Global Scale Integral Volumes

Sounak Bhattacharya[1(✉)], Lixin Fan[1], Pouria Babahajiani[1], and Moncef Gabbouj[2]

[1] Nokia Technologies, Tampere, Finland
{sounak.bhattacharya.ext,lixin.fan,pouria.babahajiani.ext}@nokia.com
[2] Tampere University of Technology, Tampere, Finland
moncef.gabbouj@tut.fi

Abstract. Integral volume is an important image representation technique, which is useful in many computer vision applications. Processing integral volumes for large scale 3D datasets is challenging due to high memory requirements. The difficulties lie in efficiently computing, storing, querying and updating the integral volume values. In this work, we address the above problems and present a novel solution for processing integral volumes for large scale 3D datasets efficiently. We propose an octree-based method where the worst-case complexity for querying the integral volume of arbitrary regions is $\mathcal{O}(\log n)$, here n is the number of nodes in the octree. We evaluate our proposed method on multi-resolution LiDAR point cloud data. Our work can serve as a tool to fast extract features from large scale 3D datasets, which can be beneficial for computer vision applications.

Keywords: Integral volume · Octree · Point cloud · LiDAR

1 Introduction

Integral images are probably best known for their use in real-time face detection [1]. Since its first introduction in the '80s [2], integral images have found many important applications in computer vision research [3–6].

In general, the sum of pixels inside any rectangular shaped region in an 2D image can be calculated in constant time using its integral image representation. Integral images can be easily extended for 3D data. 3D version of integral images or integral volumes can be used to calculate sum of pixels of different cube shaped regions in constant time. However, the additional dimension poses some new challenges as huge amount of memory is required for storing the integral volumes and it also becomes computationally expensive to query and modify the integral volumes.

Computation of integral volumes of 3D datasets are difficult as all existing approaches require the data to be loaded into memory to get the integral representations. This approach becomes impossible if we consider large scale datasets: Multi resolution world map data, 3D model of a city, country or the whole world.

G. Hua and H. Jégou (Eds.): ECCV 2016 Workshops, Part I, LNCS 9913, pp. 192–204, 2016.
DOI: 10.1007/978-3-319-46604-0_14

These datasets are just too huge to load into memory. Various compression techniques [7,8] can be used to mitigate the primary memory requirement to an extent but they will not work in case of the global scale datasets mentioned before. Another challenge is updating the integral volume values. For example, if the original data changes by a single pixel, all the values of its integral representation can change. Needless to say we need efficient methods to store, query and update integral volumes for global scale datasets in order to apply many computer vision algorithms.

Hierarchical octree data structures are often used to represent these kind of large scale multi-resolution data [9]. However, there is no existing method for efficient integral volume processing for them. The problem still lies in efficiently querying and updating the integral values.

A simple approach towards this problem would be to compute and store the integral volumes locally in the leaf nodes of the tree. The integral volume value of any query region can be simply found by adding the values of all the nodes inside that region. With this design we achieve efficient storage and also deal with the updating issue. If some data changes, we only need to update the integral volume values of that specific leaf node, which saves significant computational overheads. However, this approach is very in-efficient in case of querying the integral volume value of a point or a region, as the values from the number of nodes to be accumulated to get the result may be very high. This design can be improved to work really well if the query region is aligned with the tree node-structure (see baseline method) but can be extremely slow in case of misalignment, as we show in our experiment. Thus, we need a more robust design for integral volume query while keeping the querying computation complexity low.

In this paper we present a novel approach which queries integral volumes for global scale data, and which has querying time complexity $\mathcal{O}(\log n)$, where n is the number of nodes in the octree representation of the data. We achieve significant speed gain by storing extra values in non-leaf nodes. Thus, we settle for higher offline memory usage for fast integral volume query. We experiment and demonstrate our method on large point-cloud data (Fig. 1.) and show that it is possible to fast query integral volumes from 3D global scale datasets.

