Four Dimensional Mapping with RGB-D Sensors and Mobile Robots

Tanner Schmidt

April 13, 2012
# Contents

1 Introduction  
   1.1 Related Work and Contributions  
   1.2 Background  

2 Map Construction  
   2.1 Individual Map Construction  
   2.2 Noise Reduction Methods  
   2.3 A Data Structure for Maps with Temporal Information  
   2.4 Results  

3 Potential Applications  
   3.1 Differencing Methods  
      3.1.1 Hard Differencing  
      3.1.2 Threshold Differencing  
      3.1.3 Continuous Differencing  
   3.2 An Efficient Method for Object Segmentation in Map Differences  
   3.3 Future Work
Acknowledgements

I would like to thank to my advisors, Jeffrey Forbes and Ronald Parr, for all their help with this project. I would also like to thank my graduate mentor, Mac Mason, for his help and the use of his code.
Abstract

Mobile robots are often tasked with 3D mapping problems in dynamic environments. In such environments, to store a map of only the most recently observed state of the map is to ignore useful information. Storing map history will allow robots to make decisions based not only on the current state of the observed world, but also on previously recognized patterns. We present a memory-efficient, octree-based method of storing 3D maps with history, and a method for reducing the impact of sensor and localization noise on the memory requirements and quality of the map. We also describe map differencing approaches and an algorithm for cluster segmentation in octrees that are applicable to four-dimensional mapping applications.

1 Introduction

3D mapping and environment modeling methods have a number of uses in robotics applications, including, but not limited to, navigation, planning, manipulation, autonomous exploration, and teleoperation. There are also interesting applications outside of robotics, such as telepresence or the preservation of 3D objects of artistic and/or historical value via digital copies[16].

Extending mapping as a four-dimensional problem has all the benefits of three-dimensional mapping, with the additional ability to remember past states of the environment or to do probabilistic reasoning about the manner in which the environment has changed and likely current or future states of the observed world.
1.1 Related Work and Contributions

There have been a wide variety of approaches taken to solve the 3D mapping problem, including monocular vision[5], stereo vision[21], laser range scans[23][18], videos[19], or unsorted sets of photos downloaded from image-sharing websites[22][2]. More recently, the advent of cheap RGB-D sensors such as the Kinect[1] has provided the robotics and computer vision communities with a new sensor for map construction[6][13][9][17].

The robotics community has also recognized the need for mapping methods that are robust in dynamic environments, dating back to 2D techniques[4][10]. Many of the existing 3D RGB-D mapping algorithms take dynamism in the environment into account and allow for overwriting previously observed points. There has also been work on leveraging the difference between consecutive observations to infer object models[12]. However, even mapping applications that do not make static-world assumptions typically discard past states of the environment in favor of the most recent observations. Therefore, our contribution lies mainly in the addition of memory to the mapping problem.

While approaches such as Kinect Fusion produce compelling and detailed reconstructions of the environment being mapped, the map size is limited to about a single room, and the storage requirements would be large if memory were to be added to the system[13]. RGB-D mapping, on the other hand, is capable of mapping building-sized environments such as the one we were aiming for, but stores data in a point cloud format that also does not lend itself easily to storing temporally-aware maps[11]. We opted to instead base our work on the octree data structure[14], which has the benefit of quantizing point clouds to allow for compression along the time-axis, as well as providing a format that is easily used as an occupancy grid for navigation tasks[7].
1.2 Background

This project started with mounting a Kinect on an iRobot Create robot platform. In order to make the robot truly mobile, I had to remove the need to tether the Kinect to a wall outlet, which I accomplished with a simple circuit allowing the Kinect to draw power from the Create battery. I then began learning my way around the ROS software, particularly the packages for interfacing with the Kinect and the point cloud library[20], and eventually developing my own software within the ROS framework.

The original plan was to store a simple octree, the nodes of which took one of three states: occupied, free, or unknown. While this required much less space to store than the model eventually adopted, described in section 2.1, it was not a good choice in light of the noise level of the Kinect sensor, and there was high level of volatility in the observed values of voxels. It was for this reason that I then moved to the probabilistic model.

