Multi-scale Voxel Hashing and Efficient 3D Representation for Mobile Augmented Reality

Yi Xu Yuzhang Wu Hui Zhou
JD.COM Silicon Valley Research Center, JD.COM American Technologies Corporation
Mountain View, CA, USA
{yi.xu,yuzhang.wu,hui.zhou}@jd.com

Abstract

In recent years, Visual-Inertial Odometry (VIO) technologies have been making great strides in both research community and industry. With the development of ARKit and ARCore, mobile Augmented Reality (AR) applications have become popular. However, collision detection and avoidance is largely un-addressed with these applications. In this paper, we present an efficient multi-scale voxel hashing algorithm for representing a 3D environment using a set of multi-scale voxels. The input to our algorithm is the 3D point cloud generated by a VIO system (e.g., ARKit). We show that our method can process the 3D points and convert them into multi-scale 3D representation in real time, while maintaining a small memory footprint. The 3D representation can be used to efficiently detect collision between digital objects and real objects in an environment in AR applications.

1. Introduction

Mobile AR technologies use a mobile phone's camera and its display to provide live view of the surrounding physical environment, which is then "augmented" by computer-generated elements (*e.g.*, digital objects) [3, 6, 17]. A key component of mobile AR is monocular Simultaneous Localization and Mapping (SLAM), which constructs and updates a map of an unknown environment while simultaneously keeping track of the camera pose.

Without additional knowledge of the scene geometry, monocular SLAM approach usually relies on structure-from-motion (SfM) to bootstrap camera tracking. SfM methods reconstruct an initial structure of the scene and recover camera pose at the same time. However, these reconstructions are up to an arbitrary scale. With the recent advancement of sensor fusion technology, Visual-Inertial Odometry (VIO) becomes mature enough for commercial use. Two most prominent examples are Apple's ARKit and Google's ARCore platforms. By fusing visual and inertial sensor information, metric scale of the environment can be estimated. With accurate camera tracking and scale estimation, ARKit and ARCore allow seamless blending of digital objects into the physical environment. There are already many mobile AR applications running on these two platforms in different fields (e-commerce, gaming, education, etc.) that leverage such capability.

However, one important issue that has not been addressed by these systems is collision detection and avoidance between digital objects and real physical objects in the environment. Without such function, once digital objects are placed in the physical environment, they can be moved around freely in the space mapped from the real physical environment. This can cause digital objects and real objects occupy the same physical space; leading to unrealistic perception. Conventional approaches for collision detection and avoidance do not meet the need for mobile AR applications as 1) the computation and memory constraint of mobile phone as well as the real-time requirement for mobile AR application are both strict; 2) most AR applications, which run on third party proprietary AR platforms such as ARKit or ARCore, cannot access the complete 3D reconstruction information from the AR platforms.

To solve this problem in an efficient way on mobile devices, we developed a multi-scale voxel hashing algorithm (Fig. 1). Our method takes the 3D points generated by a monocular SLAM system as input and uses a hash map to store the data into voxels. Nearby points might be represented by a common voxel. We perform bottom-up merging of voxels in order to reduce total memory footprint. Moreover, since the voxel hashing scheme implicitly removes redundant 3D points, point correspondence between frames is not required to identify repeating points. This allows our method to work with ARKit and ARCore, where such correspondence data is proprietary and not accessible to developers.

The rest of the paper is organized as follows. Section 2 discusses the related work in monocular SLAM and 3D