2 Related Works

Integral Images and Volumes. Integral images was first introduced in 1984 by F. Crow as summed area tables which was used for texture mapping [2]. The 3D extension i.e., the idea of three-dimensional sum-tables was later given by A. Glassner [10]. In 2004, Integral Images became greatly popular by their use in Robust Real-Time Face Detection [1]. In this work a very large number of rectangular features were evaluated for each detection window and integral image representation made it possible to perform these evaluations very quickly, consequently making the system real-time in a conventional computation environment.

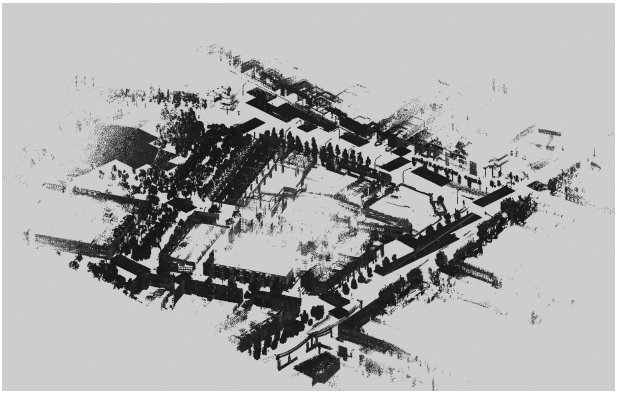

Fig. 1. Plot of a section of point cloud data at zoom 22 (maximum zoom is 23). It represents a volume of around $(612\,\mathrm{m})^3$ in about 15.2 million data points.

Despite significant speed gain while computing certain features, a common issue with using Integral Images and Volumes is that they require great amount of storage and large word length. Methods to reduce the storage size and word length has been introduced by H. Belt [8]. In that work, a lossless and a lossy method of word length reduction is introduced. The lossless method takes advantage of numerical properties of complement-coded arithmetic and the lossy method is based on rounding the original image prior to the integral image calculation. In the 3D extension, memory efficient data structure introduced by Urschler et al. [7] is very useful to reduce the memory usage when dealing with integral volumes. It works by dividing the integral volume data into equal-sized blocks and estimating the integral volume value for the points of each block using a one-parameter model and then storing the differences between the estimated values and the actual values using different word length per block. They tested their method in a weak learner ensemble based machine learning framework and the memory usage was 3 times less. However, these methods still rely on loading the input data into memory for integral volume processing. Thus, they cannot be scaled to work with global scale datasets. Also, updating the integral volume values as the dataset gets larger remains an issue with these methods.

Octrees, Point Clouds. First introduced in 1982 by D. Meagher [11] for geometric modelling, octrees have found wide range of applications. Octrees are used in efficient rendering techniques [12]. In the field of robotics, octree based 3D mapping frameworks are used [13]. Web mapping services such as Bing Maps uses 2D version of octree (quadtree) based tiling system for efficient indexing and retrieval of map data [14].

LiDAR (Light Detection and Ranging) is a great remote sensing tool to scan and record urban environments. It works by emitting beams of laser in the surrounding and recording the reflection time to create a highly detailed map. LiDAR scans usually generate large point cloud data. The octree data

structure is often a common choice for efficiently storing and processing of the point clouds generated by these kind of remote sensing technologies [9,15,16]. Also, octree based region growing algorithms have been used for segmentation of point cloud data [17].

3 Method

3.1 Octrees, Octkeys and General Offline Computation

Octrees are hierarchical data structures, where any internal node can have at most eight children. The most important property of octrees, is that they are based on the principle of recursive decomposition. For example, when representing region data, the root node of the tree represents the entire region. Each child node represents one of the eight equal cube-shaped part of the region represented by its parent node. This division of space is recursively continued until we reach a maximum depth of the tree.

