2011 IEEE International Conference on Robotics and Automation
Shanghai International Conference Center
May 9-13, 2011, Shanghai, China

A LIDAR Streaming Architecture for Mobile Robotics with Application to 3D Structure Characterization

Mayank Bansal, Bogdan Matei, Ben Southall, Jayan Eledath, Harpreet Sawhney
Vision Technologies, SRI International Sarnoff
201 Washington Road, Princeton, NJ 08540, USA
{mayank.bansal,bogdan.matei,ben.southall,jayan.eledath,harpreet.sawhney}@sri.com

Abstract— **We present a novel LIDAR streaming architecture for real-time, on-board processing using unmanned robots. We propose a two-level 3D data structure that allows pipelined and streaming processing of the 3D data as it arrives from a moving robot: (i) at the coarse level, the incoming 3D scans are stored in memory in a dense 3D voxel grid with a relatively large voxel size – this ensures buffering of the most recent data and the availability of sufficient 3D measurements within a specific processing volume at the next level; (ii) at the fine level, we employ a data chunking mechanism guided by the movement of the robot and a rolling dense 3D voxel grid for processing the data in the immediate vicinity of the robot, which enables re-use of previously computed features. The architecture proposed requires a very small memory footprint, minimal data copying, and allows a fast spatial access for 3D data, even at the finest resolutions. We illustrate the proposed streaming architecture on a real-time 3D structure characterization task for detecting doors and stairs in indoor environments and show qualitative results demonstrating the effectiveness of our approach.**

I. INTRODUCTION

The recent development of a vast variety of small and highly accurate laser sensors has greatly improved the ease with which 3D point clouds can be obtained, even from small robotic platforms. Unlike vision-based 3D sensing mechanisms using stereo, LIDAR is typically much more reliable for sensing depth, especially for untextured 3D regions and at a larger distance from the sensor. In the literature there have been presented numerous algorithms for using LIDAR for performing many tasks such as object detection [1], object recognition [2], [3], 3D modeling [4] and mapping. However, most often it was assumed that the data is first acquired over the whole area of interest and then processed off-line in a batch processing manner. There are numerous data structures which allows efficient spatial access to 3D neighbors and representation for point clouds based on k-D trees and octrees [5]. These structures however, require that the 3D distribution of points be known ahead of time for optimal space tessellation, thus they are not adequate for representing dynamic data acquired by a moving platform.

This Project Agreement Holder effort was sponsored by the U.S. Government under Other Transaction number W15QKN-08-9-0001 between the Robotics Technology Consortium, Inc, and the Government. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government.

For numerous robotics tasks such as semantic mapping and navigation or obstacle detection, the data must be processed on-board the robot platform in real-time and with minimal latency. Many robotic platforms have limited processing resources (CPU, memory) due to space, weight and battery life. LIDAR sensors have very different acquisition speeds for the 3D environment, which leads to distinct processing requirements. For example, a Velodyne sensor can acquire 1.3M points, while a Hokuyo can acquire about 20-40K points/sec. It is desirable that the architecture for LIDAR processing can be extended easily, with minimal changes, to different robotic platforms and LIDAR scanners.

In this paper, we focus on the aspects of managing and processing a large volume of point data collected by a scanning LIDAR in a streaming and pipe-lined fashion: (i) at the coarse representation level, the incoming 3D scans are stored in memory in a dense 3D voxel grid with a relatively large voxel size, which ensures buffering of the data and availability of enough 3D measurements for a specific processing volume at the next level; (ii) at the fine level, we employ a chunking mechanism guided by the movement of the robot and a rolling dense 3D voxel grid for processing the data in the immediate vicinity of the robot, which enables re-use of previously computed features. The data structures require a very small memory footprint and data copying, and allow a fast spatial access for 3D data, even at fine resolution (e.g., 10cm. or less).

A comprehensive analysis of various data structures for dynamic LIDAR processing has been done by Lalonde et al. [6]. The authors also describe a fast scrolling 3D dense grid to process the data in the immediate vicinity of the robot. Various strategies for reusing previously computed 3D features (e.g., density, surface normal estimates) are also discussed. In our work we build upon the concepts in [6] and propose a novel two-level structure that provides similar benefits of fast spatial access, re-use of computation and minimal data copying between structures. Our framework is particularly well suited to our demonstration sensor setup where a single laser scans the environment perpendicular to the robot's motion direction. Without using an expensive pan-tilt unit or a dense scanning sensor like the Velodyne, we are able to use the (pose registered) scans from the single laser in a streaming manner and demonstrate effective structure characterization. This is significant if robots are

to perform high-level tasks like object finding, in unknown environments, in live mode without having to process the over-dense point clouds typically obtained by expensive Velodyne like sensors.

In this paper we illustrate the proposed streaming architecture in conjunction with a novel algorithm to detect two particular structure types commonly found in indoor environments – doors and stairs. The proposed algorithm makes full use of the chunking infrastructure to efficiently process incoming data without redundant re-computation and is able to detect these structures in real-time as the robot is traversing through the environment. Further, the algorithm can be easily generalized to detect other object categories by defining appropriate filters. In recent years, door and stair identification has been widely studied [7], [8], [9], [10], [11]. Each of these systems uses an image, tactile or 3D data-based approach to either detecting doors directly or their handles. Similarly, systems like [12], [13] use laser or vision sensors to detect stairs. The methods from [14] employ a single 1D scan and a camera to learn walls and doors. However these methods don't need specifically to perform streaming processing and are not easily extendable to other 3D structures which cannot be discriminated based solely on an 1D scan (required for detection).

