of iterations are defined by the user.

For this application, we currently use a simple plane model; however, PCL provides an implementation of a normal-aligned plane, which requires that points must have normals within a given threshold of the plane's normal in order to be considered inliers. As we have surface normal information available, and aligning the estimated plane's normal with the surface normal is desirable in order to preserve the surface normal orientation, it may be valuable to use this model instead. For the maximum distance to the model we use s, as in III-B1, and we specify 200 iterations.

An additional advantage of RANSAC is that it is capable of estimating any model. This allows for the possibility to extend this algorithm to estimate surfaces along other 3D primitives, such as cylinders, as discussed in III-C3.

One final segmentation step is necessary because RANSAC does not consider distance between inliers. It is possible to have a surface which consists of two coplanar regions connected by a smooth, but not planar, curve. In this case, RANSAC will fit a single model to both those planes. In order to correctly represent them as two separate surfaces, the inliers reported from RANSAC are passed through Euclidean Distance clustering to separate the clusters corresponding to the surfaces. h is used as the distance threshold. Finally, all planar surfaces with larger than n points are output.

IV. Surface Representation

The desired output format for the list of surfaces consists of two pieces of information: the plane equation and surface polygon. However, we also require every surface to have a list of inliers and boundary points in order to perform surface expansion, and we store other information to benefit users of the system. In total, we store seven pieces of information for each surface: an identifier, a color, a set of inliers, a plane equation, a set of boundary points, a polygon, and an extruded triangle mesh. The identifier and color are both assigned to a surface to allow programs and humans to identify them. The set of inliers and plane equation define a surface, and the remaining items are derived from a surface.

The three derived items are dependent on the set of inliers, and therefore need to be computed either when a surface is first discovered by New Surface Detection or when it is expanded by Surface Expansion. Note that although each step in Surface Expansion adds to the set of inliers, adding inliers to the interior of the surface will not change its boundary points, so the surface does not need to be re-computed in between Add Inliers and Expand Surfaces. After New Surface Detection and Surface Expansion, there may be surface which need to have their derived data re-computed, which uses the following algorithms.

A. Boundary Points and Polygon

Algorithm 3: Compute Derived Data for Surface

```
input : Inliers of a single surface $inliers$
        Plane model of the surface $model$
output: Set of boundary points $boundary$
        Surface polygon $polygons$
        Surface triangle mesh $mesh$
$triangulation \leftarrow delaunayTriangulation(inliers)$
foreach $triangle$ of $triangulation$ do
    if any side of $triangle$ is longer than $\alpha$ then
        remove $triangle$ from $triangulation$
$adjacency \leftarrow \emptyset$
foreach $edge$ of $triangulation$ do
    if $edge$ belongs to exactly one triangle then
        add $edge$ to $adjacency$
$boundary \leftarrow$ all vertices of $adjacency$
$polygons \leftarrow \emptyset$
while $adjacency$ is not empty do
    $vertex \leftarrow$ lowest point of $adjacency$
    $polygon \leftarrow \{vertex\}$
    while $vertex$ has at least one adjacent point do
        $vertex \leftarrow$ point adjacent to $vertex$
        add $vertex$ to $polygon$
    add $polygon$ to $polygons$
$mesh \leftarrow polygonTriangulation(polygons)$
$projection \leftarrow$ direction opposite $model$'s normal vector
$mesh \leftarrow mesh \cup projectVertices(mesh, projection)$
$mesh \leftarrow mesh \cup$ triangles connecting the two surfaces
```

As part of our surface output, we require a polygon which represents the surface. Specifically, we require every point inside the surface also be inside the polygon, and that areas of a certain size which do not contain any points must not be inside the polygon. These requirements correspond with the definition of the alpha shape, a generalization of the convex hull which adds the ability to represent surfaces with holes and concavities of a specified size [10]. The alpha shape of a set of points can be computed by finding the points' closest-point Delaunay triangulation and removing all triangles which have at least one side of length greater than the α parameter. α represents the size of the smallest hole or concavity to include in the output representation, which corresponds to our h parameter.