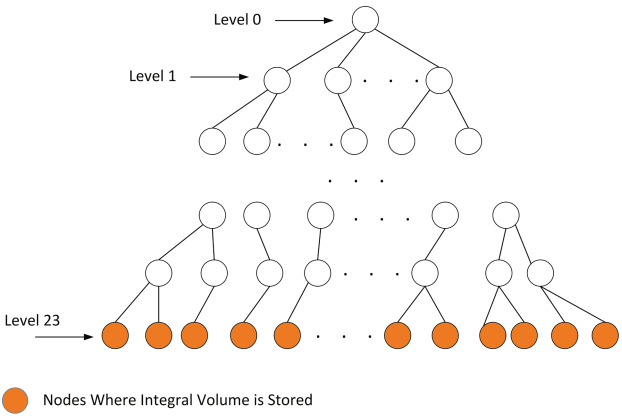

Fig. 2. Integral volumes stored at the leaf nodes of the octree

To efficiently access and visualise the data we use an octkey based tile system which is a 3D extension of the quadkey based tile system used in Bing Maps [14]. Each node in the tree is analogous to a tile at a certain zoom level and is identified by an octkey. All the tiles at a certain depth have octkeys of the same length. Also, the number of digits in the octkey is always equal to the depth of the node in the octree. In this work, we use the word tile and node interchangeably.

To achieve the task of querying the integral volume value of any point or region, we first need to compute and store the integral volume values for all the data points. As the maximum resolution data is stored at the leaf nodes, we visit all the leaf nodes of the tree and calculate the integral volume, and store it in a

file in the respective nodes. This can be visualized in Fig. 2. These computations are strictly local and are restricted to each leaf node.

We worked with 3D LiDAR scans of the city of Tampere, Finland. This data was collected by driving cars through the streets while simultaneously scanning the environment through LiDAR sensors. The range of values for the global coordinates at full resolution was $(0, 2^{31} - 1)$, as 31 bits were used to represent the coordinate values. For efficiency, the points were grouped at level 23. That means, 8 bit local displacement offset was stored in the nodes at depth 23 [15]. That's why we have an octree of height 23. The integral volumes at each leaf node are of the dimension $2^8 \times 2^8 \times 2^8$.

Now, to query the integral volume for any arbitrary region, we can add up all the tiles inside the queried region. As mentioned earlier, this is a challenge as the region gets bigger, the number of tiles to be added also gets bigger. In the worst case, if the queried region represents the root, we have to add 8^{23} tiles; Which is prohibitively inefficient. Thus, we need smarter ways to accumulate values to produce the desired result. In the next subsections we analyse different approaches to tackle this problem.

3.2 The Base-Line System

The integral volume value of a queried region can be calculated in a faster way if we can decrease the number of tiles to be added. To achieve this, we need to store values at the non-leaf nodes of the tree, otherwise we will not be able to access the integral volume values of bigger tiles without breaking it into children. Thus, we do a bottom up traversal of the tree and sum up all the children's value and store it in a file in their parents node.

Now, given a query region, if a tile partially intersects the query region, it is recursively divided into its children and tested whether the children tiles intersect the query region. A recursive branch stops if a leaf node is reached or there is no intersection of the query region with the tile. If a tile completely falls inside the queried region, its value is returned. The base line system starts with tile 0 and continues this process for tiles 1, 2 and 3.

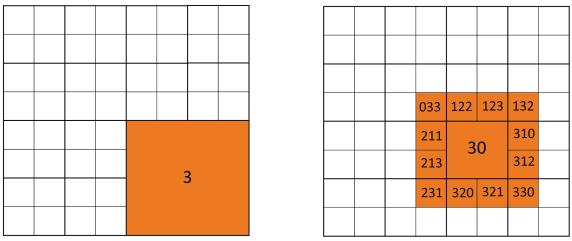

Fig. 3. Queried Region aligned with tiles (left). Queried Region not aligned with the tiles (right)

To visualise this, let's consider a simplified two-dimensional case consisting of only three zoom levels. The tiling convention is exactly same with bing map tile systems [14]. In Fig. 3, we can see the two queried regions are marked in orange. Let's consider the left image; The queried region is totally aligned with tile number 3. The tiles which actually return value are numbered in the Figure. The total number of nodes visited for querying is 4 (Tiles 0, 1, 2 and 3).