The remainder of the paper is organized as follows. The streaming framework is described in Sec. II followed by a description of our algorithm for door and stair detection in Sec. III. Experiments and results are discussed in Sec. IV followed by concluding remarks in Sec. V.

II. The Streaming Framework

Although the streaming framework we present in this paper is generally applicable to a number of mobile robotics applications, we will sometimes use our specific application and hardware setup to decide on the system parameters. The aim is to semantically label all the doors and stairs in an indoor environment using a mobile robot (shown in Fig. 1) equipped with two laser sensors. The horizontal laser scans the environment in a horizontal plane and the coronal laser scans in a vertical plane perpendicular to the robot's direction of motion (for a more detailed description of the hardware, please refer to Sec. IV). As the robot moves, the coronal scans can be registered together in a single coordinate system to get a dense 3D representation of the environment which can be used to recognize the doors and stairs. However, we want to be able to perform the recognition task in a streaming manner as the robot scans each area.

Most previous work on 3D object detection using dense 3D point clouds has relied on either batch processing of this data (after alignment from individual scans) or on capturing dense data at each instant using a 360 degree scanning LIDAR. In the following, we describe our streaming framework that allows live processing from such dense capture systems as well as from a single scan system by employing 3D grid-based data-structures for fast spatial access and by defining when it is safe to process a visited area.

Fig. 1. The Pioneer platform with the computing laptop and the scanning lasers. The coronally-scanning Hokuyo laser is mounted on the sensor mast, while the horizontally scanning unit is mounted at the front of the platform.

Notation. We will use upper-case X, Y, Z to denote real-valued coordinates in a global world coordinate system. The axes XY of the coordinate system lie in the ground-plane and the Z axis points up from the ground. Lower-case i, j, k will be used to denote integral values - typically used as indices into a 3D grid structure.

A. Point Storage

We maintain the 3D point vectors in a single linear array P of pre-initialized length L. This array functions as a circular queue and allows access to the most recent L points captured by the robot. Throughout the architecture, this array is the only place where the points are kept in memory for access by any of the other layers. Depending on the application, the value for L can be computed as a function of the laser scan rate and the speed of the robot to determine the time-duration in which the robot can move and scan a sufficient distance. For our experiments, we buffered 3 minutes worth of points. Note that for our single scan laser setup, the robot needs to be continuously moving to ensure dense 3D scanning. Therefore, in our implementation, we use a laser scan only if the robot's position has changed (by a fixed delta) since the last time a scan was used. This ensures that the 3 minute buffering does not expire previous valid data due to a stuck robot.

Any metadata associated with the points can also be maintained as separate arrays $M_1, \cdots, M_n$ parallel to this main array e.g. the point labels coming from a structure-characterization module. Specifically, we maintain an array M_0 that stores a pointer into the voxel structure for each point. This array will be useful to implement efficient point deletion as will be described later.

B. Static Voxel Array

We maintain a coarse level static voxel structure for quick retrieval of points within specific spatial bounds by the streaming data processor. The basic architecture of this structure will be re-used for the streaming voxel structure described in the next section.

At the most basic level of our 3D data-structure is the *VNode*. Each VNode represents a single voxel (of specified dimensions) in 3D space and is responsible for the 3D points contained in that space. To reduce expensive point copy operations, the VNode itself does not store the 3D points. Instead, it maintains a linked-list of integer coordinates which index into the point array P. When a laser scan arrives, each point gets added to the point array P and its index in P gets pushed into the linked-list of the VNode responsible for its spatial location. The storage of the point index also allows easy voxel-level access to any point metadata using the parallel metadata arrays $M_1, \cdots, M_n$. A pointer to the newly added linked-list node is stored in the array M_0 at the index corresponding to the added point. Since the point array P is a circular queue, points get overwritten with new points and the old points need to be deleted from the VNodes containing them. This is accomplished in constant time by directly deleting the linked-list node pointed to by the corresponding entry of the array M_0.

At the next layer above the VNode is the 3D grid structure *VArray*. This array is responsible for maintaining all the voxels within a specified spatial extent $(X_m, Y_m, Z_m) - (X_M, Y_M, Z_M)$ at a specified discretization v. These two together determine the maximum number of VNodes $N_x \times N_y \times N_z$ for which memory might be needed. Here $N_x = (X_M - X_m)/v$ is the number of grid cells along the X axis and likewise for N_y and N_z. The voxel array VArray is composed of a linked-list of VNodes and a 3-dimensional array of pointers VPtr to specific VNodes in this linked-list. Given a 3D location $\boldsymbol{p} = (X, Y, Z)$ within the spatial extent of the VArray, we can compute an integer index tuple (i, j, k) into the voxel grid structure using the equation below:

$$\begin{aligned} i &= \lfloor (X - X_m)/v \rfloor, \\ j &= \lfloor (Y - Y_m)/v \rfloor, \\ k &= \lfloor (Z - Z_m)/v \rfloor, \end{aligned} \tag{1}$$

where (X_m, Y_m, Z_m) are spatial extent coordinates and v is the voxel size.

