Mapping

McMaster University

Hamilton, Canada

Author

T. Akshay

Supervisor

Dr. Shahin Sirouspour

July 26, 2019

Contents

List of Figures

List of Tables

Depth Alignment

1 Introduction

The D435 depth camera is an active depth sensor using an infrared emitter, two infrared receivers, and an RGB camera. The centre of the RGB image thus obtained may be different than the depth image. To correct this, an extrinsic correction is applied to the depth image to transform the depth image to the RGB co-ordinates. This process is typically done in code by an aligning function on the CPU, which sequentially applies this transformation to every pixel in the depth image. This causes a significant reduction in the performance of the code. So, an GPU version of the same aligning function can be implemented to resolve this issue.

This process is comprised of three parts: de-project from depth pixel co-ordinates to depth $3-D$ co-ordinates, transform from depth $3-D$ co-ordinates to RGB $3-D$ co-ordinates, and project from RGB $3-D$ co-ordinates to RGB pixel co-ordinates.

2 Example

The program align.cu is an example code on how to implement such a function on GPU. Other functions such as filtering can also be implemented on the GPU in a similar fashion.

Mapping

1 Introduction

For autonomous navigation, obstacle avoidance and mapping are two very important problems. To achieve this the current state of the robot and information of the environment must be known.

To retrieve the state of the robot, the T265 tracking camera is used and the D435 depth camera is used to get the 3-D information of the environment locally. Since information from the depth camera is local to the robot, it needs to be transformed into a global co-ordinate system consistent with the state information from the tracking camera. This transformed map then needs to be stitched together with the pre-existing global map to create a new updated global map of the environment.
This information can then be used by the system to test whether a certain region of the space is occupied. Specifically, the type of the map required is an occupancy grid, which divides the space into regions of fixed shape and holds the information on the occupancy of grid.

The primary goal of this library is to create and update a consistent global map of the environment given the current position and orientation, and the local map of the environment. The update of the global map should happen in real-time as the information is used to avoid obstacles and plan the trajectory of the robot.

2 Problem Statement

Given the current state of the T265 tracking camera $\mathbf{x}$, and a set of points as viewed by D435 depth camera $\mathbf{L} = \{l_1,\ l_2,\ ...,\ l_n\}$ with respect to the D435 co-ordinate frame $\mathfrak{R}_{D435}$, generate and update a global map $\mathbf{G} = \{g_1,\ g_2,\ ...,\ g_n\}$ with respect to a global reference frame $\mathfrak{R}_G$.

3 Algorithm

3.1 Introduction

The data from the D435 is a $16-bit$ depth image from which the 3-D co-ordinates of each pixel in the depth image can be found out using the camera intrinsic data. In the most simplistic approach, the points from the depth camera at time i, $\mathbf{L}_i$, can be transformed from $\mathfrak{R}_{D435}$ to $\mathfrak{R}_G$ using the state information provided by the tracking camera and the relative position and orientation of the depth camera with respect to the tracking camera. The new global map $\mathbf{G}_{i+1} = \mathbf{G}_i \bigcup \{\mathbf{T}_{D435-G} \circ \mathbf{L}_i\}$. So, at each update step the number of points in the global map increases by $|L|$.

Also, since an occupancy grid is required to check whether a region is occupied or not, $\mathcal{O}(n)$ number of calculations would have to be done to determine the occupancy, where n is the total number of points in the map, if the points are stored randomly. Since n scales linearly with time, it will become increasingly hard to test regions for occupancy. So, some type of ordering is required. Eg: A simple sort of the array of points $\mathbf{G}$, would bring down the time of query to $\mathcal{O}(\log n)$. But still after every insertion, the time taken to sort the array would be $\mathcal{O}(n)$.

The number of points in the depth image $= 640 \times 480 = 307200$.

An accurate representation of each point would require at least 48 *bits* of memory.

So, the total memory consumed per update step would roughly be equal to 14 MB. This means that this approach is not scalable because the size of global map would quickly overflow the heap limit.
Also, in such an algorithm two points close to each other would be regarded separate and would individually consume memory. So, there must also be a step to merge points together into a single point if they are close enough.

So, the following points are important for any algorithm implementing such a global occupancy grid:

- There must be some sort of ordering among the points.
- There must be a step to merge points close to each other.

Two possible implementations of such an algorithm will be discussed:

a. **3-D array implementation**

   For a given space of fixed size ($= M$) in the form of a cube, the space is divided into smaller equally sized cubes of size m. For each point in the $\mathbf{G}$, the cube in which they lie is termed as occupied. If two points lie within the same smallest cube, they are merged together. Multiple such bigger cubes of size M can be used to fill up the entire space. This is illustrated in the figure below.

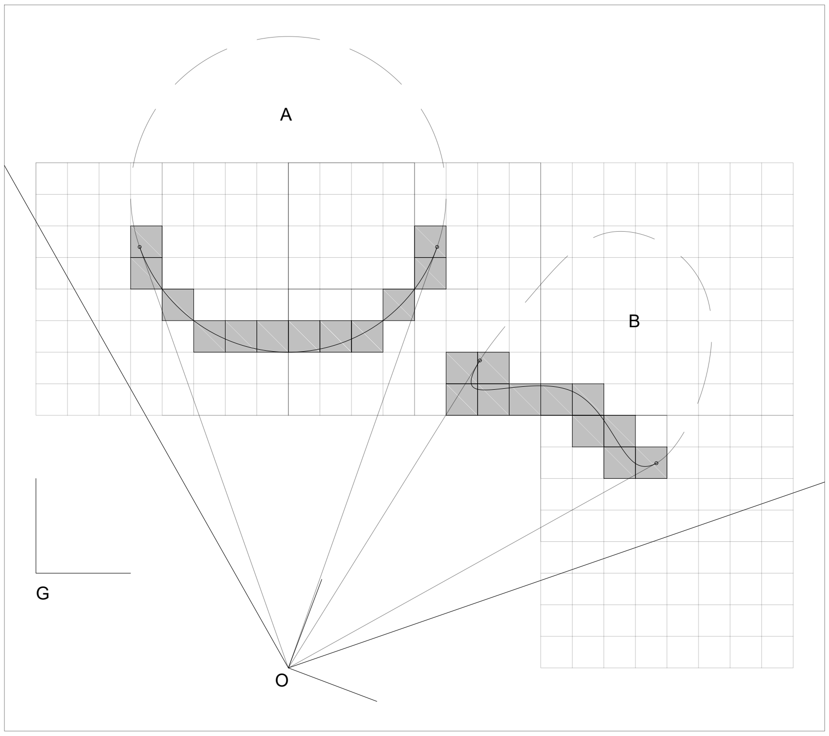


Figure 1: 3-D Array Implementation

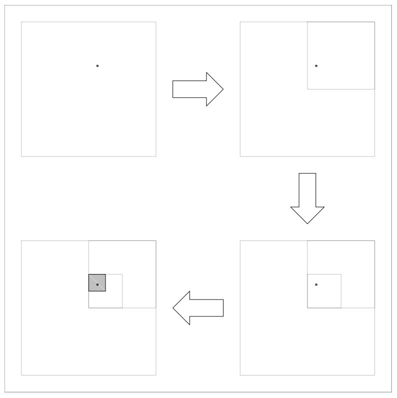
The characteristics of such a method are described in the table below.

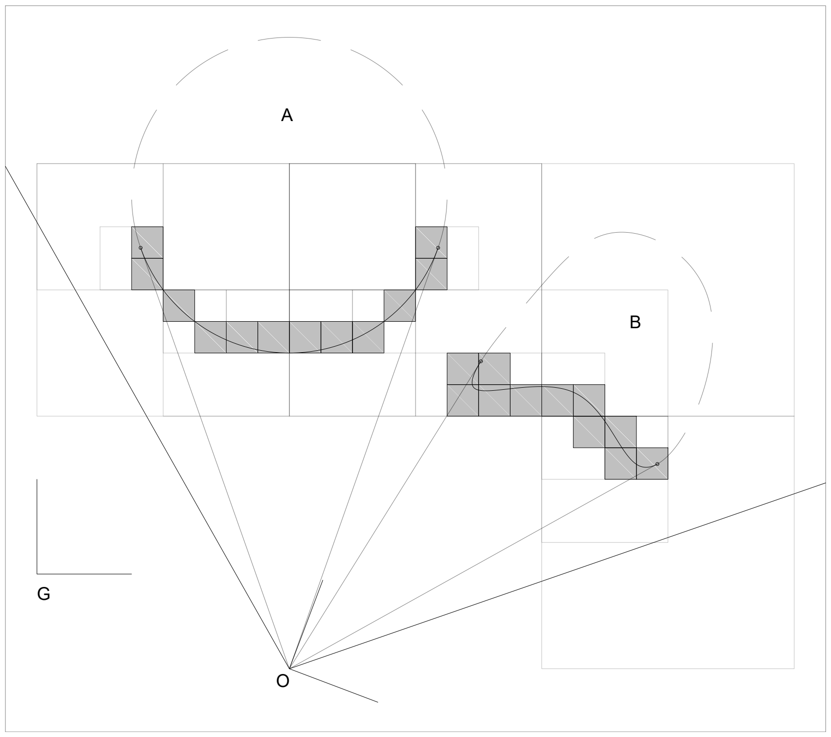
	Operations		Memory
	Insertion	Look-Up	
Time	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(L^3)$

Table 1: 3-D Array Implementation

b. **Tree implementation**

For a given space of fixed size ($= M$) in the form of a cube, the cube is divided into 8 equal parts and each of these smaller cubes are divided and so on until the smallest cube has size m. For each point in $\mathbf{G}$, the bigger cube is divided into 8 parts, but memory is allotted to only the part in which the cube lies. This is recursively followed until the smallest cube is reached. If two points lie within the same smallest cube, they are merged together. As in the previous case, multiple such cubes of size M can be used to fill up the entire space. This is illustrated in the figure below.