On the right image, the queried region is not very well aligned with the bigger tiles. Consequently, there are more recursive calls to break down tiles. The number of nodes visited for this query is 52 (Tiles 0, 00, 01, 02, 03, 030, 031, 032, 033, 1, 10, 11, 12, 120, 121, 122, 123, 13, ... , 333).

Evidently, there are some significant advantages as well as disadvantages of this approach. As we see from the given example, if the query region is aligned with the tile distribution, then the querying is very fast. However, if the query region is not aligned with the tiles then the system has to recurse down the tree to find smaller tiles to add. Thus there is a possibility of the number of nodes visited increasing exponentially as the query region gets larger.

3.3 Integral Volume by Storing Surfaces

We propose an alternative way, which bypasses the shortcoming of the base-line system. For each non-leaf node, we compute the integral volume locally at full resolution but only store 3 different surfaces. These are specifically the surfaces which do not touch (0, 0, 0)th coordinate of the tile. Figure 4 shows the surfaces coloured in orange.

The surface arrays are of length $2^{8+(23-\mathrm{d})}$ in both dimensions, where d is the depth of the non-leaf node. For example, at depth 22 the surface arrays are of the dimension 512×512. The reason behind the careful choice of these three particular surfaces for each tile is the design of how we intend to query the integral volume of a point or a region.

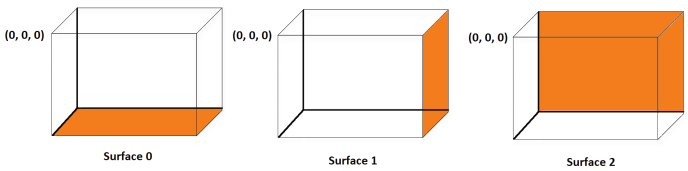

Fig. 4. Stored surfaces of a tile (Color figure online)

We calculate the integral volume value of a given query point, by starting at the root node of the tree and traversing the path towards the leaf node representing the query point. We call it the main recursion branch. For each node on this path, we calculate the projection of the query point on the relevant neighbouring sibling nodes at that particular depth. As the projected point is always on the surface of a node, we easily get the integral volume value of the

projected point from the stored surface arrays at the neighbouring nodes. We keep accumulating all the neighbour values as we repeat this process for every node on the main recursion branch. We stop the recursion at the leaf node. As there is integral volume present in the leaf node, we instantly get the integral volume value of the query point local to the leaf node. We then add this value to the accumulated neighbour values to get the desired result.

From Fig. 5 we can visualise a simple case of how the methods works. We start with a tile, which is the root node and a query point at which we intend to evaluate the integral volume value. In the figure, the root node is the cube on the top of the main recursion branch and the query point can be seen as the orange dot (a). We break down the root into children and then select the one in which the query point belongs (b). We then calculate the projection of the query point on the relevant sibling neighbours. In this case we only need to find the projection on the neighbour immediately left to the node as the starting point of the region is on the top left corner. The projected point can be seen as a black dot (c). We get the value of the projected point from the stored surfaces in the neighbour node. The surface is shown in orange colour. We then repeat this process for the selected children node. This time also, we only need to calculate the projection on the neighbour which is on the immediate left (e). We don't need to recurse down further as we have reached the maximum depth in this example (d). As the integral volume is stored here, we retrieve the integral volume value of the query point local to this tile. We then add this integral volume value with all the retrieved neighbour values to get the final result.