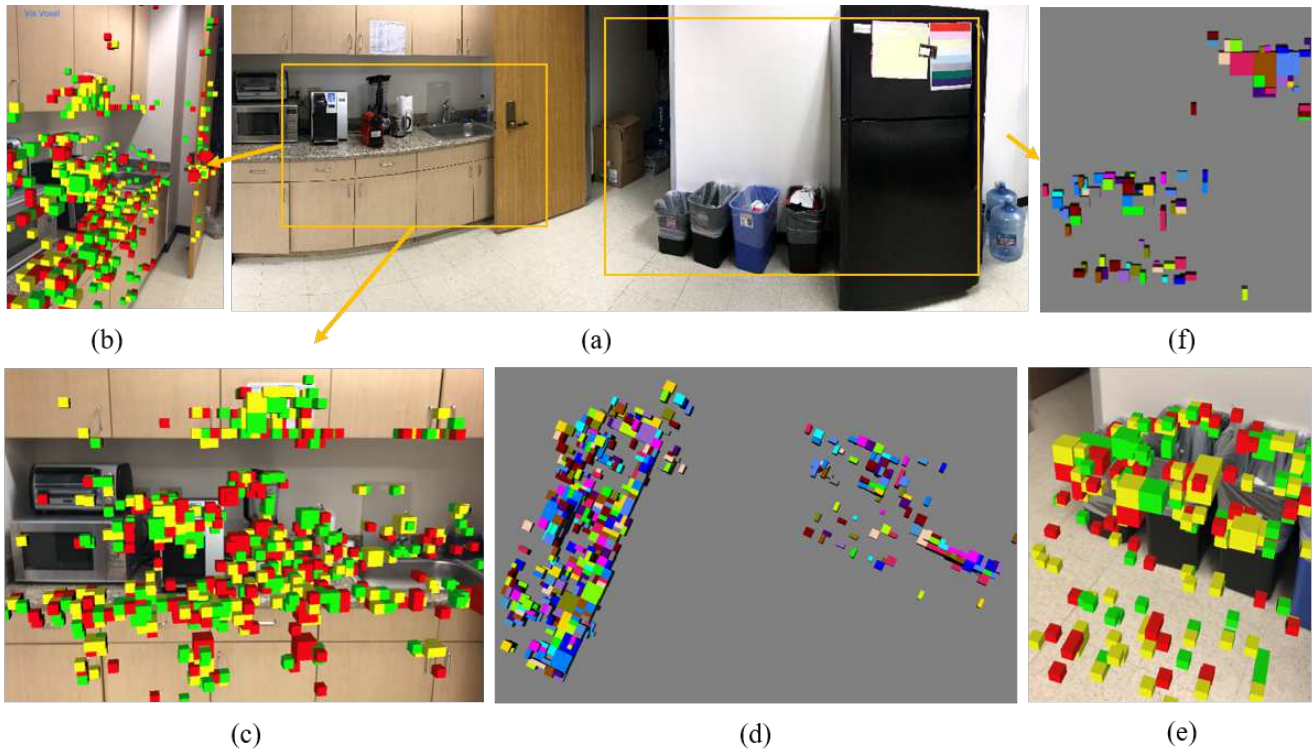


Figure 1. Multi-scale voxel representation for point cloud data generated by ARKit. (a) A panorama of an office kitchen scene. (b) Side-view of the sink area with voxels superimposed on the scene and rendered with random red/yellow/green color. (c) Front view of the sink area with voxles superimposed. (d) Top-down view of the entire voxel representation. (e) Close-up view of the trash bins and nearby floor with voxels superimposed. (f) A front view of the refrigerator area.

reconstruction, collision detection for AR applications, and our contributions. Section 3 details our multi-scale voxel hashing algorithm. Section 4 presents our implementation and experiments. Concluding remarks and discussion on limitations and future work are provided in Section 5.

2. Related Work

Our work uses point clouds generated by monocular SLAM algorithm as input. In addition, our method borrows ideas from 3D reconstruction, especially online reconstruction algorithms. However, the objective is not to reconstruct a detailed model of real objects. Finally, we enable collision detection in AR applications. In this section, we discuss related work in monocular SLAM, 3D reconstruction using point cloud data, and collision detection in AR.

2.1. Monocular SLAM

Monocular SLAM is the problem of localization and mapping of the environment using a single camera. It has many applications in robot navigation, mobile entertainment, e-commerce, etc. Early work solves the monocular SLAM problem by using filtering techniques to jointly estimate 3D map point locations and camera poses [8]. Keyframe based approaches achieve higher accuracy than filtering methods by performing more computationally intensive Bundle Adjustment (BA) optimization. Among many keyframe based approaches, PTAM is the most prominent one [13]. For first time it introduced the idea of running tracking and mapping in parallel threads and achieved impressive results for tabletop AR applications. Another excellent and complete SLAM system is ORB-SLAM [19], which is capable of performing localization, mapping, loop closing, and relocalization for large environments in real time. ORB-SLAM uses ORB features [23] to perform feature detection and tracking. Both PTAM and ORB-SLAM are feature-based. They apply optimization over feature points in the scene that are detected by feature detectors. On the contrary, LSD-SLAM applies direct optimization over image pixels to achieve similar results in real time [10]. All the above SLAM approaches use visual sensor only; thus they cannot obtain metric scale reconstruction of the environment. Recently, CNN-SLAM fuses dense depth estimation form Convolutional Neural Networks (CNNs) and sparse depth from direct monocular SLAM [24]. CNN-SLAM also mitigates the scale ambiguity problem because absolute scale of the objects can be learned from examples.

The scale ambiguity can also be solved by VIO methods, which incorporate inertial sensors into the optimization formulation. By integrating the acceleration measurements from IMUs, camera translation between frames can be roughly estimated and further optimized using visual information. Earlier work in VIO includes Extended Kalman Filtering based [16] and keyframe-based [15] approaches. A recent work that can be implemented on smart phones is VINS-MONO [22]. By incorporating both visual measurements and IMU measurements in a joint optimization framework, VINS-MONO not only can perform real time tracking and mapping, but also achieves metric reconstruction of the scene. More recently, the releases of Apple's ARKit [2] and Google's ARCore [11] allow commercial application development on iOS and Android systems.