Since the occupied space in any region of 3D spatial volume is usually sparse, we do not want to allocate VNodes for each voxel in the volume. Instead, we follow an on-demand allocation strategy for the VNodes. As each point is added, we compute its integer index tuple (i, j, k) and check if the associated VNode exists. If so, we push the point into the VNode as described earlier. If not, we allocate a new VNode, add it to the VNode list, assign its pointer to the pointer array, and finally push the point into the newly created VNode.

The VArray allows very fast retrieval of points within any given spatial bounds. First, the spatial bounds are converted using (1) to a range of integer indices (i_1, j_1, k_1) to (i_2, j_2, k_2). Then, the pointer array is traversed in the index range to find the list of VNodes within the range. Finally, the linked-lists of point indices from all the VNodes are appended together to get a master index list of the required points. The points themselves, as before, can be accessed by an $O(1)$ access from the point array P.

The multiple pointers and levels-of-indirection used in the above implementation offers crucial performance benefits in the streaming voxel array *SVArray* implementation described in the next section.

C. Streaming Voxel Array

The static voxel array *VArray* is efficient in memory and point retrieval only when the voxel discretization v is coarse. To represent a large region of space, a finer discretization would result in a huge number of VNodes being allocated which would not fit in the memory and also be slow to access. For most applications, a coarsely sampled voxel grid would not suffice as any voxel level processing such as plane fitting etc. would be very inaccurate. This implies the need for a smaller voxel array responsible for a small region of space that can be discretized at a finer resolution. In addition, for a mobile robot, this region (and hence the voxel array) has to slide around in space so that the live processing can cover the entire map navigated by the robot.

We propose a novel streaming voxel array *SVArray* that is responsible for keeping track of the 3D points in a fixed volume behind the robot. This data structure defines the volume of interest where the structure characterizer does its processing. We implement the rolling voxel array as a circular queue in two out of the three dimensions (there is no rolling in the vertical direction). This data structure avoids re-allocation of new memory as the robot moves and effectively reuses old data in the unrolled portion for structure characterization, hence avoiding unnecessary copying.

Our streaming voxel array *SVArray* shares the basic architecture with the VArray but has the capability to slide in space. This means that the extent values (X_m, Y_m, Z_m) and (X_M, Y_M, Z_M) are variable and point to the region of space currently inside the SVArray in a global coordinate system. Exactly like the VArray, the SVArray is composed of a linked-list of VNodes and a 3-dimensional array of pointers VPtr to specific VNodes in this linked-list. Now assume that we need to slide a chunk to a neighboring location along one of the axes. We would maintain some overlap with the previous chunk to avoid border effects in the successive processing. In this overlap region, we want to reuse the voxel structure from the previous chunk and avoid any copying of points or voxel data. This is achieved by implementing the VPtr structure as a circular queue in the three dimensions. The key idea is to keep the VPtr structure intact in the overlap region and reuse the non-overlap region for storing new data.

We maintain three additional dynamic indices i_m, j_m, k_m which point to the voxel grid cell containing the smallest (X, Y, Z) coordinates.

In this modified VArray i.e. the SVArray, the world coordinate (X, Y, Z) to grid index (i, j, k) conversion equations (1) need to be adapted as follows:

$$\begin{aligned} i &= (\lfloor (X - X_m)/v \rfloor + i_m) \bmod N_x, \\ j &= (\lfloor (Y - Y_m)/v \rfloor + j_m) \bmod N_y, \\ k &= (\lfloor (Z - Z_m)/v \rfloor + k_m) \bmod N_z, \end{aligned} \tag{2}$$

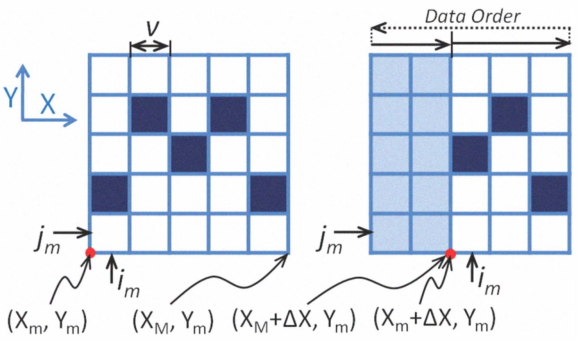


Fig. 2. SVArray: Illustration of a roll along the X-axis by $\Delta X = 2v$. The dark shaded cells depict filled voxels and they remain the same in the overlap region between the rolls. The light shaded cells depict the new region R^{new} and the voxels in this region from the previous chunk have been invalidated.

where (X_m, Y_m, Z_m) are the spatial extent coordinates, v is the voxel size and N_x, N_y, N_z are the number of cells along the X, Y, Z direction. In our system we have used $v = 0.1$ m.

We maintain three additional arrays of VNode indices to efficiently manage the allocation and deallocation of VNodes. These lists are called SliceX$[0, \ldots, N_x]$, SliceY$[0, \ldots, N_y]$ and SliceZ$[0, \ldots, N_z]$ and their names refer to the fact that they essentially maintain an account of the VNodes allocated in a given "slice" of the voxel grid structure. For example, SliceX$[i_0]$ is a list of all VNodes with indices in $\{(i_0, j, k) | j \in \{0, \ldots, N_y - 1\}, k \in \{0, \ldots, N_z - 1\}\}$. Every time a new VNode is created, its index tuple (i, j, k) is pushed into the slices SliceX$[i]$, SliceY$[j]$ and SliceZ$[k]$.