(a) Structure for a Single Point



(b) Example Implementation

Figure 2: Tree Implementation

The characteristics of such a method are described in the table below.

	Operations		Memory
	Insertion	Look-Up	
Time	$\mathcal{O}(\log \frac{M}{m})$	$\mathcal{O}(\log \frac{M}{m})$	$\mathcal{O}(n)$

Table 2: Tree Implementation

The memory consumed by the array implementation is proportional to the total volume of the smaller cubes inside the larger cube and is a constant, whereas the memory consumed by the tree implementation is proportional to number of points observed. But the array implementation can access and update points instantly, whereas time required for the tree implementation would be proportional to the depth of the tree.
Since memory consumed volumetrically would be larger than that consumed per the surface area, the second method is implemented in this library. Note that it is not guaranteed that the total memory consumed in the second method will be less than the first method. If the surface being observed has fine features spanning the whole volume, the first method will consume less memory and will be better overall; but since most surfaces observed have larger features, the second method was preferred. The memory consumed also strongly depends on the values m and M.
In both of these methods the bigger cubes of size M are arranged in a tree to facilitate look up. The arrangement of the larger cubes is independent of the arrangement inside the larger cubes.

Implementing such a tree structure has the following advantages:

a. Consumes lesser memory in general than an array implementation.

b. The time for any look up scales poorly with time.

c. Since a tree structure is used, if a large region is empty the look up time reduces. This structure can also be used to dynamically free memory by combining the information of its children nodes, at the expense of resolution of the region.

Another way to reduce memory consumption with the tree structure is to remove the intermediate nodes altogether and directly store the map as a tree of leaf nodes. For the current implementation this can be achieved by the tuning the parameters of the code.

3.2 Algorithm

3.2.1 Terminology

Leaf The smallest cube in the tree structure. They have an edge length $=\ m$. The leaf node contains the information about the co-ordinate of the point inside it along with its variance.

Voxel Short for volumetric pixel. These are the intermediate nodes in the tree which can be further divided into smaller voxels. (or leaves depending on the depth in the tree). The largest voxels are termed root voxels. The voxel also stores information about a point, which is the combined.

The origin of the leaves and the voxels is considered to be the vertex with the smallest co-ordinate in all dimensions. So, if the origin of the voxel is (x_o, y_o, z_o) and the edge length is k, the vertices of the voxel are $\{(x_o, y_o, z_o),\ (x_o + k, y_o, z_o),\ ...,\ (x_o + k, y_o + k, z_o + k)\}$.

Grid It is a tree which contains all the root voxels. The ordering is based on an index calculated from the origin of the voxels.

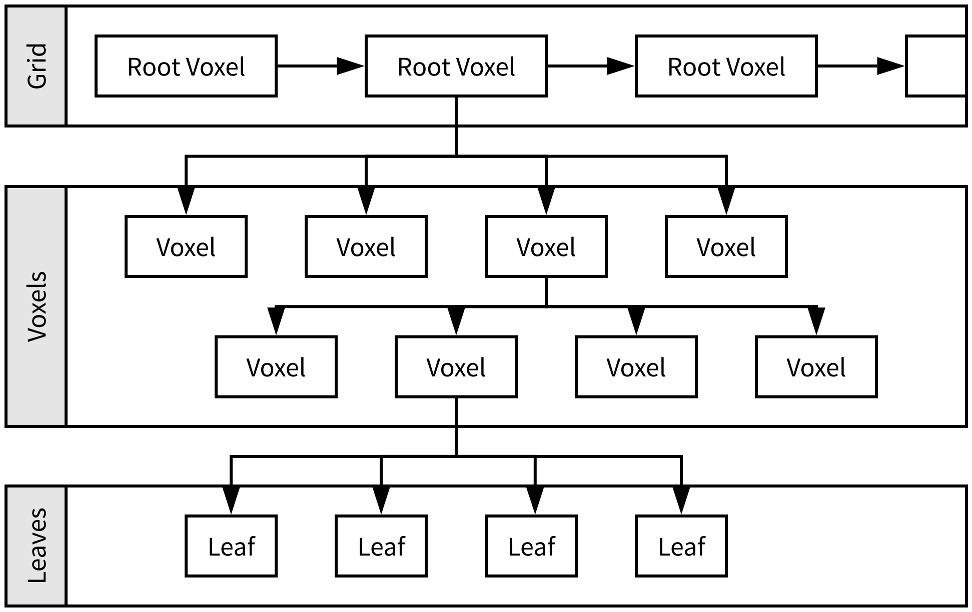


Figure 3: Octree Structure

3.2.2 Description

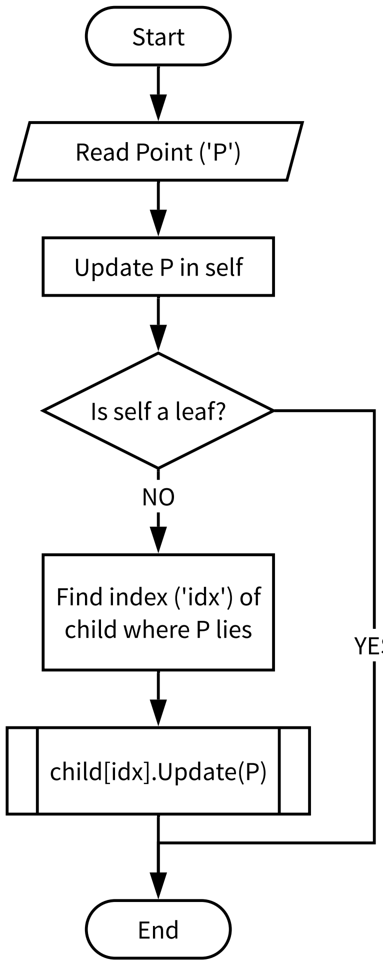
For each pixel in the depth image the 3-D co-ordinates of the point are extracted in D435 frame ($\mathfrak{R}_{D435}$). This is then transformed into the global frame ($\mathfrak{R}_G$) using the pose information from the T265 and the extrinsic information. Then it is checked if the point belongs inside any root voxel in Grid. If the root voxel doesn't exist, it is created, and inserted into Grid. If it exists, the octant in which the point exists inside the root voxel is evaluated. Then it is again checked if that child voxel exists and the above steps are recursively followed until a leaf node is reached.