For our implementation of the alpha shape, we first transform and project the inliers into the surface's plane, which reduces the dimensionality to 2D. We then use Triangle to compute the Delaunay triangulation of the inliers [11]. We then remove every triangle with a side length greater than h, then build an adjacency list over the vertices of the triangulation, where two vertices are adjacent if there exists an edge between them which belongs to exactly one triangle (i.e. an edge on the boundary of the shape). Finally, we build the polygon representation by traversing each connected component of the adjacency graph counterclockwise and adding the vertices of each connected component to an ordered list. Each vertex in this list is represented as an index into a point cloud which is stored separately. This yields a representation where each polygon with holes is represented by one or more polygons without holes. The first in the list is the outer perimeter. Subsequent polygons in the list may represent holes, or they may represent islands within holes. Any point which is inside an odd number of these simple polygons is inside the surface, otherwise it is outside.

Because each segment in the polygon is guaranteed, by the properties of the alpha shape, to have a length of at most h, the vertices of the alpha shape already represent the list of boundary points which is required for the Expand Surfaces step. Because we store the set of vertices of the alpha shape separately from the polygon, no further computation is necessary to derive the list of boundary points. This also allows for a possible optimization. Planar surfaces in human-designed environments, such as tables, walls, and floors, are often bounded by long, straight lines. However, the restrictions of the alpha shape mean that these edges will be represented by a series of near-collinear line segments. Applying polyline simplification to combine nearly-collinear boundary segments has the potential to increase the speed of algorithms, such as point-in-polygon tests, which scale with the number of segments in a polygon. Because the boundary list is stored separately, it is possible to make that optimization without harming Surface Expansion.

B. Triangle Mesh

The final piece of derived data is a 3D triangle mesh, used to aid in visualization and in collision checking. This mesh represents 3D volume created by extruding the polygon a short distance in the direction opposite its surface normal. It can be visualized along with other 3D data in the same environment, such as the input point cloud or the robot model, to allow observation of the planning process. It can also be used by the motion planner as part of a collision check, although because it only represents the planar surfaces in the environment, and not other objects, it is not suitable to be the only source of collision information.

To compute the mesh, we use Triangle's ability to triangulate a Planar Straight Line Graph (a superset of the format which we use to describe the surface polygon) to generate a triangulation of the 2D surface [11]. We then project a duplicate of the resulting triangulation a small distance in the direction opposite the surface normal, reversing the order of the vertices of each triangle to conform with the convention that indices should be ordered counterclockwise from the perspective of an observer located outside the 3D volume. Finally, to join the two faces, we add one triangle for each pair of corresponding vertices in the original and projected triangulation, alternating the position of the third point in the triangle in order to create a closed volume.

TABLE I. SURFACE DETECTION PERFORMANCE

Environment	d	Points per Second	Time to First Detection (s)	Detection Lag (s)	% of Points Categorized
Simulated Simple	0.02	4900	1.02	0.305	96.0%
	0.01	8000	0.221	0.468	93.0%
Simulated Complex	0.02	4100	1.01	0.151	90.4%
	0.01	6900	0.512	0.399	91.3%
Real-World Complex	0.02	12000	1.41	1.59	78.2%
	0.01	19000	1.94	3.04	72.3%

V. RESULTS

The algorithm was tested on real-world point cloud data and in simulation in a variety of complex environments with simulated sensor noise. Performance was measured on an 8-core 3.4 GHz Intel Core i7-3770 CPU. Tests were run using output from a real or simulated laser scanner, and data from the sensor was processed in increments of one second. The output of each set of tests is the average over five runs. In each environment, we ran one set of tests with a discretization resolution of 2cm ($d = 0.02$) and another set with a resolution of 1cm ($d = 0.01$) in order to compare performance under different data rates. In the simulated test scenarios we used a maximum hole size of 8cm ($h = 0.08$), a search radius of 15cm ($r = 0.15$), and a minimum surface size of 200 points ($n = 50$). The real-world data were collected using a different sensor, which required different parameters. In the real-world test scenarios we used a maximum hole size of 8cm ($h = 0.08$), a search radius of 15cm ($r = 0.15$), and a minimum surface size of 500 points ($n = 50$).

For each test set, we measure the average number of points per second, the time taken to detect the first set of surfaces, the average detection lag, and the average percentage of points which are categorized into a surface. The time to first detection is measured from when the first input is received to when every surface detected from the first input cloud is available to the planner. The detection lag is the average of the delays between when a point is received to when the surface which contains that point is available. Points which are not categorized, or which are categorized only after receiving additional scans, are not included. Finally, the percentage of points categorized indicates the average percentage of all received points which have been accumulated in the Accumulate Points step (see III-C1). This indicates with the percentage of the scene which has been detected as a planar surface.