Let us now look at what is involved when the chunk contained in the current SVArray needs to move to a neighboring region (see Fig. 2). Suppose that the chunk moves from the region $R^t = [(X_m, Y_m, Z_m), (X_M, Y_M, Z_M)]$ to $R^{t+1} = [(X_m{+}\Delta X, Y_m, Z_m), (X_M{+}\Delta X, Y_M, Z_M)]$ which is a movement by ΔX only along the X-axis (we will show how to generalize this later). Assuming $X_m + \Delta X < X_M$, the new chunk position overlaps with the old position in the region $R^{olap} = [(X_m + \Delta X, Y_m, Z_m), (X_M, Y_M, Z_M)]$. To go from R^t to R^{t+1}, we update the values of the dynamic indices as follows. For $\Delta X > 0$,

$$\begin{aligned} i_m^{t+1} &= (i_m^t + \Delta X/v) \bmod N_x, \\ j_m^{t+1} &= j_m^t, \\ k_m^{t+1} &= k_m^t \end{aligned} \tag{3}$$

and for $\Delta X < 0$,

$$\begin{aligned} i_m^{t+1} &= (i_m^t + \Delta X/v + N_x) \bmod N_x, \\ j_m^{t+1} &= j_m^t, \\ k_m^{t+1} &= k_m^t \end{aligned} \tag{4}$$

Let the index range between the old i_m^t and new i_m^{t+1} be $I_- = \{i_m^t, \ldots, N_x - 1, 0, \ldots, i_m^{t+1} - 1\}$ (and correspondingly $I_- = \{i_m^{t+1}, \ldots, N_x - 1, 0, \ldots, i_m^t - 1\}$ for $\Delta X < 0$). Then, the VNodes in the index range $i_- \in I_-$, $j_- \in \{0, \ldots, N_y - 1\}$ and $k_- \in \{0, \ldots, N_z - 1\}$ need to be invalidated to mark that the data there is old. This is precisely the set of VNodes pointed to by SliceX$[I_-]$. Thus, we can efficiently invalidate these VNodes by a single pass through these "X-Slices". A VNode invalidation clears its list of point indices and adds this VNode to a separate inventory (list) of pre-allocated VNodes. Whenever a new VNode is needed, this inventory is checked first and only then is an attempt made to allocate a new VNode. This reduces the overhead of VNode allocation at run-time and is particularly suited for SVArray as each roll invalidates and creates a large number of VNodes.

The update equations for a roll along the Y or Z direction are symmetric to the above equations. The region $R^{new} = R^{t+1} - R^{olap}$ is output as the coordinate range where new points are desired. Any general roll in the XY plane can be composed as a collection of rolls along the X and Y directions as we will discuss in the next section. To allow for a symmetric roll along the X and Y axes, we assume that, once initialized, the SVArray has a constant extent $(X_M - X_m)$ and that it is the same along the X and Y dimensions i.e. $(X_M - X_m) = (Y_M - Y_m)$.

We enforce a constraint on the roll magnitude that $\Delta X < \alpha * (X_M - X_m)$ to ensure that at least $\alpha\%$ of the voxel structure is re-used between rolls. Since there is always an overlap of voxel structure between successive rolls, we can compute the modulus operation in the above equations much more efficiently as follows:

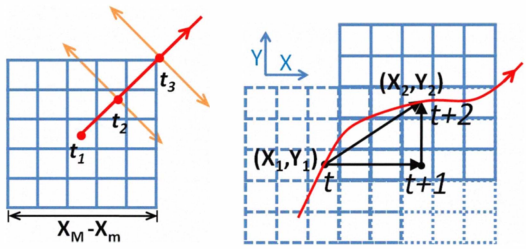
$$i \bmod N_x = \begin{cases} i - N_x & \text{if } i >= N_x \\ i & \text{otherwise} \end{cases} \tag{5}$$

and i is always ensured to be non-negative in all the previous equations.

D. Data Chunking

As the robot moves about scanning the environment around it, we need to identify the extents of the "chunk" of 3D space that is ready to be processed by the application. A chunk is ready when the entire volume inside it has been scanned by the robot at least once. Therefore, the strategy for chunk selection will be different depending on the positioning and setup of the laser sensors on the robot. Our streaming framework also puts additional constraints on the way a chunk can be selected due to the computed data re-use strategy we employ. In the following, we describe our chunking algorithm which is suitable for a coronal scanning laser setup (i.e. each scan from the laser is an arc in a plane perpendicular to the robot's direction of motion) and also satisfies the constraints from the streaming framework.

The coronal laser setup constrains the way a chunk can be selected in a number of ways. First, the latest scan comes from the robot's current position so the chunk boundary cannot extend beyond this point (as that region potentially has not been scanned before). Second, the chunks being aligned to the global XY axes, the current orientation of the robot makes it difficult to choose the correct chunk boundary without causing border artifacts. Third, when the chunk is rolled it only requests new data in the region not-overlapping



(a) (b)

Fig. 3. Data Chunking: (a) The robot's path is shown in red with pose markers at times t_1, t_2 and t_3 shown as red dots. The orange arrows depict the coronal scanning laser. The depicted chunk can only be positioned here (the pose at time t_1) when the robot has moved to at least the pose at time t_3. This ensures that none of the coronal scans that might have contributed data to this chunk are missed. (b) A displacement from (X_1, Y_1) to (X_2, Y_2) is divided into two rolls – an X-Roll from t to $t+1$ and a Y-Roll from $t+1$ to $t+2$.