3.2.3 Pseudocode

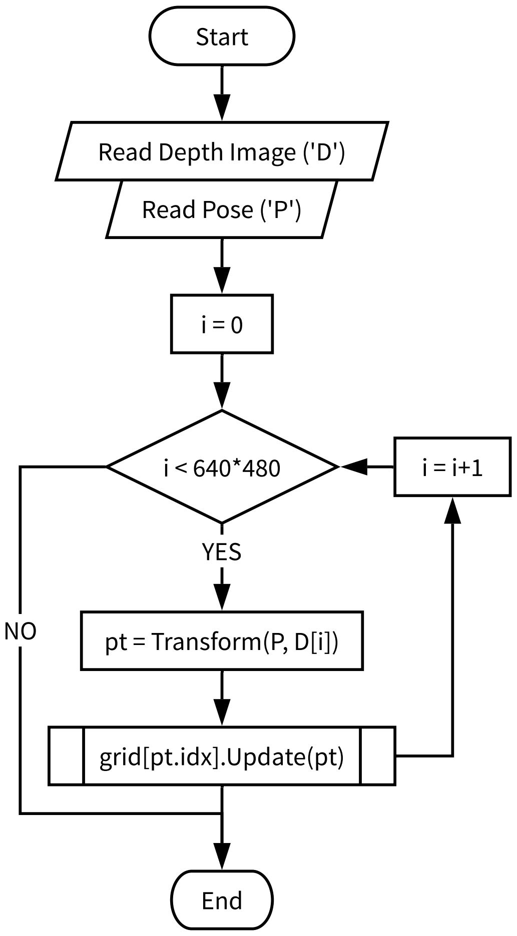
Algorithm 1 Update

```
Require: $depth[][] \neq NULL \wedge pose \neq NULL$
Ensure: $\mathbf{G}_{i+1} = \mathbf{G}_i \bigcup \{\mathbf{T}_{D435-G} \circ \mathbf{L}_i\}$
 1: for all $depth[i][j] \; != \; 0$ do
 2:     $p \leftarrow deproject(depth[i][j])$
 3:     $p \leftarrow \mathbf{T}_{D435-G} \circ p$
 4:     $node \leftarrow binary\_search(\mathbf{G}, \; index(p))$
 5:     procedure UPDATE($node, \; p$)                    ▷ Update node with point
 6:         $node.point \leftarrow p$
 7:         if $node = leaf$ then
 8:             go to 1
 9:         end if
10:         $idx \leftarrow index\_of\_child(node, p)$
11:         if $node.size \leq 2 \times m$ then                       ▷ node is a leaf
12:             if $node.child[idx] = NULL$ then
13:                 $node.child[idx] \leftarrow new(leaf)$
14:             end if
15:             Update($node.child[idx], \; p$)
16:         else
17:             if $node.child[idx] = NULL$ then                    ▷ node is a voxel
18:                 $node.child[idx] \leftarrow new(voxel)$
19:             end if
20:             Update($node.child[idx], \; p$)
21:         end if
22:     end procedure
23: end for
```

3.2.4 Flowchart



(a) Update Method



(b) Map Update in CPU

Figure 4: Global Map Update in CPU

3.3 GPU Algorithm

3.3.1 Description

The algorithm for updating the octree structure is the same, but since multiple threads operate concurrently on the same structure, synchronization methods must be applied to ensure consistent operation.

The update of any particular voxel must happen atomically. Mutex locks can be used to enable this. However, using mutex locks has two disadvantages:

a. Since the number of voxels changes dynamically, the mutex locks must also be declared dynamically. This can pose some memory issues. On the other hand, if a few statically declared mutex locks are used at some higher level of the tree, it would limit the parallel efficiency of the GPU operation.

b. According to CUDA forums, the mutex lock is unreliable between threads. So, mutual exclusivity may not be guaranteed.

Atomic functions are used to overcome these problems. Atomic functions are operations which occur atomically on the GPU. One such function is the atomic Compare and Swap function.

$$int\ ret \;=\; AtomicCAS(int\ *\ add,\ int\ comp,\ int\ val)$$

This function compares the value *old*, at the address *add*, to the value *comp*. If $old = comp$, the value *old* at the address *add* is replaced with *val*, and *val* is returned. Else, the value at *add* is changed and *old* is returned. This operation is guaranteed to occur atomically, meaning only one thread can access the value at *add* at any given point. This function is sufficient to ensure consistency of the tree structure. For example, the update rule in the most basic case is:

$$x_{k+1} \;=\; \frac{v_p \times x_k + v_k \times p}{v_p + v_k}$$

$$v_{k+1} \;=\; \frac{v_k \times v_p}{v_k + v_p}$$

These two equations are coupled and therefore need to be done in a single step. However, if they are converted to the following form:

$$\frac{x_{k+1}}{v_{k+1}} = \frac{x_k}{v_k} + \frac{x_p}{v_p}$$

$$\frac{1}{v_{k+1}} = \frac{1}{v_k} + \frac{1}{v_p}$$

The equations are decoupled, and the Compare and Swap function can be applied to this equation. This is reason for storing $\frac{x_k}{v_k}$ and $\frac{1}{v_k}$ as the parameters instead of x_k, and v_k. Similarly, for the $3-D$ gaussian case, the update rule is:

$$V_{k+1}^{-1}x_{k+1} = V_k^{-1}x_k + V_p^{-1}x_p$$

$$V_{k+1}^{-1} = V_k^{-1} + V_p^{-1}$$

Similar to the $1-D$ case, the transformed variables should be stored instead of the original ones.

To avoid the problem of multiple threads creating the same instance of voxel, $atomicCAS()$ is performed on the pointers to the voxel.

3.3.2 Pseudocode

Algorithm 2 GPU Update

```
Require: $depth[][] \neq NULL \wedge pose \neq NULL$
Ensure: $\mathbf{G}_{i+1} = \mathbf{G}_i \bigcup \{\mathbf{T}_{D435-G} \circ \mathbf{L}_i\}$
 1: for all $threads$ do
 2:     $p \leftarrow deproject(depth[BID][TID])$
 3:     $p \leftarrow \mathbf{T}_{D435-G} \circ p$
 4:     $node \leftarrow binary\_search(\mathbf{G}, index(p))$
 5:     procedure UPDATE($node$, $p$)                    ▷ Update node with point
 6:         $atomicAdd(\&node.point, p)$
 7:         if $node = leaf$ then
 8:             go to 29
 9:         end if
10:         $idx \leftarrow index\_of\_child(node, p)$
11:         if $node.size \leq 2 \times m$ then                       ▷ node is a leaf
12:             if $node.child[idx] = NULL$ then
13:                 $t \leftarrow new(leaf)$
14:                 if $atomicCAS(\&node.child[idx], NULL, t) \neq NULL$ then
15:                     delete($t$)
16:                 end if
17:             end if
18:             Update($node.child[idx]$, $p$)
19:         else
20:             if $node.child[idx] = NULL$ then                     ▷ node is a voxel
21:                 $t \leftarrow new(voxel)$
22:                 if $atomicCAS(\&node.child[idx], NULL, t) \neq NULL$ then
23:                     delete($t$)
24:                 end if
25:             end if
26:             Update($node.child[idx]$, $p$)
27:         end if
28:     end procedure
29: end for
```

3.3.3 Flowchart

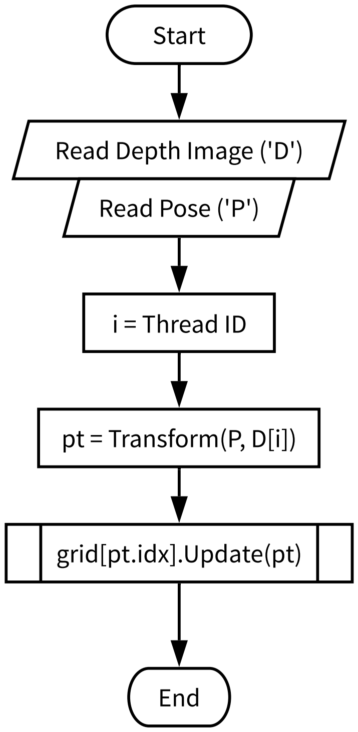


Figure 5: Global Map Update in GPU

4 User Interface

4.1 Map

Some functions are implemented to help the user of the package to access the map.

Point() This function returns the occupancy of input point, ie. it returns a Boolean value stating if the input point is occupied or not. This enables the map to be used as an occupancy grid.

AllPoints() This function returns all the points stored in the map. This function uses a single thread to read all the points in the map in a non-sequential manner. This function can affect the performance of the package if called at every update iteration.

4.2 Logging

Various types of logging mechanisms have been provided to aid debugging. Data can also be displayed in real-time.

Depth Video Logging This type of logging can be used to store the depth images from the D435 as a gray-scale video. However, since $C++$ does not support $16-bit$ gray-scale videos, the video is compressed to 8 $bits$.

Pose Logging The six degree of freedom pose information from the T265 is logged in a *.csv* file along with the timestamp of each measurement.

Depth Intrinsics Logging The parameters intrinsic to the depth camera D435 are logged in a *.csv* file.

Map Logging The full map is logged as a collection of points in a .csv file.

Depth Video Display This can be used to view the depth image received from the depth camera.

$3-D$ **Depth Display** This is a $3-D$ representation of the depth image from the depth camera.