2.2. 3D Reconstruction

Our approach also borrows ideas from research work in 3D reconstruction, especially online reconstruction. Unlike offline methods, online approaches aim to fuse range data incrementally into a consistent 3D representation. One approach is to use a point-based representation for the 3D geometry that ignores connectivity information. The point cloud captured by 3D sensors can be registered together with the already reconstructed 3D model. Loop closure is addressed to make sure reconstruction errors do not accumulate. This type of approach has been applied to small scale objects [25] and large scale indoor environments [12]. The point-based representation is memory intensive and does not allow for easy collision computation. In such cases, additional space partition data structure (*e.g.*, octree) might be needed.

An alternative approach uses a volume-based representation. The seminal work by Curless and Levoy [7] converts depth samples into signed distance field and stores the values in a regular voxel grid. Surfaces can then be extracted from the volumetric data using isosurfacing method. KinectFusion [20] extends the volumetric approach to achieve real-time 3D reconstruction using Kinect sensor. Nießner *et al*. [21] adopt a spatial hashing scheme to access and update surface data in real time. Their system allows 3D reconstruction at large and fine scale.

2.3. Collision Detection in AR

Collision detection for Augmented Reality has been studied for more than 20 years. In his work, Aliaga [1] implemented collision detection and response between virtual objects and real objects by manually creating a digital model for the real environment. Collision is detected between the digital model of the environment and virtual objects. Breen *et al*. [5] tried to solve this problem by registering the camera image with a known 3D model of the environment and by using depth estimation from stereo camera pair. These methods do not work with monocular SLAM system where the geometry of the environment is unknown. Decker *et al*. [9] proposed a method to detect collision in the free-viewpoint video setting. However, they use multiple calibrated cameras. Using monocular camera, Lee *et al*. [14] proposed an image-based method to detect collision between a user's hand and virtual objects; but collision in 3D space is not addressed. To the best knowledge of the authors, ours is the first method that implements 3D collision detection for Augmented Reality applications.

2.4. Our Contributions

Building on top of SLAM systems, our approach is capable of processing the 3D sparse point cloud captured by such systems. Our approach also enables collision detection and avoidance, which has a significant impact on the realism of an AR application. In addition, our method can work with other types of range scanners if the pose of the scanning device can be tracked in real time, either using an external device or using a range-based SLAM algorithm. The fine models created by 3D reconstruction methods can be used for collision detection and avoidance; however, a detailed 3D reconstruction is not only computationally expensive but also unnecessary. We choose to represent the 3D environment using a set of multi-scale voxels for mobile AR applications.

Our contributions are:

1. An efficient 3D representation using sparse point sets generated by VIO systems as input.
2. A multi-scale bottom-up merging scheme that effectively merges voxels together and reduces memory footprint.
3. An easy-to-implement collision detection and avoidance approach for mobile AR applications.

3. Our Approach

We extend the voxel hashing scheme and apply that to the sparse point cloud data generated by a VIO system; allowing for easy and fast collision detection and avoidance between digital objects and real objects in AR applications.

For sparse point data, using a regular grid for storage is not effective because most of the voxels will be marked as empty. A hierarchical data structure will alleviate this problem, but it is still computationally expensive. Moreover, isosurfacing from sparse volumetric data is impractical. We use a spatial voxel hashing scheme similar to the one used in [21]: a hash key is computed for each point using its integer coordinates and voxel data is then stored in a hash table. With such a hashing mechanism, the number of voxels still increases with the number of input 3D points. When an area of the scene contains rich textures, there will

be many entries in the hash table. This limits the capability of performing certain tasks such as collision avoidance. Each small voxel has to be tested against the digital object to detect collision. To solve this problem, we propose a multi-scale bottom-up merging procedure that can effectively merge voxels together to reduce memory footprint.

3.1. Voxel Hashing

For each point, we map the integer components of the point's 3D world coordinates to a hash value using the formula [21]:

$$H(x, y, z) = (x * p_1 \oplus y * p_2 \oplus z * p_3) \bmod n \qquad (1)$$

where p_1, p_2, and p_3 are large prime numbers, n is the number of slots in the hash table, and $\oplus$ is exclusive OR operator. If the integer components of coordinates have limited number of digits, we multiply the coordinates by a scaling integer S. For example, in ARKit, coordinates are specified in meters; therefore, measurements have 0 or 1 digit of integer. We use $S = 100$ to scale the world coordinates of 3D points. After rounding, points with the same integer coordinates are represented implicitly by a voxel, whose size is $1cm^3$. When multiple voxels map to the same hash key value, hash table collision happens. To handle this, at each hash table slot, we store a linked list of voxel entries. Whenever a new voxel entry is created, it is inserted to the end of the list at the corresponding hash table slot.