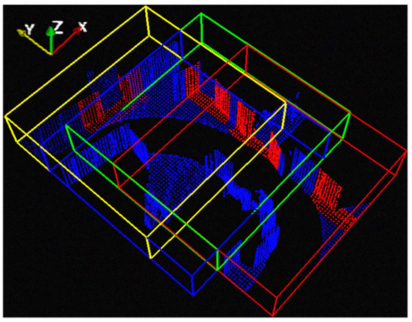


Fig. 4. Chunking Example. The figure shows four positions of the chunk as the robot traverses a curved path around an office corridor. The bounding boxes are color coded in order of time: Red, Green, Blue, Yellow. The offsets show a roll in X from Green to Blue and rolls in Y in all other cases. Also shown are the composite point clouds with Red points marking detected doors and Blue points marking unlabeled structure.

with the previous chunk extent. This means that we cannot expect any data missing within the current chunk bounds to be filled in the next roll - the current chunk has to ensure that its boundaries do not extend into the to-be-scanned-in-future zones.

We maintain a queue Q of potential future locations (X, Y) for the chunk center. These locations are sampled from the path of the robot in a streaming manner. For each pose update from the robot, we first compute the Euclidean distances between this pose and the elements at the front $Q \rightarrow front$ and end $Q \rightarrow end$ of the queue as d_{front} and d_{end}, respectively. If the distance d_{end} is greater than a threshold t_{olap}, then the pose is pushed into the queue, else it is ignored. Thus, this threshold determines the sampling rate of chunks along the robot's path as well as the overlap between successive chunks. Similarly, if the distance d_{front} is greater than another threshold t_{min}, then it is safe to move the chunk to the location $Q \rightarrow front$. This is illustrated in Fig. 3(a). From the figure, it is clear that the threshold t_{min} should be set to the value $t_{min} = (X_M - X_m) * \sqrt{2}/2$ to ensure that at the new chunk position, the entire extent of the chunk has been covered by the robot's path. This, in turn, always keeps the processing chunk trailing behind the robot as desired for our coronal sensor configuration. Note that this algorithm ensures that no chunks are generated when the robot is stationary or moving very slowly; for slower processing algorithms, this feature can be combined with the robot control loop to slow down the robot whenever the processing trails behind.

Give the above procedure to generate locations for the chunk center position, let us now look at how we accomplish this move. Consider two successive pose markers (X_1, Y_1, Z) and (X_2, Y_2, Z) where we assume that there is no change in the robot's altitude. The center of the chunk needs to be moved by the 2D displacement vector $(X_2 - X_1, Y_2 - Y_1)$. Since we allow only X and Y rolls for the chunk, this displacement can be divided into a series of X and Y rolls by a line rasterization algorithm like [15]. The sampling resolution for the rasterization grid is chosen depending on the fidelity requirement of the processed chunks w.r.t the actual path of the robot. In our implementation, we found the coarsest sampling to be adequate i.e. the displacement is simply divided into just two rolls: first an X-Roll of $(X_2 - X_1)$ and then a Y-Roll of $(Y_2 - Y_1)$ as shown in Fig. 3(b). Fig. 4 shows a chunking (and structure characterization) example from one of our experimental runs.

It is to be noted that one drawback of rolling w.r.t fixed global coordinate axes is that a diagonal path will be split into X and Y rolls even though it is more efficient to just slide along the local dominant direction. However, we have not seen this to have noticeable impact on our system performance in practice.

E. Example Computational Blocks

A basic but important piece of information available from dense 3D data is the ability to determine planar structures. These planes can be then be used for a number of applications like structure characterization, object finding etc. In the next section, we will describe one such application where we find doors and stairs in indoor environments using a mobile robot. In the following, we briefly discuss two example low-level tasks that are computationally expensive but can be re-used between chunks.

Plane Fitting. Plane estimation can be performed at the allocated VNodes (voxels) by computing the covariance matrix of all the points within a radius from each voxel centroid (computed as the average of points within the given voxel) and then using it to compute the local plane normal. However, it is necessary to perform the plane estimation at multiple radii in order to ensure robustness to registration noise and to outliers caused by nearby structures at surface discontinuities. This requires access to neighboring voxels inside the SVArray structure.

3D Invariant Features. Objects can be recognized directly from LIDAR using rotationally invariant 3D features such as spin-images or shape context [3]. These features typically require larger 3D scales compared with planar estimates. Therefore, voxel neighborhood access is important for this task as well.

For both the examples above, neighbor access is an important capability. In the SVArray structure, this is eas-

ily accomplished by using the index tuple (i, j, k) for a particular VNode in conjunction with the VPtr structure. We first compute the index tuple for the neighbor as say $((i+1) \bmod N_x, j, k)$, and then follow the VPtr link to the actual VNode. The actual voxel-level results from tasks like the above can be maintained as links from the corresponding VNodes.