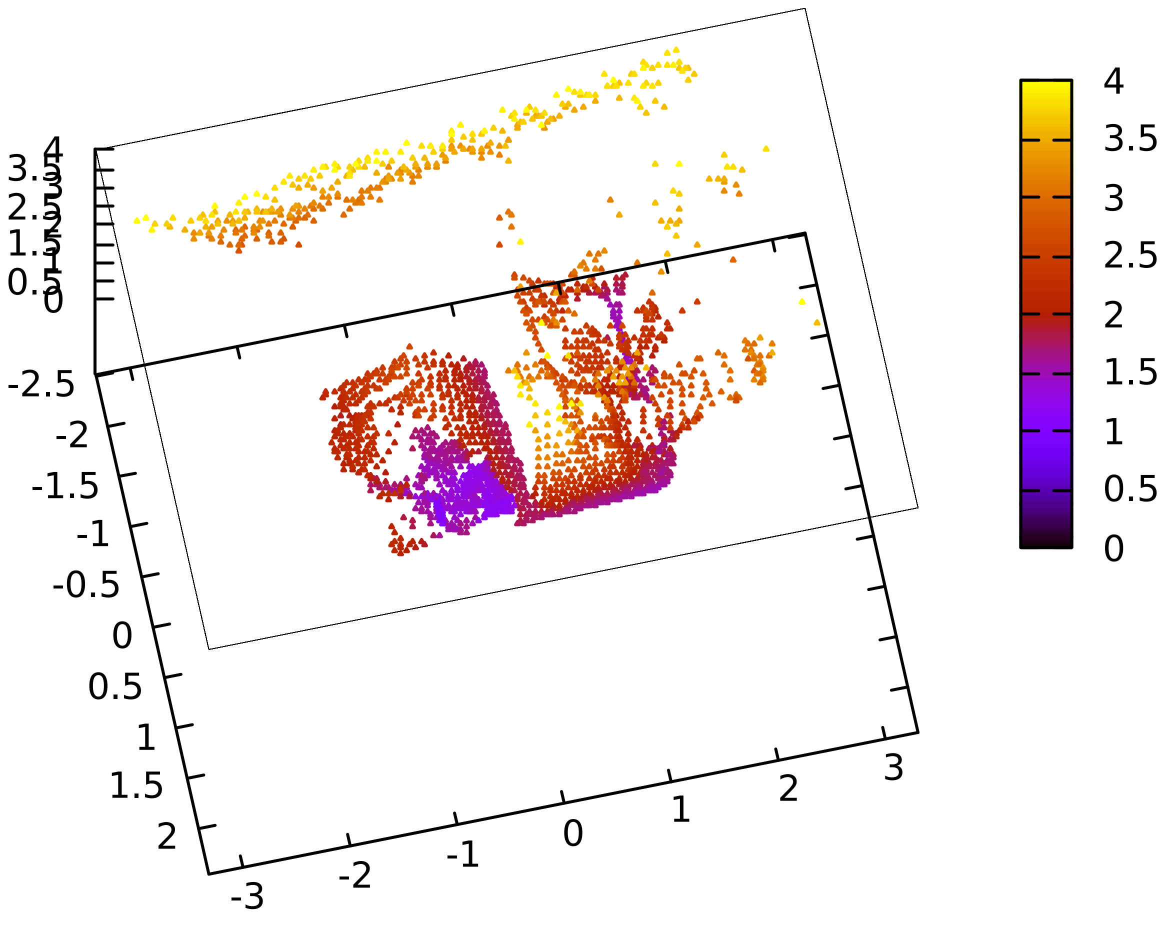


Figure 6: Local Map from D435

4.3 Camera Publishers

Since any camera attached on the computer can only be accessed by a single process at any given point in time, a separate process can be used to publish the data from the two cameras into a shared memory buffer. This can be later accessed from any other process using the shared buffer.

To avoid inconsistent reads and writes from the shared buffer, synchronization mechanisms should be applied while reading or writing data. In the class provided a shared named semaphore is used to synchronize the reader processes and the writer processes.

In addition to mutual exclusivity, some other features are also desirable. One such feature is that the shared memory should be inaccessible to the writer process when some other process is reading and vice-versa, but multiple readers should be able to concurrently read from the same buffer. Another feature is that if there is a reader process currently using the shared memory and the writer process and a few other reader processes are waiting, the preference should be given to the writer process. This will help ensure a more uniform rate of update of the data.

This is the typical reader-writer problem with writer preference.

4.4 Performance

The module was tested on Dell Inspiron 7559, with an Intel core *i7-6700HQ CPU @ 2.6 GHz*, and 16 *GB* RAM running Ubuntu 16.04. The GPU used was NVIDIA GeForce GTX 960M.

Depth range: 0.11 m $-$ 4 m

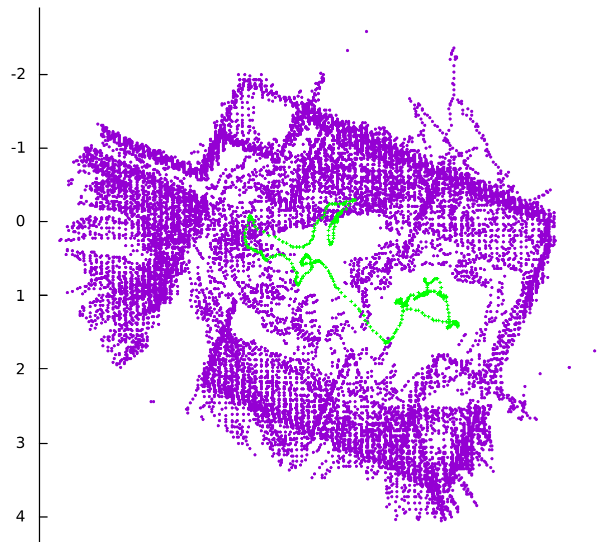
Leaf node size: 2 cm

Update time for the map (depends on the complexity of the environment):

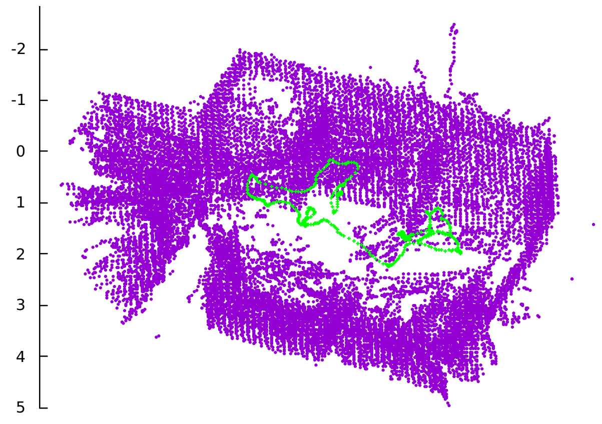
- **CPU** : $100ms\ -\ 2s$
- **GPU** : $10ms\ -\ 100ms$

The update time increased by about 1.5 times when the leaf node size is decreased to 1 cm. As expected, the update time is found to increase logarithmically with the number of levels in the tree, and therefore logarithmically with the inverse of the leaf node size.

The update time doesn't depend strongly on the depth range, since only the number of points updated from the depth image matter, which is limited to $307,200$ points per update.



(a) Plan View



(b) Isometric View

Figure 7: Global Map Example

Note: All the performance values were measured in absence of logging and display functions. Real-time logging and display will affect the time taken for each loop, and therefore the overall performance.

5 Future Work

Few different lines of approaches for future work on the package have been listed in this section.

5.1 Camera Parameters and Post Processing

Testing of D435 reveals that the raw image from D435 is prone to some errors. These errors include spurious data points, large uncertainties in depth values, and range limited to 4 m. Parameters such as disparity limit on features in stereo images, exposure, and power of infrared emitter can be tuned to achieve optimum results. Post processing techniques like spatial filters, and temporal filters can be used to increase reliability of data. Other image processing methods like removing connected components less than a threshold value can be applied to avoid false positives.