Now, given a cube shaped query region, we find out the integral volume value of all its corner points. Then by simple arithmetic addition and subtraction operation we can get the integral volume value of the queried region. For example, In Fig. 6. we see an cube whose corners are numbered by letters A−H. To get the integral volume for this cube, we calculate the integral volume value for each corner point using surface method. These values indicate the sum of all the point from the beginning of the global region represented by the octree to that specific point. Once we have the values for all corners A−H, we get the integral volume value by Eq. 1 shown above the figure.

$$iv_{cube} = (\mathrm{H} - \mathrm{G} - \mathrm{F} + \mathrm{E}) - (\mathrm{D} - \mathrm{C} - \mathrm{B} + \mathrm{A}) \tag{1}$$

The advantage of this approach is, for each query point we only have to go down on one particular path of the tree. Thus, if the query region has 8 corner points and the depth of the tree is d, when need to visit at most $8 \times d$ nodes in the main recursion branches. Considering the visits to the neighbours the maximum number of total nodes visited will be $k \times d$. Thus, the time complexity of this method is $\mathcal{O}(\log n)$, where n is the number of nodes in the tree. This approach is also totally independent of the alignment of the query region with the tiles. While the best case of the base-line system might be better than this method, the surface method is much better for the average case and the worst case.

This method can be further optimized if we choose the starting node to be the common ancestor node of all the 8 query points, which has the highest depth in the tree. This way we don't always have to start at the root node for small

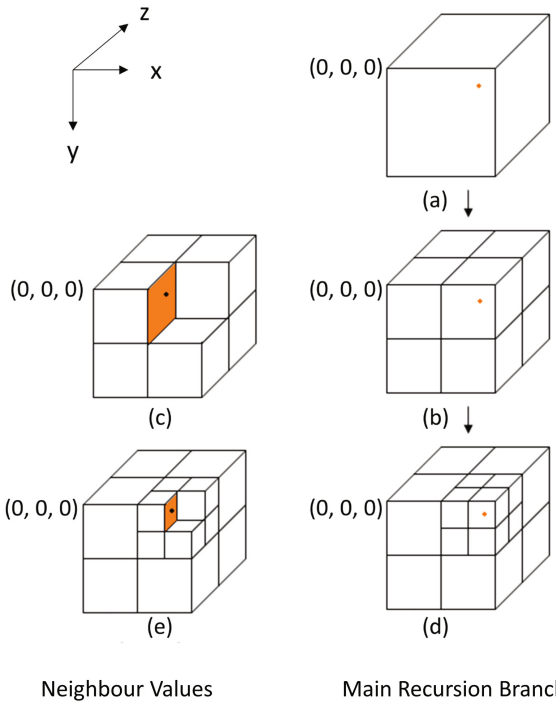

Fig. 5. Visualisation of the surface method (Color figure online)

query regions, we can start at a node much lower in the tree. We only have to start at the root node when the query region is big enough.

3.4 Integral Volume by Storing Surface Borders

The method described in the previous section works by storing surfaces at different non-leaf nodes. This approach is fast but relies on the offline storage of the surfaces at full resolution. This may be challenging as the surface size becomes significantly big, especially near the root node. For example, in our dataset we use an octree of height 23. Each surface at depth 14 is of the dimension $2^{17} \times 2^{17}$. With gzip compression this array takes around 100 Gbs of storage space. For each node we need to store 3 of them, i.e., 300 Gbs of storage per node. If we want to store them at the root, each surface would require approximately 36000 Petabytes of storage. Thus, to tackle this offline storage problem, we present another approach which works by only storing the boundary vectors of the surfaces. Previously, we were storing 3 surface arrays per node. In this approach we only store two boundary vectors for each of the surface arrays. Thus, we store 6 different vectors per non-leaf node.