F. Chunk Transitioning

The chunking mechanism ensures that there is a minimum overlap between successive chunk positions. As described in Sec. II-C, after the chunk rolls over to the new location, the chunk roll mechanism outputs the bounds of the unprocessed region R^{new} where new points need to be filled. We use these bounds to retrieve a supremum list of point indices in this region from the static voxel array VArray. These points are then pushed into the SVArray which rejects any points not in the correct bounds. Note that the list of points obtained from the VArray is a supremum because of its coarser resolution relative to the SVArray. Also note that no actual copying of points occurs in this process - only the index lists of points inside the VNodes are populated in the SVArray structure.

Our streaming framework allows effective re-use of fitted planes and/or 3D features from one chunk to the next. The SVArray structure only allows processing for the newly filled voxels in the region R^{new}. The processed planes and other features for the overlap region are re-used from the last chunk processing (since they are available by following a link from the corresponding VNodes). For applications like object detection, to avoid missing objects at the edges of a chunk, some parts of the computation still have to be carried out on the whole chunk each time but these steps are usually faster than the low-level tasks like plane fitting. As we describe in the application section, in our algorithm for door and stair finding, the matched filtering is such a task and it is performed very quickly on the entire chunk in each cycle.

III. Application: Structure Characterization

In this section, we describe an application of the proposed streaming architecture for live processing of 3D data for detection of doors and stairs in the environment. The approach clearly outlines the general principles involved in designing an even more complex application using the architecture.

In the following, we describe the door and stair detection algorithm as illustrated with an example in Fig. 5. The algorithm processes each SVArray chunk as follows:

1) For each *unprocessed* voxel in the chunk under consideration:
   a) Fit a local 3D-plane using the method described in Sec.II-E. If the estimation matrix is badly conditioned, the voxel is marked as a non-planar voxel.
   b) For each planar voxel, use the estimated plane normal direction to decide if the voxel contains a locally planar horizontal surface or a vertical surface.
2) Create a 2D grid in the ground-plane coordinate system to accumulate support from the scanned vertical structure. All voxels containing vertical surfaces are vertically projected to this 2D grid giving a *vertical histogram* representation. Each cell in this histogram measures how many voxels directly above this cell contain vertical surfaces.
3) Repeat the above step for the scanned horizontal structure to create *horizontal histogram.*
4) Compute a Radon transform of the vertical histogram to estimate the direction of maximum variation in the histogram. Since most structure corresponds to wall like structures, the Radon transform will give distinct maximum along the wall direction. Use the computed direction to axis-align both the vertical and the horizontal histograms so that the doors are aligned with either the X or the Y direction.
5) Apply object specific matched filters to the axis-aligned histograms and compute the filter responses.
6) Combine the vertical and horizontal filter responses using an appropriate function. In our implementation, we use the geometric mean of the two as the overall response.
7) Threshold the response to find the detection locations and perform non-maximal suppression to suppress spurious responses. Rotate the response map to the original orientation so that the detection location can be directly mapped to the voxels and hence the point-cloud.

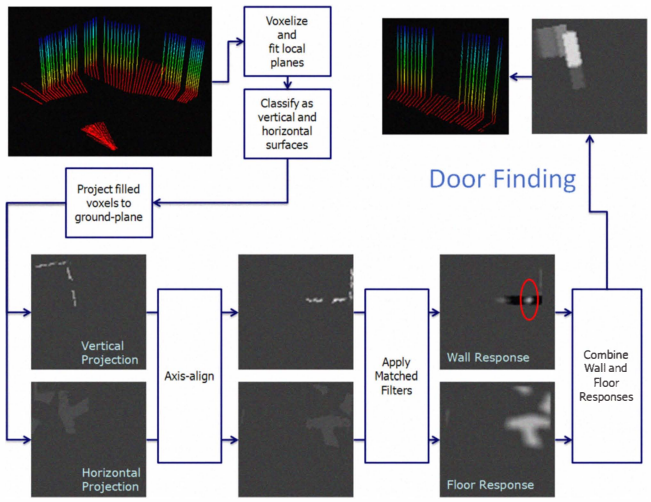


Fig. 5. Door detection flowchart. The algorithm uses a matched filter on the shape of the projected vertical and horizontal structure defining a door.

Step (1) above is the compute intensive step and happens only on the unprocessed voxels. The rest of the steps involve filtering and other image processing operations on a 2D grid which has a resolution of $N_x \times N_y$ (i.e. the SVArray). These steps are therefore very efficient.

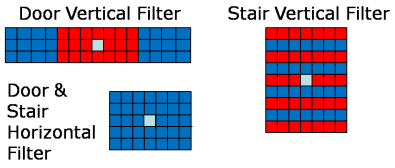


Fig. 6. Matched Filters. Blue denotes positive filter weights and red denotes negative filter weights. The light blue location is the center of the filter where the evaluation result goes.

A. Matched Filters

The matched filters for door and stair detection are designed based on the intuition of what these structures look like in the scanned data. The filters are depicted in Fig. 6.

For a door, we need to spot the presence of vertical structure with a door-sized gap in between and a scan of the floor in that gap. The floor scan verifies that the gap is traversable and is hence a door. The matched filter for the vertical (wall-response) histogram is thus the negative of a top-hat filter - the tails give positive weight to the presence of any wall-like structure while the center penalizes any wall-like structure. Similarly, the matched filter for the horizontal histogram is just a box-car filter which gives positive weight to the presence of a floor-scan which is at least the width of a door. The combination of the two matched filter responses thus evaluates if a door-width gap is present between vertical structures with a floor-scan in the gap.