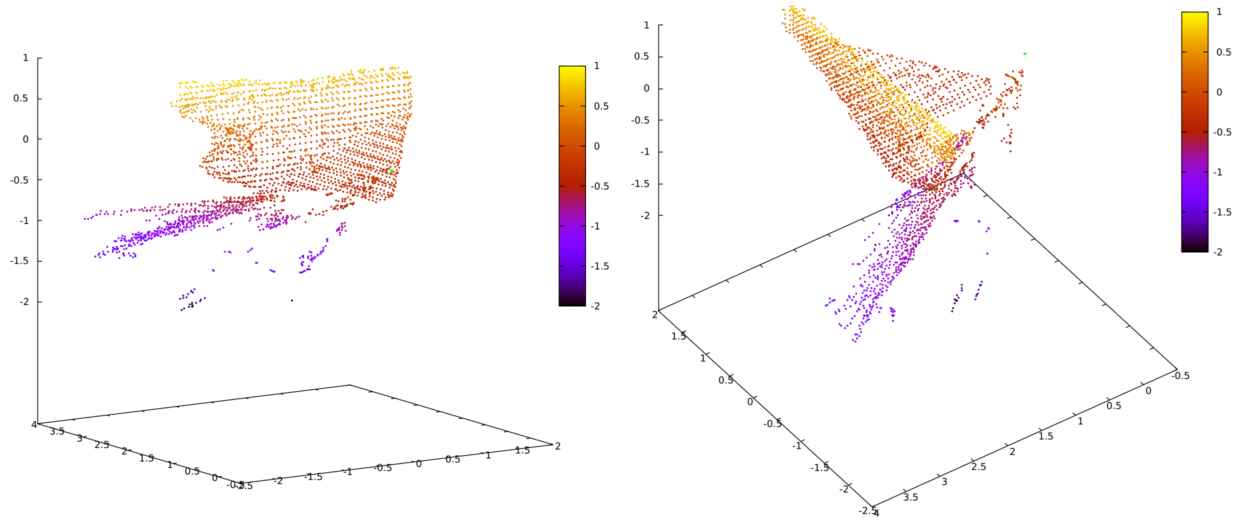


Figure 8: Spurious Data from D435

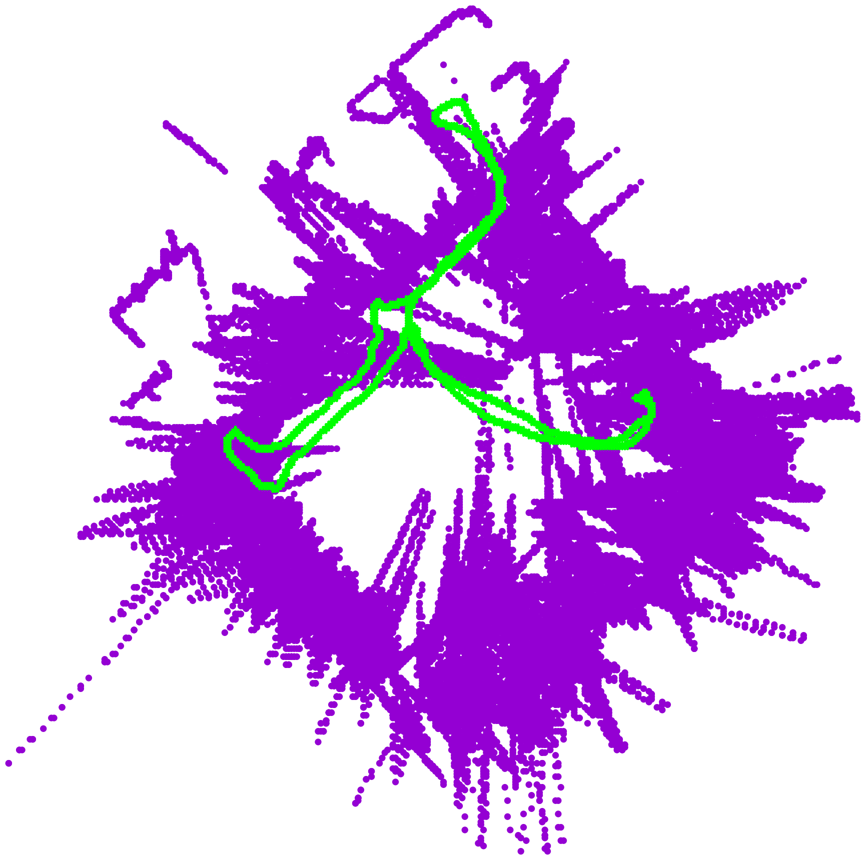


Figure 9: Global Map Example with erroneous Data Points

5.2 Memory Management

Currently each leaf node occupies a space of 16 *bytes* and each voxel node consumes 96 *bytes* in the memory. There is an additional overhead when memory is allocated using *malloc*() function call. A major contribution to the large space occupied by the voxel node is the pointer array to the children of the voxel. The pointers are 8 *bytes* ($64 - bit$ architectures) in size, and consume a total of 64 *bytes* per child. Custom memory management techniques can come handy in optimizing the memory used. A few very effective alternatives are listed below.

- *malloc*() function call is preferable when the memory to be allocated is of an unknown size. However, since the memory solely consumed by each struct is known a priori, a slab allocator would eliminate external fragmentation issues. A custom slab allocator can be designed by allocating pages from memory and dividing it into fixed size chunks, separately for leaf and voxel nodes. Another benefit from this is that pointer sizes can be reduced from 8 *bytes* to 4 *bytes*, and a mapping can then be defined from the stored pointer to the virtual address space pointer. Further according to information required, redundant variables such as size of a voxel, or the variance of the point can be removed.

- Another interesting mechanic to keeping the memory bounded can be 'forgetting', as inspired from humans. The amount of memory can be calculated at any point, and if the total memory consumed by the map reaches a threshold limit, parts of the map visited the farthest back in time can slowly be removed from memory. Since each parent voxel contains the combined point from each of its children, the children can be destroyed at the expense of reducing the accuracy by half. Owing to the tree structure used, since the number of leaf nodes would be of the order of the total number of voxels, such a method would be very effective. Different heuristics on which nodes and how many of them to be destroyed can be tuned.

- The memory efficiency of the octree structure compared to an array implementation is strongly dependent on the surface being observed (see Section 3.1). A dynamic combination of the two implementations could be maintained. In such an implementation, any given space would start out as an octree structure and would switch to an array structure when the memory consumed by the tree exceeds that which would have been consumed by an equivalent array structure.

5.3 Extrinsic Calibration

The extrinsic calibration data from the T265 tracking camera to the D435 depth camera currently being used is from the CAD model of the camera mount. The calibration data can be obtained independently using the data received from the cameras. The pose information combined with pose estimation from D435 (for instance using *ArUco* markers) can be solved simultaneously at each point, and the least square fit from the translation and rotation at each time instant can be used to obtain any desirable accuracy of the extrinsic parameters.

5.4 Front-End Functions

The current iteration of the package provides only two front-end functions to access the global map. Everything else is abstracted from the user. Additional front-end methods could be supplied from the back-end, optimized depending on the purpose of use. One such function could be returning an occupancy grid as an array of Boolean values, taking the size of the grid as the input. This can be achieved using the front-end methods currently provided, but such an implementation would not be optimal.

5.5 SLAM

The ‘confidence’ of the pose information returned from the T265 is a function of the lighting conditions, exposures, the local environment, etc. The confidence is expressed as an integer between 1 and 3. On testing it was found that the loop closure property of the T265 is prone to vibrations, and its accuracy was found to vary with the circumference of the loop.

Since the global map is stored and updated, SLAM can be performed using the depth camera whenever the confidence of the T265 stoops down to a low value. SLAM would be performed only in the event of a low confidence level or sometimes to check loop closure. An important pre-requisite to performing SLAM calculations would be the observation probability function. This function is a probability distribution function which reports the probability of an observation given the map as a function of the position of the robot. One simple method to calculate the probability for a given position of the robot, would be from a simple correlation between the observed map and the global map. To increase the efficiency, other hash functions could also be defined. The optimum solution would then be searched in a domain of states of the robot. An appropriate optimization method would have to be selected for this.

The co-variance of the position returned from T265 could be known confidence levels, and this could be used to define the domain of search.

Sparse methods such as recognising landmarks in the image and correlating them with landmarks, either stored in memory or deduced from the map, could also be used instead of the dense SLAM method.

5.6 Mapping Free Spaces

The depth camera returns a depth image of size 640×480, of which each pixel contains the distance from the camera baseline to the point, which is along the line joining the pixel coordinates at the projected plane and the origin of the camera. Information can be deduced from this image and the pose in addition to the trivial data available. Any point in the depth image is visible to the camera only because the space directly in front of the camera and point is empty. This can also be incorporated into the global map.

This can be useful in distinguishing areas previously visited or observed from those not. It also removes the ambiguity in a map which only contains a binary state for each grid cell. The trivial approach to this problem would be to set each node between the obstacles and the camera origin to be empty. However, directly applying this method can pose some issues:

- Any node in front of an obstacle point need not be only dependent on the obstacle point, but also on other nearby obstacle points. That is to say the emptiness of a node in front of an obstacle point is not well-defined since the node is a finite volume as opposed to ta single point. Different parts of the node may fall in the line of sight of different pixels, and therefore to declare any node as empty all the pixel which could potentially occlude the node must be checked.

- The is method is not scalable with smaller size of nodes. Since each node has to be manually swept through, the time taken to complete this for one update step would be proportional to the number of nodes in the observed region, which is proportional to the volume. Also, many steps may be redundant during this method considering that if all children of a voxel are empty, the voxel is empty and each of its children need not be set as empty individually.

A potentially faster algorithm has been proposed and described in the subsequent sections.

5.6.1 High-Level Overview

The main aim of this algorithm would be to update all the appropriate nodes as empty while eliminating the fallbacks previously listed.

A node X is defined to be dependent on a node Y, if there exist two points P_1 and P_2 in X and Y respectively, such that P_2 lies in the line segment P_1O (O is the origin of the camera). This can be rewritten as: X is dependent on Y if Y lies in the line of sight of X from O. The volume of view is defined as the volume visible from camera at the current time instance.

This is a polyhedron defined by the four planes, each of which is a limit on the angle of view of camera, and the plane which limits the maximum distance measured. This forms a pyramid.

The fundamental principle governing this algorithm is that a voxel is empty if and only if it completely lies inside the volume of view and does not depend on any occupied node. The algorithm would start with the root voxels and any voxel would be split into its children only if the voxel is not fully empty. This would avoid some redundant calculations because all the leaf nodes are not checked individually. Moreover, it is the expected time for an update step would only be proportional to surface area, because nodes would be more finely fragmented only along the boundaries of the volume of view and not in the bulk.