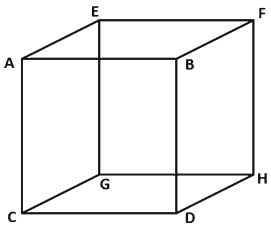

Fig. 6. Integral volume of a cube

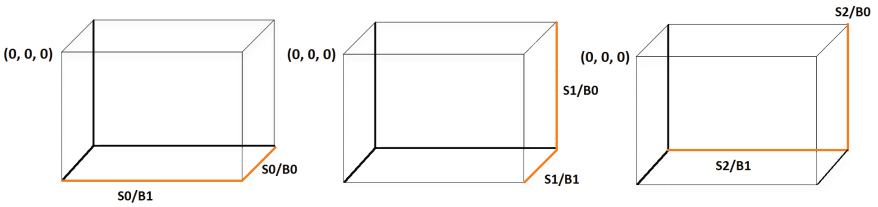

Fig. 7. Stored borders of a tile

This approach hugely reduces the off-line memory storage requirement. Instead of storing $3 \times m^2$ elements per node, where m is the width of the surface array, we now store $3 \times 2 \times m$ elements per node. Thus, the storage requirement decreases by $\frac{m}{2}$ times. This factor increases exponentially as we move up the tree. At depth 14, now we need to store $3 \times 2 \times 2^{17}$ elements per node instead of $3 \times 2^{17} \times 2^{17}$, i.e., 2^{16} times less elements. Thus, the off-line storage decreases from 300 Gbs to around 5 Megabytes per node.

However, as a consequence of not storing surfaces, now we cannot readily access neighbour values if the projection is not exactly on the border. In that case, we need to start at the root and recursively accumulate values similar to the surface method, until we reach the depth where integral volume is stored. We need to do this every time a neighbour value needs to be accessed. Thus, in this method there are two separate sets of recursion. The first set is the main branch of recursion, which starts from the root node and ends at the leaf node containing the query point; The number of recursive calls is equal to the depth of the tree. The other set of recursive calls are related to the task of accumulating each neighbour values. Obviously, this approach is computationally more expensive than the previous one, as it has to recurse down the tree every time a neighbouring value needs to be accessed. This method can be used if there is a not enough memory available to store surface arrays.

4 Experiments and Results

In this section we will discuss about the experiment and analyse the results. All the implementations were done using python 2.7 in a Linux environment (4 GB Ram, 8-cores).

4.1 Construction

On an average it took 0.4 s per leaf node to compute and store integral volumes. In the LiDAR scan of the city of Tampere, there were 509,652 leaf nodes. The total time taken was approximately 53 h. For the surface method the construction of surfaces were done by traversing the tree in bottom up direction, each depth at a time. When the surface sizes became too big, they were constructed chunk wise. The surface borders were also constructed in a similar manner. All the data were stored in hdf5 files.

4.2 Experiment Setup

We chose different sizes of query region and compared the performances of all the three methods (base-line, storing surfaces, storing surface-border). As a performance measure we chose the average nodes visited per query. The average nodes visited is a good metric of performance as it is implementation and hardware independent. For a fixed sized query region, we randomly shifted it thousands of time and recorded the average number of nodes visited for all the methods.

We use the term blocksize as a measure of the queried region size. The unit blocksize has the same dimension as the smallest tile, i.e., tiles at level 23. Block size 2 means the queried region is two unit blocks long in each dimension, and so on. The ground resolution at zoom level 23 is 0.0187 m per pixel. Each tile at zoom level 23 is of the dimension $256 \times 256 \times 256$ pixels. Thus, the unit block size physically represents a region of approximately $5\,\text{m} \times 5\,\text{m} \times 5\,\text{m}$. We repeat the experiment for increasing block sizes and observe how the methods behave.

4.3 Comparison: Average Nodes Visited

From Fig. 8 we can see that the average number of nodes visited for the base-line system increases almost exponentially as the query block size increases. This is expected as high number of nodes visited when the query regions are not aligned. The base line system has to recurse down to the leaf nodes many times. As the query region gets bigger the cost of the mis-alignment also increases.

The number of nodes visited by our proposed method (storing surfaces) stays almost constant as at most $k \times d$ number of recursive calls are required for each of the 8 query point corners. This is the key advantage as the number of nodes visited remains totally independent of the query region size.