For stairs, the vertical and horizontal histogram pattern is much more distinct. The vertical histogram contains a grating-like structure corresponding to vertical surfaces separated by horizontal structures. The horizontal histogram just provides evidence of the presence of a continuous horizontal structure. Thus, the matched filter for the vertical histogram is designed as a rectangular block with alternating rows containing positive and negative weights. We look for the presence of at least 5 steps in the matched filter to make the stair detection robust. The horizontal matched filter is again just a box-car filter.

In this paper, we illustrated the filters for these two specific objects. However, the approach is general enough to be suitable for detecting other types of objects by design of an appropriate filter.

IV. Experiments

A. Experimental Setup

Our hardware platform is a Pioneer P3AT equipped with both computing (a Dell M6400 Quad-core laptop) and sensor payloads. As shown in Fig. 1, our sensor payload is comprised of a number of laser, video and inertial sensors but we used only the laser sensors for all the experiments. These are the two Hokuyo UTM-30LX scanning laser rangefinders; one is mounted in horizontal position for 2D floor-plan style mapping. The second is mounted to scan in a coronal fashion, providing data in a vertical plane (in a 270 degree sector) normal to the robots direction of motion.

In our live system, an on-board pose estimation module uses the horizontal scans and wheel odometry to estimate a global pose of the robot. The coronal scans are then transformed into the global coordinate system (using the estimated pose) and are presented to the streaming architecture for live processing by the structure characterization algorithm. The structure characterizer populates the metadata array M_1 with the computed label for each point.

TABLE I
Parameter Settings

	X_m(m)	X_M(m)	Y_m(m)	Y_M(m)	Z_m(m)	Z_M(m)
VArray	−200.0	200.0	−200.0	200.0	−5.0	5.0
SVArray	0.0	8.0	−4.0	4.0	−0.05	1.45
	v(m)	N_x	N_y	N_z	α	t_{olap}(m)
VArray	0.5	800	800	20	-	-
SVArray	0.1	80	80	15	0.75	2.0

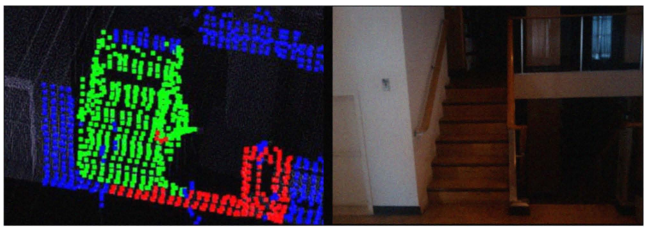


Fig. 7. Example of detection of a door and a staircase. Red points mark the two sides of the detected door and green points mark the staircase. Unlabeled points are colored blue. The image is shown only for reference.

B. Parameter Settings

All the parameter settings used for the experiments are shown in Table-I. We buffered 3 minutes worth of points in the point array which amounts to a length of:

$$L = 1080\,\text{points/scan} \times 40\,\text{scans/sec} \times 60 \times 3\,\text{mins}$$

C. Results

We extensively tested our system in different environments including a cluttered basement area with numerous doors and stairs. The robot was set in an autonomous exploration mode where it simultaneously mapped and characterized the environment with all the processing happening on the on-board computer. The characterization results from our system were delivered in real-time while the robot drove at approximately 0.3 m/s.

Fig. 7 shows an example of detection of a staircase and a doorway. Fig. 4 shows some door detection results around a turn in an office environment. Fig. 8 gives some more examples. The first row shows results for a clutter-free office corridor with the right image showing all the doors previously detected along the L-shaped corridor as white boxes along with the chunk currently being processed. The second and third rows demonstrate the robustness of our algorithm in a cluttered basement. In the picture with the door result, one can observe many (door-like) indented structures in the opposite wall that our system did not get fooled by. The clutter in this environment can be appreciated in the pictures in the third row - including a false-positive staircase picked up by our system on a stack of boxes.