Some theoretical results on these dependency graphs have been proved to build up to the algorithm. All subsequent discussions are done equivalently for a $2-D$ space; digressions to $3-D$ space will be made if necessary.

5.6.2 Preliminaries

As described earlier, a node is dependent on another if the latter lies in the line of sight of the former. This dependency is denoted by an arrow from the latter node to the former dependent node. Nodes which don't have any dependencies will be referred to as sources, and those without any node dependent on them as sinks.
Also, the domain enclosed inside a node is bounded only on two edges and on one vertex, such that every point in space belongs only to one node.
Note: The claims made in the following sections may not be true depending on the boundary cases for the domain of a node. Boundary cases should always be checked separately.

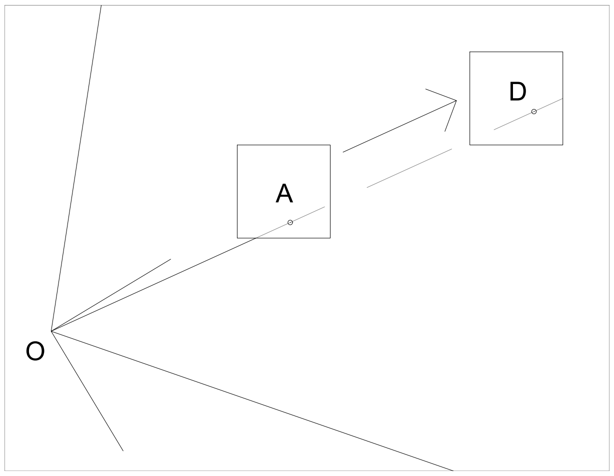


Figure 10: Basic Dependency Example

Lemma 1

Any convex set of nodes fully containing the camera origin O will not depend on any other node outside the set.

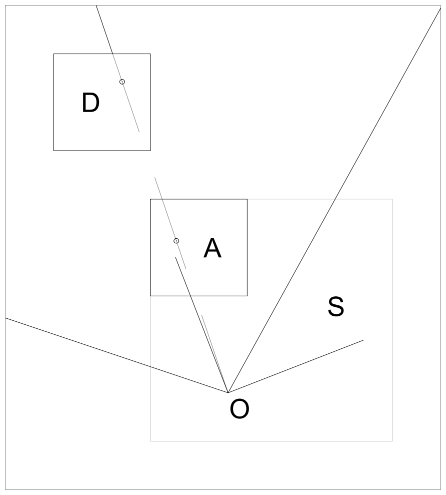


Figure 11: Lemma 1

Lemma 2

If a node depends on some other node, it also depends on at least one of its neighbours. Each node not containing the camera origin will depend on at least one of its neighbours.

Conversely, if a node depends on another node, it also depends on at least one of latter's neighbours.

Note: Check the boundary conditions for the domain of a node.

This is true because a node is completely enclosed by its neighbours.

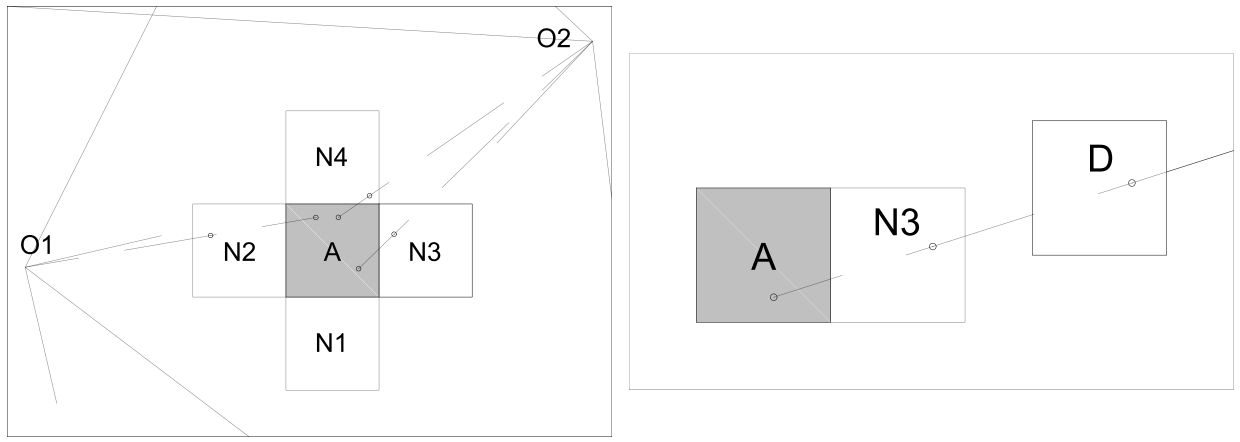


Figure 12: Lemma 2

The information of dependence of a node on its neighbours can be used to deduce the possible locations of the camera origin. The dependence of a node on a particular neighbour reduces the possible locations of the camera origin to a half space with the common face between the neighbours as the separating plane.

Lemma 3

If the node depends on more than one neighbours, the possible camera origins can be found using the intersection of such half spaces. Conversely, it also follows that if the node depends on a node located on its left half space, the node also depends on its left neighbour. (The above statement is weak, since stronger statements about the position of the camera origin)

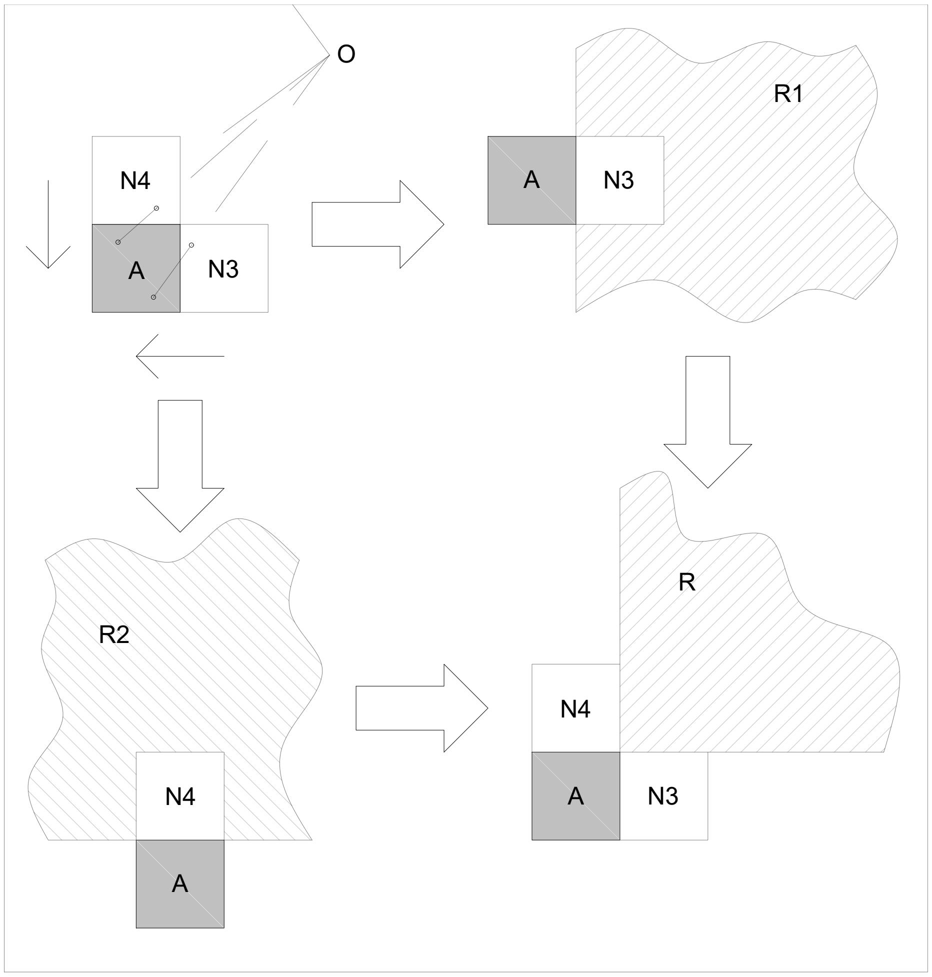


Figure 13: Lemma 3

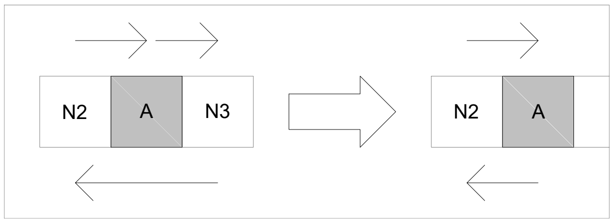
Next a convex set of nodes enclosing the camera origin will be considered. This assumption can later be relaxed. The nodes in this set do not depend on any other nodes outside the set; meaning a finite directed graph **G** can be constructed from the dependency relations in the set.