Each voxel entry contains the integer coordinates (x, y, z), which correspond to the coordinates of the vertex that is closest to the origin. Each voxel also stores a density value d, which equals to how many points are bounded by this voxel. The density value d can be used to filter out voxels that contain a small number of points (*e.g.* $d < 50$) during collision computation; thus removing possible outliers produced by VIO system.

3.2. Multi-scale Voxel Representation

The minimum size of a voxel m_v can be controlled by the scaling integer S. For example, $S = 100$ corresponds to $m_v = 1cm$, while $S = 10$ corresponds to $m_v = 10cm$. A small S value leads to large voxel size and a large S value leads to small voxel size. If the m_v is too small, we run into the risk of large memory footprint and higher computational cost for collision detection and avoidance. If m_v is too big, the resulting 3D representation might be over simplified and not tight enough. To mitigate this problem, we introduce a multi-scale voxel hashing representation.

The key idea is to include a level value l at each voxel. This allows us to store voxels of different sizes in the hash table. For an input 3D point, we first compute its hash key using Equation. 1. Then we perform a linear search on the list of voxel entries indexed by the hash value and check if the point is already encompassed by any voxel. If a voxel is not found, we create a new voxel at the smallest scale for this point. In the voxel, we store the integer coordinates (x, y, z) and its scale level $l = 0$.

Once the voxel is added to the hash map, we use a recursive procedure to merge neighboring voxels into higher level voxels. A straightforward way is to merge uniformly along three axes. If all 8 voxels that form a voxel at the next level are all present in the hash map, we remove the 8 voxels and insert a new voxel at the higher level into the hash map (Fig. 2a). The density of the new voxel is the sum of all densities from the 8 smaller voxels. This procedure is repeated until no more voxels can be merged or a predefined maximum level of voxel is reached. For any voxel, the 7 buddy voxels that form a voxel at the next level can be easily located from the hash map using integer division and multiplication.

In reality, this method is not efficient because it requires all 8 voxels to be present before we can merge them. This is especially not practical for surface data where the interior of

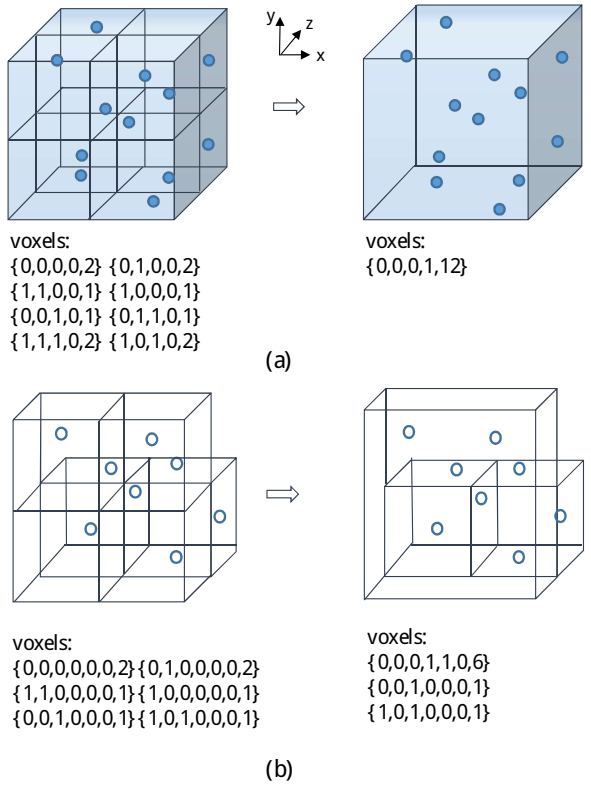


Figure 2. Multi-scale voxel merging. (a) Uniform merging along three axes. Each voxel is stored as a 5-tuple: $\{x, y, z, l, d\}$, which represents coordinates, scale of the voxel, and density. (b) Non-uniform merging. Each voxel is a 7-tuple: $\{x, y, z, l_x, l_y, l_z, d\}$, which represents coordinates, scale along each of the three dimensions, and density.

the scene is not visible. To solve this problem, we use a different scale level for each of the three dimensions: l_x, l_y, l_z. After a new voxel is inserted into the hash table, we merge the 4 voxels that form a voxel at the next level on 2 out of the 3 dimensions if possible (Fig. 2b). For example, if 4 voxels that form a larger voxel with *X*, *Y* dimension at level 1 but *Z* dimension at level 0, we proceed to merge these 4 voxels and insert a new voxel with $l_x = 1, l_y = 1, l_z = 0$. In case there are more than one options of merging (*e.g.*, the bottom 4 voxels can also be merged in Fig. 2b), we simply choose one and ignore the others. Since this can only happen at the intersection of two planes, it will not have a large impact on the performance. Planar structures that are approximately axis-aligned can be effectively represented by our approach. The 3 buddy voxels can also be located from the hash map easily.