Finally, I also realized that in order to build large maps with the Create, I would need to solve the localization problem. Rather than attempt to develop in a parallel something similar to KinectFusion or RGB-D mapping, I moved to using a dataset collected by a PR-2 that included pose estimates, with the assumption that a 3D SLAM approach could be used in the future to generate the point clouds and pose estimates that would be the input to the system presented in section 2.
2 Map Construction

Our goal was to construct a map of a building-size environment that stores the state of the environment from the first observation to the most recent. We assume that the input is a number of data sets collected by an RGB-D camera mounted on a mobile robot, and the time at which each data set was collected. Each data set is comprised of a series of point clouds (or RGB-D videos) with pose estimates for each point cloud (or frame). We do not do any alignment using the unprocessed point cloud or image data, so we assume relatively accurate pose estimates. These estimates could be gathered, for example, using a 2D map and standard localization techniques.

2.1 Individual Map Construction

The first step in creating a temporally-aware map is to convert each data set into an octree. This comes first because the octree representation of a set of point cloud data requires much less storage space than the point cloud data itself, due to quantization and subsequent aggregation of nearby data points. We envision a system where new data can be incrementally added to the map, so despite the potential utility of having the raw point cloud data for each time sample, it would not be practical to store that information.

We start with a data set $d$ consisting of a set of point clouds, $C = \{c_0, c_1, ... c_n\}$, where point cloud $c_i$ in turn consists of a set of points $P_i = \{p_{i,0}, p_{i,1}, ... p_{i,m}\}$ and a camera origin, $o_i$. Each point stores its 3D coordinates, and a color using RGB values. We output an octree $t$, which we define with an initial size and the smallest size to which
a voxel can be refined, effectively limiting the depth of the tree. We first discard any point \( p_{i,j} \) that is further than 3 meters from its respective camera origin \( o_i \), due to the high error in depth values reported by the Kinect at distances far from the sensor\[15\].

For each remaining point in each \( P_i \), we cast a ray through the octree from \( o_i \) to \( p_{i,j} \) as described in \[3\]. We refine nodes in the octree such that every node traversed by the ray is of the smallest size allowable by tree \( t \); we will refer to such nodes as being of 'unit size'. For each unit size node \( n_i \) along the path of the ray, we calculate the length of the segment of the ray that traversed \( n_i \) and add it to a running counter of the total length \( l_i \) of all rays that have passed through \( n_i \). For the last node on the path, i.e. the unit-sized node in which the ray terminates, we increment a counter of the total number of rays that have terminated in that node, \( x_i \), as well as counters for the total color components of the corresponding points. We can calculate the probability that any node \( n_i \) is occupied by:

\[
P(n_i) = 1 - e^{-x_i/l_i}
\]

which is based on DP-SLAM 2.0\[8\]. The case where \( l_i = 0 \) is defined as a special case in which the occupancy state of \( n_i \) is unknown, which is treated differently from cases where \( l_i > 0 \) and \( x_i = 0 \), in which case \( n_i \) is considered a known-free voxel. This distinction becomes important when comparing maps, as an event in which node \( n_i \) that is occupied at time \( t_0 \), i.e. \( P(n_i) > 0.5 \), and unoccupied at time \( t_1 \) should be considered a change only if \( n_i \) is known-free at \( t_1 \), and not if it was simply unobserved.

After casting all points in \( d \), a compression step may be run on the tree to further save space. A node \( n_i \) with 8 known-free children can be compressed by summing \( l \) and
for all children, storing the information in $n_i$, and deleting the children.

### 2.2 Noise Reduction Methods

The ability to determine how an environment changes over time requires multiple observations of the environment. However, in our experiment as described in section 2.4, the pose estimates were not completely accurate despite the fact that each octree was cast based on localization using the same 2D map. Therefore, objects appearing in multiple maps may be shifted by a number of voxels from map to map, depending on the octree resolution.