Lemma 4

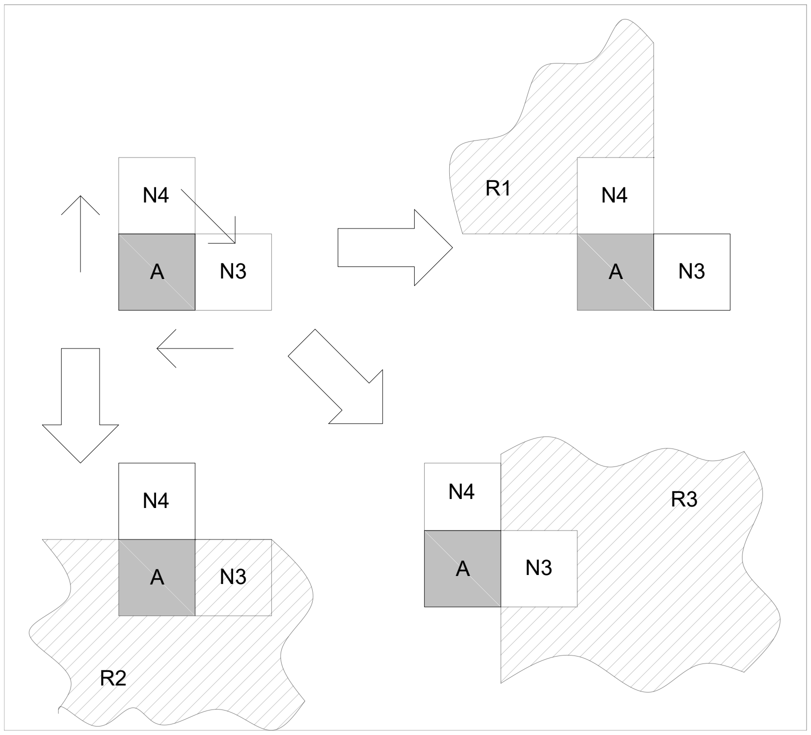
The graph **G** has exactly one source, namely the node containing the camera origin. Every node in the space depends on this node, and conversely this node doesn't depend on any other node.

Also note that dependence in this graph is not transitive; meaning if A depends on B, and B depends on C, A need not depend on C.

It is clear that two neighbouring nodes cannot depend on each other, because each dependence would generate a separate half space of possible locations for the camera origin, and these two spaces do not intersect. It can also be further seen that three neighbouring nodes cannot depend on each other in a cyclic fashion. This can again be seen by finding the intersection of the half spaces, which turns out to be empty.



(a) Lemma 4: Case 1



(b) Lemma 4: Case 1

Figure 14: Lemma 4

Theorem 1

There cannot exist any loop in a dependency graph **G**.

Note: This is only true if there is a single camera and the nodes themselves are convex. Even in these cases, cyclic dependency cannot be ruled out. However, in the special case where the nodes are squares (or cubes in $3-D$) and arranged in a grid, there is no cyclic dependency. An example for cyclic dependency is shown below.

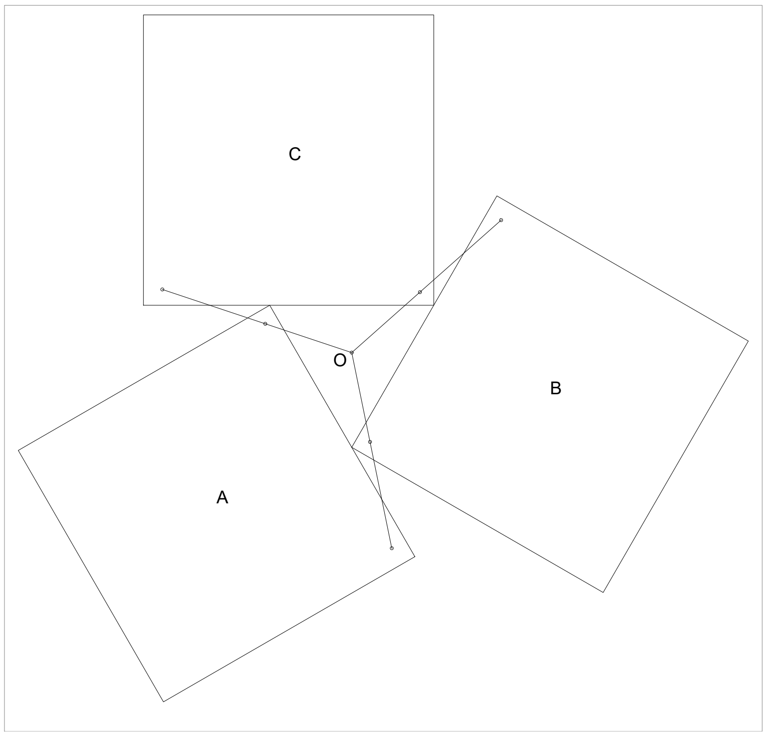


Figure 15: Cyclic Dependency Example

Proof:

This can be proved by contradiction. Assume there exist a set of nodes $\{N_1, N_2, ..., N_m\}$, such that there is a cyclic dependence among the nodes. The nodes are arranged in a sorted order based on the abscissas of the origins of the nodes. If the abscissas of all the nodes are same, they are sorted using the ordinates. Both of them cannot hold simultaneously for distinct nodes. Let the node with the maximum index in this sorted list be N_k. So N_{k-1} depends on N_k, and N_k depends on N_{k+1}. By the converse statement in Lemma 3, N_{k-1}

depending on N_k would imply that the camera origin lies on the open half space right to the origin of N_k. Similarly, N_k depending on N_{k+1} would imply that the camera origin lies on the open half space left to the origin of N_{k+1}. Since N_k has the maximum abscissa of all the origins in the set, the intersection of these two spaces is a null set, yielding a contradiction. Thus, $\mathbf{G}$ can have no loops. $\square$

Corollary 1

$\mathbf{G}$ always has at least one source and at least one sink.

Note: The above theorem would work irrespective of whether the nodes in $\mathbf{G}$ contain the camera origin.

This property directly holds in a directed graph if it has no loops.

The sources in the original graph $\mathbf{G}$ are defined to be of *Level* 0. If all the *Level* 0 sources are removed, the resulting graph $\mathbf{G}_1$, which also has no loops, would again have some sources. These are called *Level* 1 sources and so on until the graph has no nodes. In this way the graph $\mathbf{G}$ is divided into different hierarchies such that nodes in a higher level depend only on the nodes of a lower level.

The set of nodes in a *Level* k are defined as L_k. The union of all the sets of nodes from *Level* 0 to *Level* k, including *Level* k, is defined as U_k.

Lemma 5

Let the set of all the neighbours of the nodes in U_k, which don't belong to U_k, be defined as X_{k+1}. Then L_{k+1} is a subset of X_{k+1}.

This follows trivially from the converse statement in Lemma 2. All the nodes depending on nodes in U_k would also depend on at least one of their neighbours (not belonging in U_k). It follows that all the sources in G_{k+1} belong to X_{k+1}.

Theorem 2

The converse of Lemma 5 is also true, ie. X_{k+1} is a subset of L_{k+1}. So, $X_{k+1} = L_{k+1}$.

Proof:

The proof is by induction. The initial source is known to be the node containing the camera origin from Lemma 1. This node is also the sole element in L_0.

Step 1: There are four elements in the set X_1, which are the four neighbours of the node in L_0. It can be easily checked that none of the elements in X_1 depend on each other or on any node in $\mathbf{G} \setminus \{U_0 \bigcup X_1\}$. Since the nodes in X_1 only depend on L_0, $X_1 = L_1$.

Step k: The theorem is assumed true for nodes up to *Level* k. Now it only needs to be shown that the nodes in X_{k+1} do not depend on the nodes $\mathbf{G} \setminus \{U_k \bigcup X_{k+1}\}$. In that case all the nodes in X_k would be sources in the graph $\mathbf{G}_{k+1}$ and therefore X_{k+1} would be equal to L_{k+1}.
This statement can be proved by contradiction. From figure 16, assume that the node A belongs to X_{k+1} and the node N has *Level* $\geq k+1$ and that A depends on N. From Lemma 2, A also depends on one of its neighbours, D. Also, by definition of X_{k+1}, one of the neighbours of A, B, is of *Level* $\leq k$. The Level of B cannot be $< k$, since then A would have to be of *Level* k from the assumption of *Step* k, and would violate the definition of X_{k+1}. So, B is of *Level* k. From Lemma 3, there is neighbour of B other than A, C, such that B depends on C. This is using the converse of the argument used to show that A has a neighbour D. Then the level of C would be $k-1$, and the level of D, which is a neighbour of C, would be k. But this is a contradiction, because D, which is of *Level* k, depends on N, which is of *Level* $\geq k+1$.
The above induction step has only been proved for the non-trivial case. It can also easily be checked for the trivial cases.