We can also allow merging two voxels along one axis. In this way, a thin long structure can be represented using our approach. In experiments, we found that it does not improve the performance much.

Optionally, we can adjust the criterion for merging voxels to allow for more flexibility. For example, when 3 out of 4 voxels that form a voxel at the next level are present in the hash table, we create a new voxel and remove the 3 voxels from the data structure.

3.3. Align Voxels with Room Orientation

One limitation of our voxel representation is that the voxels are axis-aligned; i.e., they are aligned with the coordinate axes of the space; and these axes are not necessarily aligned with room orientation. They depend on the initial orientation of the mobile device when VIO system bootstraps camera tracking. The merging process looks for planar structures along the three axes; therefore if the voxels are not aligned with the room orientation, the bottom-up merging process cannot effectively reduce the total number of voxels.

To solve this problem, we use vertical plane detection to estimate the orientation of a room. In our AR application, we guide the user to scan the part of the scene with strong presence of vertical surfaces. Then, from the reconstructed 3D points, we use RANSAC to robustly estimate a plane that is perpendicular to the ground plane. The ground plane orientation can be estimated by motion sensing hardware on a mobile phone. When the number of inliers of the vertical plane is larger than a threshold, we use the normal of the plane as the new X-axis and transform all captured 3D points into the new coordinate system. A simple rotation is sufficient because the Y-axis always points to the opposite direction of gravity as detected by the motion sensing hardware in a VIO system like ARKit or ARCore.

3.4. Collision Detection

Collision detection can be efficiently computed using our multi-scale voxel representation. A digital object can be represented by a bounding box, a set of bounding boxes, a bounding cylinder, or even a set of polygons or voxels. At real time, the geometry is compared against all the multi-scale voxels. Because the voxels are all box-shaped, collision can be efficiently detected. For example, to detect collision between two boxes, we only need to check if two boxes overlap at all three dimensions, each requires 2 comparisons. This results in only $3 * 2 = 6$ comparisons. In our implementation, for robustness, a collision is detected only when the number of collided voxels with the digital object is larger than a threshold (*e.g.*, 10).

In typical ARKit or ARCore applications, a horizontal support plane (ground or table surface) is first determined. Then a digital object is placed on the support plane for viewing and interaction. During collision detection, we omit those voxels which are close enough to the object support plane; otherwise, feature points from this plane will lead to unintended collision with the digital object.

4. Experimental Results

To validate our approach, we implemented our algorithm in C++ and an iOS application using ARKit and Object C. We tested our application on iPhone 8. The VIO algorithm of ARKit runs at 60 fps and produces dozens to hundreds of 3D points per frame depending on the complexity of the scene. Our algorithm processes these 3D points and inserts them into multi-scale voxel hash data structure in real time.

We tested our approach on 4 different scenes including an office kitchen, a storage room, a table, and a conference room. We also captured a sequence of point clouds and images for each scene to process offline. Inter-point distance varies for different scenes. When a scene is closer to the camera, the reconstructed points are closer to each other; thus requiring smaller voxels to better approximate the geometry of the scene. We estimate the scene depth from a sequence of initial frames and use the depth to determine the minimal voxel size m_v. In our system, we use the following empirically determined thresholds:

$$m_v = \begin{cases} 2\,cm; & D < 0.75m \\ 4\,cm; & 0.75m \leq D < 1m \\ 8\,cm; & D \geq 1m \end{cases} \quad (2)$$

where D is the initial scene depth.

In each case, our method can efficiently process the incoming 3D points generated by ARKit and convert them into multi-scale voxels in real time. Table 1 shows the results. We list capture time, number of total 3D points reconstructed by ARKit, number of voxels reconstructed by

Scene	Kitchen	Storage Room	Table	Conference Room
Capture Time (sec)	22.8	64.9	62.6	36.3
Number of 3D Points	276,137	594,096	465,637	40,796
Number of Voxels	4,026	7,808	853	387
Min. Voxel Size (cm)	4	4	4	8
Compression Ratio	82.3	91.3	655.1	126.5
Memory (kB)	39.3	76.25	8.33	3.78