The nodes visited in the surface-border method is much higher than the surface method as it needs to recurse down the tree to the maximum depth each time there is a neighbour call. Interestingly, there is no significant growth of average nodes visited with increasing block size, as its complexity is directly proportional to the depth of the tree. As the tree depth remains constant in this experiment there is no observable growth in the average nodes visited measure.

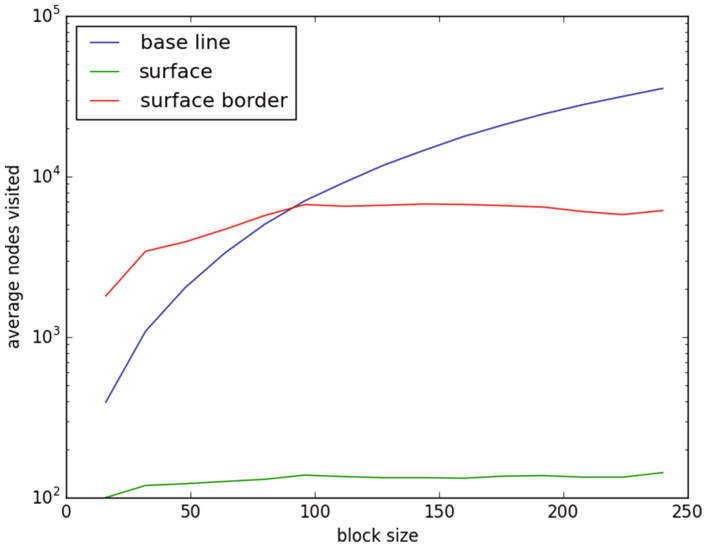

Fig. 8. Average nodes visited vs block size

4.4 Comparison: Querying Time

In Table 1 we see the average query time per block size in seconds for different methods. The naive approach refers to the conventional integral volume query. It is not possible to query the integral volume for a region greater than blocksize 4 with our setup as the memory required exceeds the memory available. We observe that base line system is faster for small sized query region. From block size 8 onwards the surface method performs better. Block size 8 corresponds to a physical region of approximately 40 m × 40 m × 40 m. The surface-border method has the poorest performance in terms of querying time for all block sizes due to high amount of overhead.

Table 1. Average Querying Time for different methods

Block sizes	1	2	4	8	16	32	64
Naive	0.005	0.008	-	-	-	-	-
Base line	0.05	0.06	0.08	0.17	0.33	2.91	3.26
Surface	0.13	0.12	0.15	**0.16**	**0.17**	**0.38**	**0.46**
Surface border	0.46	0.60	0.78	1.50	2.28	4.24	5.25

5 Discussion and Conclusion

Discussion. There are some limitations with our proposed systems. One of them is offline storage. To fast compute the integral volume values we need to store the surfaces at each non-leaf node in full resolution. This is not so difficult for nodes at higher depth of the tree but as we get nearer to the root of the octree, the size of the stored surfaces can be huge. We can consider various compression techniques to achieve this task, but as we get closer and closer to the root level it will become increasingly difficult to store the surfaces. This problem is mitigated by using the surface-border method and the base line system, but both of them are significantly slower than the surface method. All the methods can be viewed from a compromise between storage and computation perspective. The base line system is most memory efficient but it's poorest performing due to exponential growth. The surface method is the best in terms of performance but it is difficult to implement due to high storage requirement near the root. The surface-border method is very slow but relatively much more memory efficient than surface method. The user can choose between these methods according to memory and computation constraints.

Secondly, we cannot query the integral volumes in constant time. However, unlike traditional methods, our methods make it possible to query integral volumes for global scale data in a reasonably fast way.