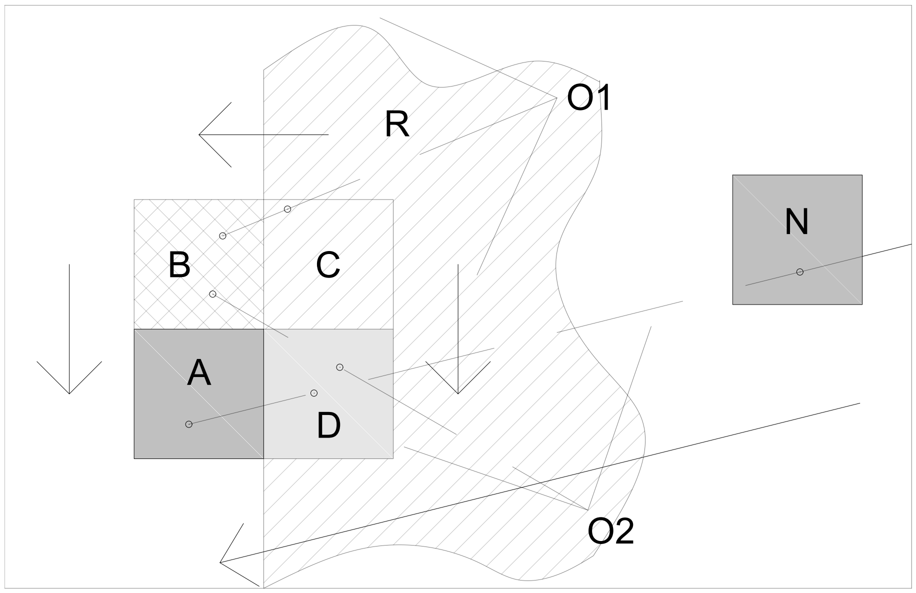


Figure 16: Theorem 2

By induction it will follow that $X_k = L_k$ for all k. $\square$

Figure 17 is an example of how these levels can be found out using Theorem 2. In the figure the coloured squares are the current level, and the shaded squares are the next level for each step in the process.

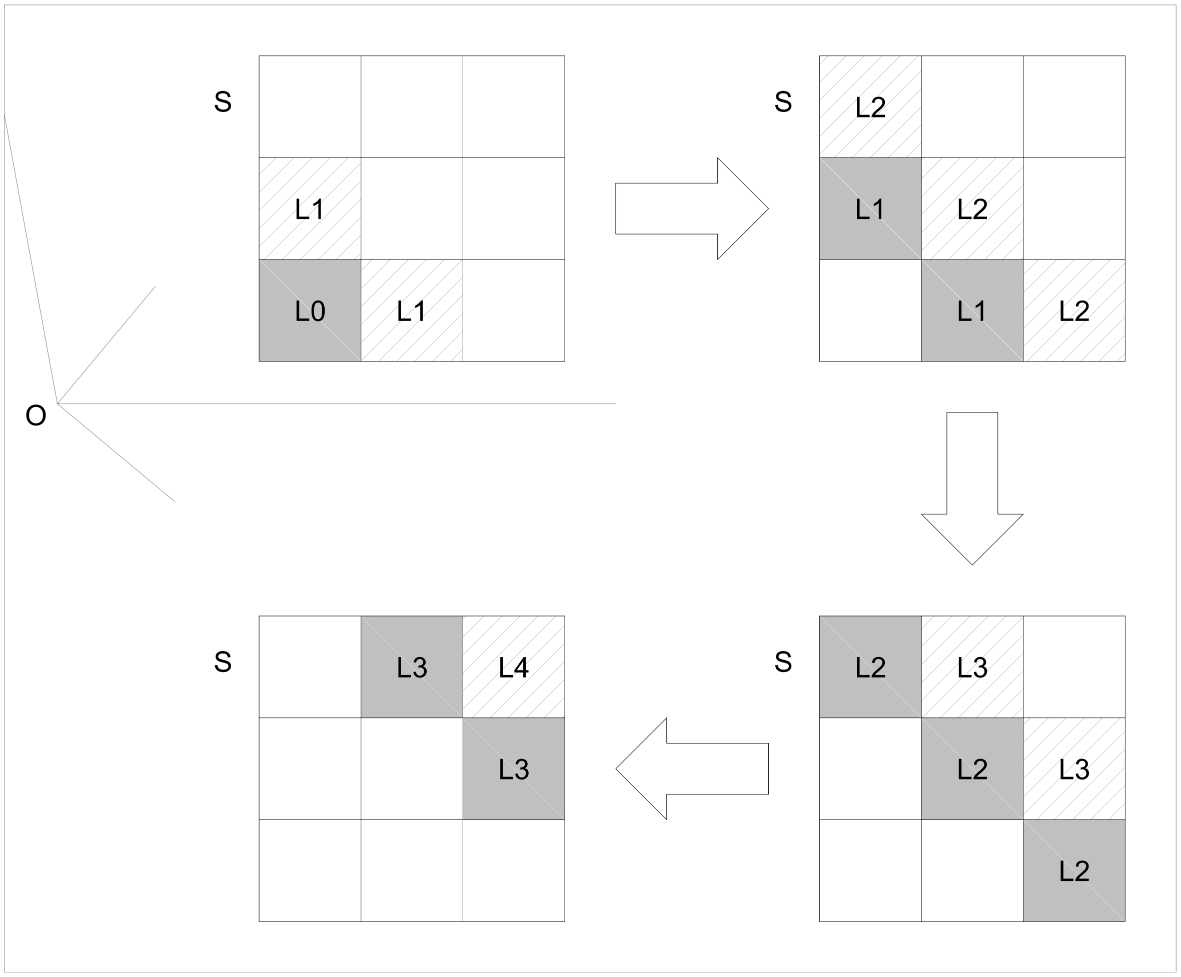


Figure 17: Dependency Graph Example

5.6.3 Algorithm

The algorithm aims to fill up the space in front of the camera with the information about whether the node is occupied or empty or if it has not been observed yet. The occupied nodes are separately entered at the usual update algorithm as mentioned in Section 3.2. If a voxel has entirely not been observed or is fully empty, no children are assigned to the voxel and the voxel is set as unobserved or empty respectively. However, if the voxel is only partially occupied or empty, children and assigned to the voxel and this process recursively continues until leaf nodes are reached. The voxel will have been said to be updated when this property is satisfied.

The algorithm starts with the root voxel containing the origin of the camera, which like all other voxels has eight children. In this set of eight child nodes, the node at dependency *Level* 0, ie. the child node containing the origin of camera is found. The algorithm updates this voxel and then moves on to the voxel nodes at the next level inside the set of eight nodes of the root voxel. Every time the algorithm moves to the nodes at the next level, the nodes at the previous level are passed to them. If a voxel is empty it can directly and completely be seen from the camera, whereas if a voxel is unobserved, it is either completely behind an obstacle or completely outside the view of the camera. Throughout the algorithm if any voxel is unobserved due to it being occluded by obstacles, it carries information on which nodes occlude it. The recursive update algorithm is described below:

Given all the nodes at a particular *Level* n and the nodes at its parent *Level* $n-1$, the update starts at any node of *Level* n and iterates through all the nodes in that level before moving on to the next level.

For any node at *Level* n, first it is checked whether the node is completely inside the view of the camera.

- If the node is completely outside the view of the camera, the node is assigned to be unobserved, if it has also not been observed over the entire trajectory of the camera.
- If the node is completely inside the view of the camera and all the nodes on which the node depends are also empty, the node is assigned to be empty.
- In any other case, first it is checked if the node is a voxel node or a leaf node.
    a. If the node is a voxel, the algorithm moves down one depth to the children of the node. The node of *Level* 0 is identified amongst the eight children and the nodes

at *Level* $n-1$ which are the neighbours of this node are passed to the node for updating. After this level has been updated, the nodes of *Level* 1 are found out and they are updated and so on. After all the eight children are updated, the control returns to the calling parent node.

b. If the node is a leaf and it is already not set to be occupied, the following cases are checked and the status of the leaf node is set based on the first case which is true:

   1. If the centre of the leaf node is outside the view of the camera, it is set to be unobserved.
   2. If all the parent nodes of this node in the dependency graph $\mathbf{G}$ are empty, the leaf is set to be empty.
   3. If at least one of the parent nodes in $\mathbf{G}$ is occupied, the leaf is set to be unobserved.
   4. Else, the dependencies of the parent nodes in $\mathbf{G}$ are checked. If the centre of the leaf is occluded by some of the dependencies, the leaf is set to be unobserved. Otherwise, it is set to be empty.

Appendix

1 Code

Mapping GitHub Repository: https://github.com/AkshayThiru/Mapping

Depth Alignment: https://github.com/AkshayThiru/Mapping/blob/master/src/align.cu

2 Documentation

Web Browser: HTML Documentation

LaTeX: PDF Documentation