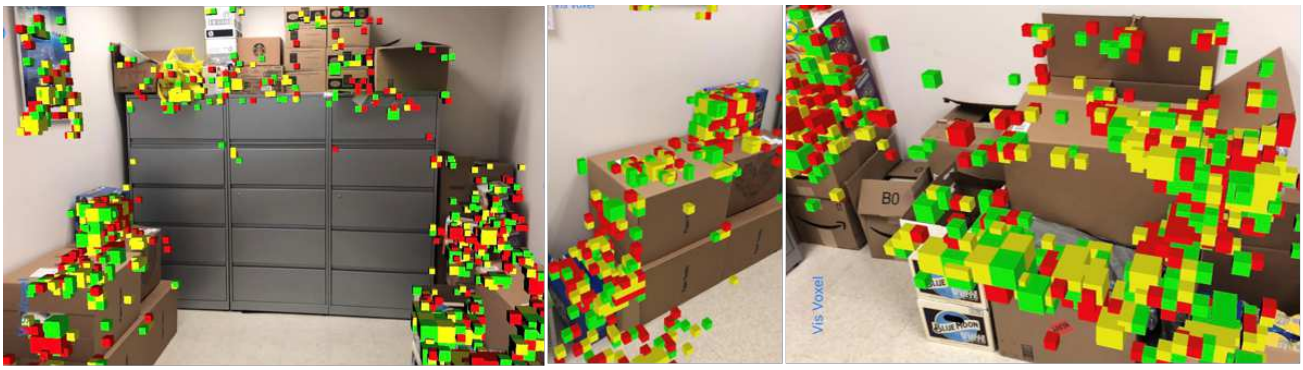
Table 1. Results of our approach applied to different scenes.

Total Capture Time	30s	60s	120s	240s
Number of 3D Points	107,730	172,649	315,339	544,110
Number of Voxels	2,215	2,885	3,749	4,542

Table 2. Increasing capture time on the Kitchen scene.

Scene	Kitchen	Storage Room	Table	Conference Room
Single Scale Representation	6,278	13,569	1,251	631
Multi-Scale Representation	4,026	7,808	853	387
Reduction	35.9%	42.5%	31.8%	38.7%

Table 3. Compare number of reconstructed voxels using multi-scale and single scale schemes.



(a) (b) (c)

Figure 3. Multi-scale voxel representation for the storage room scene. (a) Complete view of the storage room with voxel representation superimposed. (b) A different view of the left side of the room. (c) A different view of the right side of the room.

our algorithm, and minimal voxel size used by our algorithm. We also listed compression ratio, which is the ratio between storage required for all 3D points (float point coordinates) to the storage required for multi-scale voxel data structure. Each voxel entry requires 3 bytes for integer coordinates (x, y, z), 3 bytes for l_x, l_y, l_z scales, and 4 bytes for density. For less complex scenes, such as Table and Conference Room, the compression ratio is much larger than complex scenes like Kitchen and Storage Room. It is worth noting that VIO systems like ARKit do not provide inter-frame correspondence information to developers; therefore, the 3D points are largely redundant, and the total number of points is much larger than the actual features in the scene.

Fig. 1 shows a panorama of the office kitchen scene and the reconstructed voxel structure from a few different viewpoints. Fig. 3 shows several pictures captured by an iPhone running our AR application and with reconstructed voxels superimposed on the view. Fig. 4 shows a picture of the table top scene (top) and the same scene with our multi-scale voxel representation superimposed (bottom). As illustrated, our approach can build a 3D representation of the scene in real time.

Our approach scales well with increasing number of 3D points when the AR application runs for longer period of time. To show this, we increase the running time from 30 seconds to 240 seconds and record the total number of

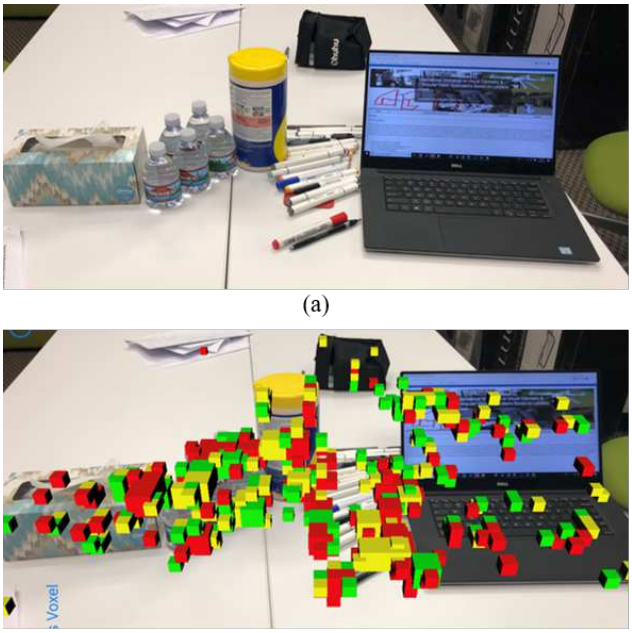


Figure 4. Multi-scale voxel representation for the table top scene. a) Top image shows a photo of the scene. b) The bottom image shows voxel representation.