Conclusion. In this paper, we presented novel methods for querying integral volume for global scale data. Our methods are based on octree hierarchical data structures. We can query the integral volume of any region with time complexity $\mathcal{O}(\log n)$ by storing surfaces at non-leaf nodes, where n is the number of nodes in the octree. The query time is independent of the size of the query region. Thus, our method preserves this beneficial and crucial property of integral representations.

In the future, we may think of a system which is a mixture of Surface and Surface-Border methods. As storing the surfaces are difficult near root, we can store surface-borders there, and we can store surfaces at the nodes with less height. We can choose the mixture according the memory and computation constraints.

References

1. Viola, P., Jones, M.J.: Robust real-time face detection. International J. Comput. Vis. **57**(2), 137–154 (2004)
2. Crow, F.C.: Summed-area tables for texture mapping. ACM SIGGRAPH Comput. Graph. **18**(3), 207–212 (1984)
3. Bay, H., Ess, A., Tuytelaars, T., Van Gool, L.: Speeded-up robust features (surf). Comput. Vis. Image Understand. **110**(3), 346–359 (2008)
4. Holzer, S., Rusu, R.B., Dixon, M., Gedikli, S., Navab, N.: Adaptive neighborhood selection for real-time surface normal estimation from organized point cloud data using integral images. In: 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 2684–2689. IEEE (2012)

5. Shafait, F., Keysers, D., Breuel, T.M.: Efficient implementation of local adaptive thresholding techniques using integral images. In: Electronic Imaging 2008, International Society for Optics and Photonics, pp. 681510–681510 (2008)
6. Wang, X., Doretto, G., Sebastian, T., Rittscher, J., Tu, P.: Shape and appearance context modeling. In: IEEE 11th International Conference on Computer Vision, 2007, ICCV 2007, pp. 1–8. IEEE (2007)
7. Urschler, M., Bornik, A., Donoser, M.: Memory efficient 3d integral volumes. In: Proceedings of the IEEE International Conference on Computer Vision Workshops, pp. 722–729 (2013)
8. Belt, H.: Storage size reduction for the integral image. Technical report, Philips Research (2007)
9. Elseberg, J., Borrmann, D., Nüchter, A.: One billion points in the cloud-an octree for efficient processing of 3d laser scans. ISPRS J. Photogrammetry Remote Sens. **76**, 76–88 (2013)
10. Glassner, A.S.: Multidimensional sum tables. In: Graphics Gems, pp. 376–381. Academic Press Professional, Inc. (1990)
11. Meagher, D.: Geometric modeling using octree encoding. Comput. Graph. Image Process. **19**(2), 129–147 (1982)
12. Laine, S., Karras, T.: Efficient sparse voxel octrees-analysis, extensions, and implementation. NVIDIA Corporation 2 (2010)
13. Wurm, K.M., Hornung, A., Bennewitz, M., Stachniss, C., Burgard, W.: Octomap: a probabilistic, flexible, and compact 3d map representation for robotic systems. In: Proceedings of the ICRA 2010 Workshop on Best Practice in 3D Perception and Modeling for Mobile Manipulation, vol. 2 (2010)
14. Schwartz, J.: Bing maps tile system. https://msdn.microsoft.com/en-us/library/bb259689.aspx
15. You, Y., Fan, L., Roimela, K., Mattila, V.V.: Simple octree solution for multiresolution lidar processing and visualisation. In: 2014 IEEE International Conference on Computer and Information Technology (CIT), pp. 220–225. IEEE (2014)
16. Babahajiani, P., Fan, L., Gabbouj, M.: Object recognition in 3d point cloud of urban street scene. In: Jawahar, C.V., Shan, S. (eds.) ACCV 2014, Part I. LNCS, vol. 9008, pp. 177–190. Springer, Heidelberg (2015). doi:10.1007/978-3-319-16628-5_13
17. Vo, A.V., Truong-Hong, L., Laefer, D.F., Bertolotto, M.: Octree-based region growing for point cloud segmentation. ISPRS J. Photogrammetry Remote Sens. **104**, 88–100 (2015)