We overcome the problem of localization noise and map misalignment by using the sum of the data stored in all maps. Starting with $N$ separate maps constructed from data gathered at various times, we build a consensus map by essentially averaging all the data contained in all $N$ maps (i.e., summing the hit, color, and ray length counters for all nodes representing the same volume). In the case of free voxels of non-unit size, it was assumed that the odometer count should be evenly distributed amongst all the constituent unit size voxels. The intuition behind map averaging is that, by the central limit theorem, the averaged map should approach ground truth in the environment. For dynamic regions of the environment, the assumption is that brief changes would be averaged out to free voxels, while sustained changes would appear in the average. Areas of frequent change would result in a sort of smudge in the average, which could cause problems for nearby objects that are the same color as parts of the smudge.

Once the average map is acquired, a simple heuristic is employed to re-align voxels in each map based on the average map state. We want to differentiate real changes
from misalignment, so we will look at the neighbors of seemingly nearby voxels. If a neighbor is sufficiently similar, the shift in position is likely due to alignment error, and we simply move the incongruent data. To define similarity, we score the fitness \( s_{i,j} \) of a pair of nodes \( n_i \) and \( n_j \) using:

\[
    d_{i,j} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}
\]

\[
    c_{i,j} = \sqrt{(r_i - r_j)^2 + (g_i - g_j)^2 + (b_i - b_j)^2}
\]

\[
    p_{i,j} = |P(n_i) - P(n_j)|
\]

\[
    s_{i,j} = \frac{1}{(d_{i,j}^{\omega_d})(c_{i,j}^{\omega_c})(p_{i,j}^{\omega_p})}
\]

where \( \omega_d, \omega_c, \) and \( \omega_p \) are parameters of the algorithm that determine the relative weighting of the distance, color, and probability terms.

The process starts with taking a hard difference between a single map \( m \) consisting of nodes \( N = \{n_0, n_1, \ldots, n_i\} \) and the averaged map \( m_{\text{avg}} \), consisting of nodes \( N_{\text{avg}} = \{n_{0,\text{avg}}, n_{1,\text{avg}}, \ldots, n_{i,\text{avg}}\} \), where it is assumed \( n_i \) represents the same volume as \( n_{i,\text{avg}} \). This results in an octree containing all nodes \( n_i \) from map \( m \) where \( P(n_i) > 0.5 \neq P(n_{i,\text{avg}}) > 0.5 \). This isolates all nodes in \( m \) that are inconsistent with the average case in \( m_{\text{avg}} \). If \( n_i \) is an occupied voxel, i.e. \( P(n_i) > 0.5 \), we iterate through the neighbors of \( n_{i,\text{avg}} \) in \( m_{\text{avg}} \), within a certain maximum distance \( d_{\text{max}} \), scoring them against \( n_i \). If the highest scoring neighbor \( n_{j,\text{avg}} \) in the average map scores above some pre-defined threshold \( \alpha \), we move the data stored in \( n_i \) in map \( m \) to node \( n_j \) corresponding to the highest scoring neighbor. If on the other hand, \( n_i \) is a free voxel, we iterate through...
the neighbors of $n_i$ and score them against $n_{i,\text{avg}}$. If the highest scoring neighbor $n_{j,\text{avg}}$ scores above $\alpha$, we move the data stored in $n_j$ to node $n_i$.

### 2.3 A Data Structure for Maps with Temporal Information

Once individual maps have been aligned with the global average map, they will also be more consistent with each other. Furthermore, in indoor environments such as the one in which our experiments were run, most of the volume does not change occupancy state between two consecutive mapping runs. It is thus a wasteful practice to store full octrees representing a series of maps of the same environment constructed at different times. Rather, we propose a modified octree structure that takes into account temporal information using a run-length encoding scheme, and can thus be considered a data structure for storing four-dimensional data.
Figure 2: (a) A visualization of a simple example of the modified octree structure. The circles represent nodes as in Figure 1, and the rectangles represent the maps that take the place of nodes in the traditional structure. The timestamp represents the key to the map. (b) A visualization of the series of octrees represented by the data structure. The full octree as it was at each of the three timestamps present as keys in the node mappings can be reconstructed from the data structure.
Whereas a traditional octree node maintains references to up to eight children nodes, a modified octree node maintains references to eight maps. The keys are timestamps, and the values are modified children nodes. It is assumed that a child node (and all of its children, recursively) is valid from the timestamp with which it is associated until the next timestamp in the key set of the parent node. This data structure has a number of desirable properties:

- A change to a leaf node does not invalidate its parent node, which means only one new node needs to be created when a leaf node changes.

- The depth of the tree is only dependent on the volume of the environment being mapped and the minimal allowed voxel size.

- The passage of time and accretion of changes only causes the tree to grow horizontally, i.e. by adding new entries in the maps. The depth of the tree is constant over time.

- Inserting a new node into the tree has essentially the same complexity as inserting a new node into a pure octree, with some additional constant time for insertion into the map.

- The 3D map as it was most recently observed at any point in time can thus be reconstructed by recursing through the tree, looking up the children that were valid at the target time at each level of the tree. Because the depth of the tree does not grow with additional maps, the time required to reconstruct snapshots is also roughly constant over time.
• Only changes in the map require the allocation of new map nodes. Volumes that remain constant over time do not require any changes to the data structure.

• The structure could theoretically be used as a sort of '3D movie' format, given proper visualization tools.

2.4 Results

We tested our methods on a data set consisting of 66 point cloud sets \( P = \{p_0, p_1, \ldots, p_{65}\} \), collected by a PR-2 robot at the Willow Garage offices. The point clouds were provided via a Kinect mounted on the head of the PR-2. Pose estimates were provided by localizing scans from the base laser of the PR-2 using a pre-computed 2D map of the environment, as seen in Figure 4. The coverage of each map was variable, so not every part of the map was observed an equal number of times. Point cloud sets were collected about two or three times a day over the course of a month and a half.

The maps were originally stored as individual octrees using a standard octree implementation. They averaged about 1.4 million nodes per map, for a total of almost 93 million nodes to store all the trees separately. However, given that all of these maps were of the same environment, which only exhibited minor changes from map to map, these nodes store...
Figure 3: The memory savings of the presented methods over time. The total number of nodes for storing all data is approximately linear with time. By implementing RLE, this number grows more slowly. Implementing re-alignment further decreases this requirement by removing some noise before RLE.
Figure 5: (a) The second map in the set of 66. (b) The third map in the set of 66. Note that the PR-2 encountered another robot in the hallway that was not there in the second map. Also note the matching noise patterns in the two maps; this is because these images show the re-aligned maps. (c) A hard difference between the original maps. Not also the high degree of noise in the background. (d) A hard difference between the re-aligned maps. The robot remains intact, while much of the noise has been eliminated. Note that, interestingly, the shadow of the robot appears, because it shifted the color of the voxels such that they no longer were sufficiently matched to the surrounding averaged wall.
Figure 6: (a) The first map in the set of 66, showing the general shape of the offices. (b) A visualization of voxels that changed occupancy state more than 6 times over the course of 66 maps. Note that the vast majority of the map was mostly static. The areas of high change rates are doors, a frequently visited room, a room in which tables and chairs shifted often, and one hallway that was subject to abnormally high localization error.

mostly redundant data. Applying the RLE encoding scheme described in section 2.3 reduces this figure to about 14 million nodes, about an 85% reduction. After storing the initial map, each additional map adds only an average of about 187,000 nodes.

While we have saved a lot of space by simply applying RLE, we are still storing noise along with data regarding real changes to the map, which is what we really care about. The next step was to apply the noise reduction algorithm described in section 2.2. This further reduced the total number of nodes required to store the data to about 8.6 million, with on average 105,000 nodes being added per map. This is an interesting result; if we assumed that the difference could be entirely accounted for by noise, that would mean that almost 44% of the new voxels added per map using RLE were noise. Of course, this assumption is likely not completely valid and remains to be empirically tested. However, upon visual inspection of the results, most of the changes filtered out
appear to be noise. A demonstration of one such visual inspection is given in Figure 5.