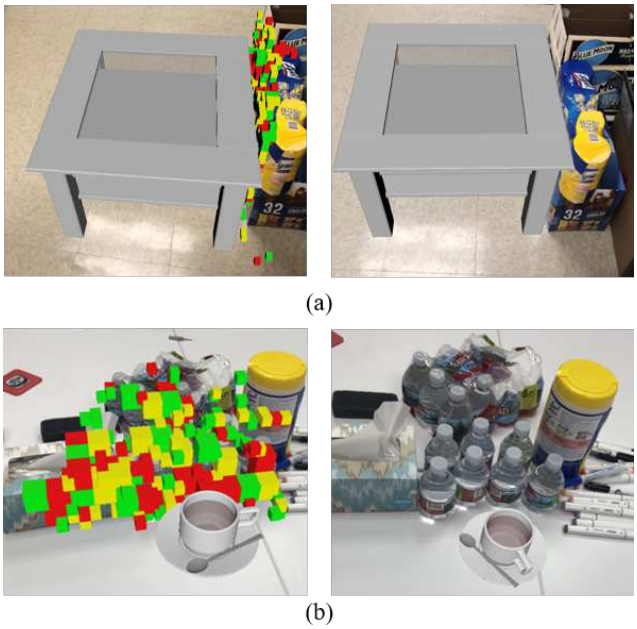


Figure 5. Collision detection and avoidance using our approach. (a) Collision between virtual table and real objects in the storage room with voxel representation superimposed (left) and without (right). (b) Collision between virtual coffee cup and real objects on the table.

points generated by ARKit and number of reconstructed voxels for the Kitchen scene (Table 2). The minimal voxel size was set to be 8cm. While the number of points increases approximately linearly with capture time, the number of reconstructed voxels grows at a much slower rate.

To test the effectiveness of our bottom-up merging scheme, we apply our voxel hashing algorithm with and without multi-scale processing. The results are shown in Table 3, where we list the number of reconstructed voxels using single scale (first row) and multi-scale (second row) schemes. Our multi-scale scheme reduces total memory footprint consistently by 30% to 40%.

We implemented collision avoidance in our AR application. When a user interacts with the digital object during AR view, we detect collision in real time using the overlapping bounding box approach described in Section 3.4 and stop the movement of the digital object if potential collision is detected.

Fig. 5 shows collision detection results using our approach. We "move" the virtual objects towards the obstacles and show the rendering results from the mobile phone. On the left images, we superimpose the voxel representation to illustrate that the collision is well detected using the voxels. The video accompanying this paper shows collision detection and avoidance implemented on iOS device using ARKit.

5. Conclusion and Future Work

In this paper, we present an efficient algorithm that can process 3D sparse point sets generated by a mobile VIO system (*e.g.*, ARKit and ARCore). Our approach is capable of transforming the 3D point cloud data into a multi-scale voxel hash structure with real-time performance. Our data structure requires a small memory footprint, scales well with increasing capturing time, and supports efficient collision detection for mobile Augmented Reality applications. Moreover, our approach does not require point correspondence between frames. This allows our method to work with ARKit and ARCore, where such correspondence data is proprietary and not accessible to developers.

One of the limitations is that we assume a static scene. Thus our method can not detect collision between virtual objects and moving real objects. This is because the structure of moving objects cannot be reliably recovered by a monocular SLAM algorithm. Another limitation is that our method relies on the feature points detected by feature detectors; thus our method does not work with texture-less surfaces within the scene.

In the future, we would like to improve our approach in several aspects. Right now, the minimal voxel size is determined using estimated scene depth from an initial sequence of frames. In the future, we want to explore dynamically adjusting the minimal voxel size based on the current scene

depth. Second, the proposed algorithm uses plane detection to estimate the orientation of the room so that voxels are aligned with the major vertical surfaces in the scene. In the future, we want to explore the possibility of adding orientation to the voxels to better approximate surfaces with different orientations. Finally, we would like to explore the possibility of using image data to better infer 3D surface information. For example, superpixel can be used to recover piecewise-planar surfaces [4]. Another possibility is to reconstruct more 3D points for weakly-textured areas using an advanced feature descriptor [18].

Acknowledgement

The authors would like to thank the reviewers for their insightful comments and detailed reviews. The authors would also like to thank JD's AR/VR product development team for inspiring discussions.