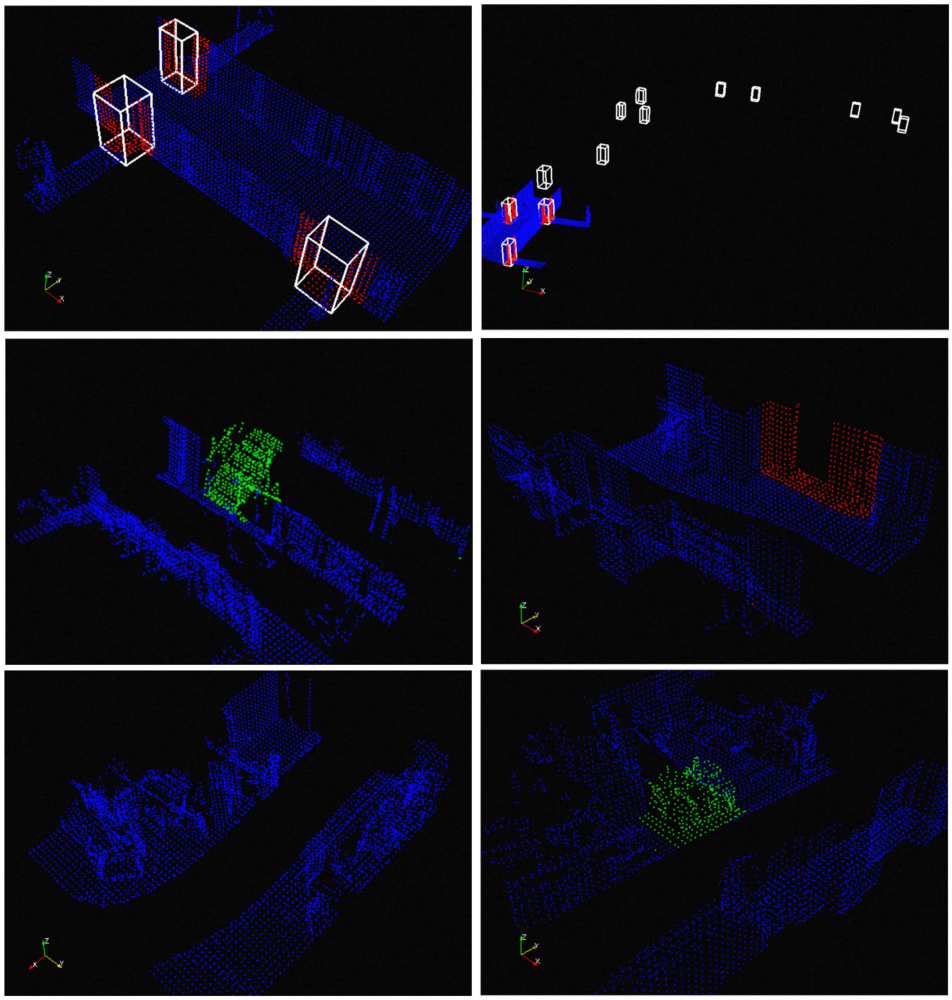


Fig. 8. A few examples of detection of doors (in red) and stairs (in green). Unlabeled points within the current chunk are colored dark blue. The first row shows door detection in a typical office environment with the right image showing a trail of all the detected doors (white wireframes) around an L-shaped corridor. The second and third rows show the system robustness in a cluttered basement. The third row specifically shows the large amount of clutter in the environment as well as a stair-like structure picked up as a false-positive. *This figure is best seen in color.*

V. Conclusion

In this paper, we have outlined an architecture for streaming processing of LIDAR-data from a moving robot. We have demonstrated the effectiveness of the framework by a real structure characterization application that detects doors and stairs in live streaming mode. In future, we plan to extend the architecture into a multi-level rolling representation and also evaluate system performance for more difficult structure characterization problems.

References

[1] L. Spinello, K. O. Arras, R. Triebel, and R. Siegwart, "A layered approach to people detection in 3d range data," in *24-th AAAI Conference on Artificial Intelligence*, 2010.

[2] M. Eich, M. Dabrowska, and F. Kirchner, "Semantic labeling: Classification of 3d entities based on spatial feature descriptors," in *Workshop Best Practice in 3D Perception and Modeling for Mobile Manipulation (ICRA-2010)*, 2010.

[3] B. Matei, Y. Shan, H. S. Sawhney, Y. Tan, R. Kumar, D. Huber, and M. Hebert, "Rapid object indexing using locality sensitive hashing and joint 3d-signature space estimation," *PAMI*, vol. 28, 2006.

[4] B. Matei, H. Sawhney, S. Samarasekera, J. Kim, and R. Kumar, "Building segmentation for densely built urban regions using aerial lidar data," in *CVPR*, 2008.

[5] H. Samet, *Multidimensional and metric data structures*. Morgan-Kaufmann, 2006.

[6] J.-F. Lalonde, N. Vandapel, and M. Hebert, "Data structure for efficient dynamic processing in 3-d," vol. 26, October 2007.

[7] E. Aude, E. Lopes, C. Aguiar, and M. Martins, "Door Crossing and State Identification Using Robotic Vision," in *8th International IFAC Symposium on Robot Control (Syroco 2006), Bologna, Italy*, 2006.

[8] A. Jain and C. Kemp, "Behaviors for robust door opening and doorway traversal with a force-sensing mobile manipulator," in *RSS Manipulation Workshop: Intelligence in Human Environments*, 2008.

[9] E. Klingbeil, A. Saxena, and A. Ng, "Learning to open new doors," in *Robotics Science and Systems (RSS) workshop on Robot Manipulation*, 2008.

[10] A. Jain and C. Kemp, "Behavior-based door opening with equilibrium point control," in *RSS Workshop: Mobile Manipulation in Human Environments*, 2009.

[11] R. Rusu, W. Meeussen, S. Chitta, and M. Beetz, "Laser-based perception for door and handle identification," in *Proceedings of International Conference on Advanced Robotics*, 2009.

[12] M. Fair and D. Miller, "Automated Staircase Detection, Alignment & Traversal," in *Proceedings of International Conference on Robotics and Manufacturing*. Citeseer, pp. 218–222.

[13] S. Se and M. Brady, "Vision-based detection of staircases," in *Proc. Asian conf. computer vision*. Citeseer, 2000, pp. 535–540.

[14] D. Anguelov, D. Koller, E. Parker, and S. Thrun, "Detecting and modeling doors with mobile robots," in *ICRA*, 2004.

[15] J. Bresenham, "Pixel-processing fundamentals," *IEEE Computer Graphics and Applications*, vol. 16, no. 1, pp. 74–82, 1996.