All results presented in this section were run using the re-alignment parameters $\alpha = 2$ and $\omega_d = \omega_c = \omega_p = 1$. Visualizations were made using a slightly modified version of octovis, a visualization tool for the OctoMap library[24].

3 Potential Applications

There are a variety of potential applications of the methods introduced in section 2. In this section, we will present a number of these applications, and directions in which related work could be taken in the future. We also present map differencing methods, and an algorithm for finding segments in sparse octrees, which is a useful counterpart to map differencing.

3.1 Differencing Methods

Given the continuous-valued and probabilistic nature of the occupancy information stored in our octree data structure, there are many methods of determining the difference between two maps, each of which may be more suitable than others for certain applications. We present here a few of the methods used in the course of our research that have proved useful in reasoning about how a map has changed.

The input to each of the following methods is two octrees representing the same environment at different times. Note that both input trees could actually be stored as one tree as described in section 2.3, but we will treat them as if they are separate octrees for simplicity.
3.1.1 Hard Differencing

‘Hard’ differencing begins with binning each voxel in both maps such that it takes only one value in the state space \( V = \{ \text{occupied, free, unknown} \} \). An value of unknown as assigned to any unobserved node as defined in 2.1. For all known nodes \( n_i \), a value of occupied is assigned if \( P(n_i) > 0.5 \), otherwise free is assigned.

After binning, an output octree is constructed by generating a structure with an output node \( n_{i,\text{out}} \) for each pair of corresponding input nodes \( n_{i,1} \) and \( n_{i,2} \) that represent the same spatial location in their respective trees. Then \( n_{i,\text{out}} \) takes on the value of \( n_{i,2} \) if \( n_{i,1} \) is occupied and \( n_{i,2} \) is free or vice versa. If either input is unknown or if both have the same state, the corresponding output node \( n_{i,\text{out}} \) takes on the unknown state. Thus the resulting output tree represents all the changes that would need to make to tree \( t_1 \) in order for it to have occupancy values matching \( t_2 \), although the exact probabilities of corresponding nodes may still vary.

3.1.2 Threshold Differencing

Threshold differencing is very similar to hard differencing, but attempts to avoid reporting erroneous changes when there is high uncertainty about the occupancy of both voxels, i.e. \( P(n_{i,1}) \) and \( P(n_{i,2}) \) are both close to 0.5. There are actually two variants of threshold differencing that were investigated. The first is to establish a threshold \( \alpha \) on the difference between the occupancy probabilities. Then \( n_{i,\text{out}} \) takes on the value of \( n_{i,2} \) when both input nodes are known and the following criterion is satisfied:

\[
|P(n_{i,1}) - P(n_{i,2})| > \alpha
\]
In all other cases, $n_{i,\text{out}}$ takes on the unknown state as before. The other variant is to establish a sort of ‘gray’ area near the middle of the probability space. It is nearly identical to hard differencing, except that binning is done by assigning $v_i = \text{occupied}$ when $P(n_i) > 0.5 + \alpha$, $v_i = \text{free}$ when $P(n_i) < 0.5 - \alpha$, and $v_i = \text{unknown}$ otherwise.

### 3.1.3 Continuous Differencing

Continuous differencing, unlike the previously discussed differencing methods, does not make binary decisions about whether or not a corresponding pair of voxels are different enough to be considered as changed, but rather encodes the degree to which each corresponding pair of voxels has changed in another parallel tree. While this is certainly produces a more descriptive output, it has a much bigger memory and processing footprint than the binary decision methods, which, when applied to environments such as the one we tested that is mostly static, produce sparse output octrees. A continuous difference will return a tree at least as big as the smallest input tree. It is created by simply setting output node $n_{i,\text{out}}$ a continuous value as defined by:

$$v_{i,\text{out}} = |P(n_{i,1}) - P(n_{i,2})|$$

This output tree can then be used to reason about the probability that the part of the environment represented by each voxel has changed.
3.2 An Efficient Method for Object Segmentation in Map Differences