In all cases, the first set of surfaces is available to begin planning within two seconds. Subsequent surfaces are identified within three seconds of becoming visible. In all environments, halving the discretization resolution causes an approximately $1.6\times$ increase in the number of points received per second. The detection lag reflects the decrease in speed this creates. This is present in the simulated simple environment, which is much less prone to requiring multiple scans to detect a surface because it lacks small surfaces, but has a larger effect on the complex environments where an increased number of points can make the difference between detecting a surface after one scan or two. The time to first detection also reflects the decrease in performance resulting from a lower discretization resolution in the real-world complex environment. In the simulated environments, where the value of n is significantly lower, the increased number of points allows the first surfaces to be detected after only one scan, while the real-world environment still requires two scans when using both of the tested resolutions.

The percentage of points categorized shows that, in the real-world environment, Surface Detection categorizes 70% to 80% of the environment. In simulated environments, which lack interference from small, non-planar objects which exist in the real-world environment, the percentage categorized is higher. In the real-world environment the percentage of the environment which is categorized into a plane is lower with a lower discretization. Because most of the non-planar objects' surfaces are irregular, the discretization resolution has a greater effect on the number of points on those surfaces, increasing the percentage of points which lie outside any planar surface.

VI. Conclusion

We have presented a process for automatically detecting all planar surfaces in an environment in real-time using only the sensors that are commonly found on humanoid robots. These surfaces can be used in a humanoid robot motion planner, allowing it to operate in completely unknown environments. The process runs continuously as the sensor moves throughout the environment, and can report a new surface within seconds of the surface becoming visible to the sensor. To achieve this we use a series of segmentation algorithms to detect surfaces in the environment and a surface expansion process which efficiently categorizes points which represent new areas of existing surfaces. The process is designed to allow extension into other 3D primitives, in addition to planes. It is also designed to be agnostic to the type of sensor used, and does not rely on features of a point cloud which are unique to certain types of sensor.

Our future work will include changes which allow the system to make use of additional points captured on existing surfaces to improve the quality of the surface representation. We will also consider improvements to the surface expansion process to be more robust to outliers.

References

[1] A. Escande, A. Kheddar, and S. Miossec, "Planning contact points for humanoid robots," *Robotics and Autonomous Systems*, vol. 61, no. 5, pp. 428–442, May 2013.

[2] K. Bouyarmane and A. Kheddar, "Humanoid Robot Locomotion and Manipulation Step Planning," *Advanced Robotics*, vol. 26, no. 10, pp. 1099–1126, Jun. 2012.

[3] S. Brossette, A. Escande, J. Vaillant, F. Keith, T. Moulard, and A. Kheddar, "Integration of non-inclusive contacts in posture generation," in *IROS*, Sept 2014, pp. 933–938.

[4] S.-Y. Chung and O. Khatib, "Contact-consistent elastic strips for multi-contact locomotion planning of humanoid robots," in *ICRA*, 2015, pp. 6289–6294.

[5] S. Brossette, J. Vaillant, F. Keith, A. Escande, and A. Kheddar, "Point-cloud multi-contact planning for humanoids: Preliminary results," in *IEEE Conference on Robotics, Automation and Mechatronics*, Nov 2013, pp. 19–24.

[6] R. B. Rusu and S. Cousins, "3d is here: Point Cloud Library (PCL)," in *ICRA*, Shanghai, China, May 2011.

[7] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C. T. Silva, "Computing and rendering point set surfaces," *IEEE Transactions on Visualization and Computer Graphics*, vol. 9, no. 1, pp. 3–15, 2003.

[8] T. Rabbani, F. Van Den Heuvel, and G. Vosselmann, "Segmentation of point clouds using smoothness constraint," *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. 36, no. 5, pp. 248–253, 2006.

[9] M. A. Fischler and R. C. Bolles, "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography," *Communications of the ACM*, vol. 24, no. 6, pp. 381–395, 1981.

[10] H. Edelsbrunner, D. G. Kirkpatrick, and R. Seidel, "On the shape of a set of points in the plane," *IEEE Transactions on Information Theory*, vol. 29, no. 4, pp. 551–559, 1983.

[11] J. R. Shewchuk, "Triangle: Engineering a 2d Quality Mesh Generator and Delaunay Triangulator," in *Applied Computational Geometry: Towards Geometric Engineering*, ser. Lecture Notes in Computer Science, M. C. Lin and D. Manocha, Eds. Springer-Verlag, Nay 1996, vol. 1148, pp. 203–222.