References

[1] D. G. Aliaga. Virtual and real object collisions in a merged environment. *Proc. Virtual Reality Software and Technology '94*, pages 287–298, 1994. 3

[2] Apple Inc. Arkit - apple developer, 2018. "https://developer.apple.com/arkit/. 3

[3] C. Arth, R. Grasset, L. Gruber, T. Langlotz, A. Mulloni, and D. Wagner. The history of mobile augmented reality. *arXiv preprint arXiv:1505.01319*, 2015. 1

[4] A. Bódis-Szomorú, H. Riemenschneider, and L. V. Gool. Fast, approximate piecewise-planar modeling based on sparse structure-from-motion and superpixels. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 469–476, June 2014. 8

[5] D. E. Breen, R. T. Whitaker, E. Rose, and M. Tuceryan. Interactive occlusion and automatic object placement for augmented reality. *Computer Graphics Forum*, 15(3):11–22, 1996. 3

[6] D. Chatzopoulos, C. Bermejo, Z. Huang, and P. Hui. Mobile augmented reality survey: From where we are to where we go. *IEEE Access*, 5:6917–6950, 2017. 1

[7] B. Curless and M. Levoy. A volumetric method for building complex models from range images. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 303–312, New York, NY, USA, 1996. ACM. 3

[8] A. J. Davison, I. D. Reid, N. D. Molton, and O. Stasse. MonoSLAM: Real-time single camera SLAM. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(6):1052–1067, June 2007. 2

[9] B. D. Decker, T. Mertens, and P. Bekaert. Interactive collision detection for free-viewpoint video. In *GRAPP 2007, Proceedings of the Second International Conference on Computer Graphics Theory and Applications, Barcelona, Spain, March 8-11, 2007, Volume AS/IE*, pages 114–120, 2007. 3

[10] J. Engel, T. Schöps, and D. Cremers. LSD-SLAM: Large-scale direct monocular SLAM. In *European Conference on Computer Vision (ECCV)*, September 2014. 2

[11] Google Inc. Arcore overview, 2018. "https://developers.google.com/ar/discover/. 3

[12] P. Henry, M. Krainin, E. Herbst, X. Ren, and D. Fox. RGB-D mapping: Using Kinect-style depth cameras for dense 3D modeling of indoor environments. *The International Journal of Robotics Research*, 31(5):647–663, 2012. 3

[13] G. Klein and D. Murray. Parallel tracking and mapping for small AR workspaces. In *2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality*, pages 225–234, Nov 2007. 2

[14] D. Lee, S. G. Lee, W. M. Kim, and Y. J. Lee. Sphere-to-sphere collision estimation of virtual objects to arbitrarily-shaped real objects for augmented reality. *Electronics Letters*, 46(13):915–916, June 2010. 3

[15] S. Leutenegger, S. Lynen, M. Bosse, R. Siegwart, and P. Furgale. Keyframe-based visual-inertial odometry using nonlinear optimization. 34, 02 2014. 3

[16] M. Li and A. I. Mourikis. High-precision, consistent EKF-based visual-inertial odometry. *Int. J. Rob. Res.*, 32(6):690–711, May 2013. 3

[17] H. Ling. Augmented reality in reality. *IEEE MultiMedia*, 24(3):10–15, 2017. 1

[18] G. Lu, L. Nie, S. Sorensen, and C. Kambhamettu. Large-scale tracking for images with few textures. *IEEE Transactions on Multimedia*, 19(9):2117–2128, Sept 2017. 8

[19] R. Mur-Artal, J. M. M. Montiel, and J. D. Tards. ORB-SLAM: A versatile and accurate monocular SLAM system. *IEEE Transactions on Robotics*, 31(5):1147–1163, Oct 2015. 2

[20] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohi, J. Shotton, S. Hodges, and A. Fitzgibbon. KinectFusion: Real-time dense surface mapping and tracking. In *2011 10th IEEE International Symposium on Mixed and Augmented Reality*, pages 127–136, Oct 2011. 3

[21] M. Nießner, M. Zollhöfer, S. Izadi, and M. Stamminger. Real-time 3D reconstruction at scale using voxel hashing. *ACM Trans. Graph.*, 32(6):169:1–169:11, Nov. 2013. 3, 4

[22] T. Qin, P. Li, and S. Shen. VINS-Mono: A robust and versatile monocular visual-inertial state estimator. *arXiv preprint arXiv:1708.03852*, 2017. 3

[23] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski. ORB: An efficient alternative to SIFT or SURF. In *2011 International Conference on Computer Vision*, pages 2564–2571, Nov 2011. 2

[24] K. Tateno, F. Tombari, I. Laina, and N. Navab. CNN-SLAM: Real-time dense monocular SLAM with learned depth prediction. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6565–6574, July 2017. 2

[25] T. Weise, T. Wismer, B. Leibe, and L. V. Gool. In-hand scanning with online loop closure. In *2009 IEEE 12th International Conference on Computer Vision Workshops, ICCV Workshops*, pages 1630–1637, Sept 2009. 3