Map differencing operations have a tendency to produce octrees characterized by sparse clusters of known volumes. In the case of hard differencing methods that make binary decisions about whether voxels are different, this happens naturally as a result of the fact that the majority of volume in an indoor environment does not change between passes. In the case of probabilistic differencing, a sparse, clustered octree can be extracted from the result by placing a threshold on the difference in the occupancy probability for a pair of voxels to be considered different.

Algorithm 1 BuildBoundingBox($n$)

```plaintext
if $n$ is leaf then
    if $n$ is known then
        return BoundingBox($N$)
    else
        return BoundingBox($\emptyset$)
    end if
else
    for $i \leftarrow 1$ to 8 do
        $b_i \leftarrow$ BuildBoundingBox(Child($N$, $i$))
    end for
    for $i \leftarrow 1$ to 8 do
        for $j \leftarrow i + 1$ to 8 do
            if $b_i$ borders $b_j$ on faces $m$, $n$ then
                MATCHFACES(FACE($b_i$, $m$), FACE($b_j$, $n$))
            end if
        end for
    end for
    return BoundingBox($b_1$, $b_2$, $b_3$, $b_4$, $b_5$, $b_6$, $b_7$, $b_8$)
```

18
While a human viewing a visualization of a sparse, clustered octree might be able to easily determine which sets of voxels represent contiguous volumes, the octree structure itself does not actually lend itself to a trivial determination of contiguous volumes. This is due to the fact that the lowest shared parent of two contiguous voxels could be anywhere between the immediate parent of both voxels and the root of the tree. We now present an efficient, bottom-up recursive method for extracting contiguous volumes from an octree is presented.

Algorithm 2 \texttt{MatchFaces}(f_1, f_2)

\begin{verbatim}
if $f_1$ points to volume \textbf{then}
  if $f_2$ points to volume \textbf{then}
    join \texttt{VOLUME}(f_2) and \texttt{VOLUME}(f_1)
  else if $f_2$ is not a leaf \textbf{then}
    for $i \leftarrow 1$ to 8 \textbf{do}
      \texttt{MatchFaces}(f_1, \texttt{Child}(f_2, i))
    end for
  end if
end if
else if $f_2$ points to volume and $f_1$ is not a leaf \textbf{then}
  for $i \leftarrow 1$ to 8 \textbf{do}
    \texttt{MatchFaceNode}(\texttt{Child}(f_1, i), f_2)
  end for
else if $f_1$ and $f_2$ are not leaves \textbf{then}
  for $i \leftarrow 1$ to 8 \textbf{do}
    \texttt{MatchFaceNode}(\texttt{Child}(f_1, i), \texttt{Child}(f_2, i))
  end for
end if
\end{verbatim}

The algorithm is based on a primitive object referred to in the pseudocode as a BoundingBox. A BoundingBox \(b\), as it’s name implies, represents the six faces that bound a certain volume at any level of the octree from which contiguous volumes are being extracted. Each BoundingBox \(b\) stores six quadtrees representing the faces,
\{f_0, f_1, f_2, f_3, f_4, f_5\}, as well as a set \(V = \{v_0, v_1, \ldots, v_i\}\) of contiguous volumes that are entirely enclosed by \(b\) (i.e., there is no node \(n\) that is a part of contiguous volume \(v\) that appears on any face \(f\)). The quadtrees store, at the leaf level, pointers to contiguous volumes that are not fully enclosed, and which therefore may be joined with contiguous volumes in neighboring BoundingBoxes. Contiguous volumes are simply stored using separate octrees. Therefore, an input octree \(t\) that has \(N\) contiguous volumes will produce output set \(\{t_0, t_1, \ldots, t_N\}\), where each output tree is of the same size and rooted at the same coordinates as the \(t\), allowing contiguous volumes to be easily located in the original tree.

The algorithm first proceeds recursively to the leaf level of the input octree, and trivial BoundingBoxes are created with six single-node quadtrees with null pointers if the leaf node in the input octree is unknown, or a copy of the leaf node if it is known. Bigger BoundingBoxes for intermediate nodes are then constructed by combining the eight BoundingBoxes enclosing the volumes of their children. This is done by first extending each contiguous volume octree rooted at \(n_i\) that came from child \(c\) by creating a new root node \(n_r\) and setting \(n_i\) as child \(c\) of the \(n_r\).

We then proceed by comparing the quadtrees representing all of the pairs of opposing faces that will be interior to the new, larger BoundingBox. If a match is found, the octrees are combined via ‘grafting’; the choice of the word grafting here is meaningful, because there may be more than one contiguous volume from any single child that are connected via other BoundingBoxes rather than internally. It is therefore important that such volumes do are not mutually overwritten in the combined octree, but rather the sum of the information in each octree is recorded.
Eventually, the recursion ends with a BoundingBox that represents the bounds of the entire input octree. The result will then be all of the contiguous volumes completely enclosed by this root-level BoundingBox and all of the contiguous volumes referenced by the faces of the BoundingBox. At this point, all of the octrees storing the contiguous volumes will have grown to the size of the input octree, and the location of the known voxels will exactly mirror their locations in the input, making it easy to find the contiguous volumes in the original tree.

The algorithm described in this section was conceived, as noted earlier, for use with sparse, clustered octrees such as those generated by a map differencing operation. However, it is also readily extendable to many other clustering operations on octrees. First of all, the algorithm as written here finds contiguous volumes where ‘contiguous’ is defined by the fact that each voxel in the volume shares at least one face with at least one other voxel in the volume. This could be expanded to include voxels that only share an edge, or even just a vertex, by having BoundingBoxes store binary trees indicating which contiguous volumes are bordering each edge, as well as pointers indicating which contiguous volumes may be bordering each vertex. Further extension could be achieved by redefining the method by which voxels are matched as used by the MATCHFACES method; here, the trivial case in which voxels are matched if they are both known is used. However, one could conceivably want to cluster based on color, in which case voxels would be combined into a contiguous volume if and only if they were physically neighbors and also within a predefined color distance. This type of cluster extraction would then make sense on much more dense octrees, and could potentially be used for static object segmentation by color in octree maps.
3.3 Future Work

There are a number of directions that this work could be taken. First of all, it would be interesting to see how pre-processing steps on the raw point cloud data, or even on the RGB-D image sequence, prior to octree construction would affect the resulting four-dimensional tree. One might adopt methods similar to KinectFusion or RGB-D mapping to do point cloud alignment before the steps described in section 2.1. However, while our data sets are subject to substantial error in local alignment, the global registration is relatively accurate and consistent across all maps. In order to make inferences about a time series of maps constructed using methods as in KinectFusion and RGB-D mapping, not only must all loops be closed, but they must be closed in the same way for every instance. If the global alignment gets too far off, the noise reduction methods of section 2.2 likely will not be able to help.

Another line of work would be to empirically test the benefit of having a map with history as opposed to a map with only the most recent state of the environment. For example, the structure described in section 2.3 has information about how often each voxel changed readily available, simply by counting the number of map entries for that voxel. This could be applied to a navigation algorithm, allowing a mobile robot to plan paths that avoid areas that were most recently observed to be free, but are either frequently changing occupancy states or simply frequently occupied. Similarly, storing history would enable robotic systems to learn how the mapped environment changes with time. Learning how the color of voxels changes throughout the course of a typical day could be useful for applications that use visual features that are dependent on lighting conditions, as colors could be predicted based on the time of day. There could also be
security applications in which mobile robots patrol a building, making a new map on each pass. The system could determine, based on history, which of the observed changes were low-probability events (i.e., they do not fit observed change patterns), which might correspond to the theft or the introduction unknown objects into the environment.

References


