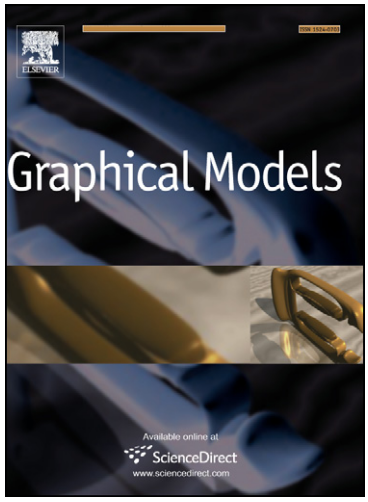



Octree-based Fusion for Realtime 3D Reconstruction

Ming Zeng, Fukai Zhao, Jiaxiang Zheng, Xinguo Liu

PII: S1524-0703(12)00076-8
DOI: http://dx.doi.org/10.1016/j.gmod.2012.09.002
Reference: YGMOD 828

To appear in: *Graphical Models*

Received Date: 29 August 2012
Revised Date: 18 September 2012
Accepted Date: 25 September 2012

Please cite this article as: M. Zeng, F. Zhao, J. Zheng, X. Liu, Octree-based Fusion for Realtime 3D Reconstruction, *Graphical Models* (2012), doi: http://dx.doi.org/10.1016/j.gmod.2012.09.002

# Octree-based Fusion for Realtime 3D Reconstruction

Ming Zeng Fukai Zhao Jiaxiang Zheng Xinguo Liu[†]
State Key Lab of CAD&CG, Zhejiang University
[†] Corresponding author: xgliu@cad.zju.edu.cn

**Abstract**

This paper proposes an octree-based surface representation for KinectFusion, a realtime reconstruction technique of in-door scenes using a low-cost moving depth camera and a commodity graphics hardware. In KinectFusion, the scene is represented as a signed distance function (SDF) and stored as an uniform grid of voxels. Though the grid-based SDF is suitable for parallel computation in graphics hardware, most of the storage are wasted, because the geometry is very sparse in the scene volume. In order to reduce the memory cost and save the computation time, we represent the SDF in an octree, and developed several octree-based algorithms for reconstruction update and surface prediction that are suitable for parallel computation in graphics hardware. In the reconstruction update step, the octree nodes are adaptively split in breath-first order. To handle scenes with moving objects, the corresponding nodes are automatically detected and removed to avoid storage overflow. In the surface prediction step, an octree-based ray tracing method is adopted and parallelized for graphic hardware. To further reduce the computation time, the octree is organized into four layers, called top layer, branch layer, middle layer and data layer. The experiments showed that, the proposed method consumes only less than 10% memory of original KinectFusion method, and achieves faster performance. Consequently, it can reconstruct scenes with more than 10 times larger size than the original KinectFusion on the same hardware setup.

*Keywords:* Octree, KinectFusion, 3D Reconstruction, Graphics Hardware, Signed Distance Function, Ray Casting

## 1. Introduction

Reconstruction of 3D scenes has been an important and active research area in computer vision and computer graphics. The traditional methods take advantages of 3D scanning hardware, depth camera, stereo vision and structured lights techniques to obtain the surfaces of the scene. Recently, Newcombe et al. [1] proposed a realtime reconstruction method for in-door scenes, called KinectFusion. They use Kinect, a low-cost moving depth camera, to capture the scene, and merge the captured depth maps into the final result. The scene is represented as a signed distance function (SDF) and stored in an uniform subdivided grid of voxels, so that they can use a commodity graphics hardware to parallelize the computation in tracking the pose of the Kinect and fuse the depth map in the scene, and achieve real time performance.

Since KinectFusion represents the signed distance function in an uniformly subdivided grid of voxels, it requires a lot of memory to store volumetric data. Meanwhile, most of the geometry of scene is essentially 2D manifold, and is very sparse in the scene volume. As shown in the experiments, most of the voxels in the signed distance function is empty, and are wasted consequently. When capturing larger scale scenes with finer details, the memory consumption will dramatically increases, which easily exceeds the memory limit of the commodity graphic hardware. For a resolution of $1024 \times 1024 \times 1024$, the grid data for the signed distance function takes $4GB$ meomery (assuming each voxel takes 4-byte for the distance value).

To address the above problem, we proposed an octree data structure to store the signed distance function of the scene, and developed several algorithms to maintain the octree, and parallelize the computations in graphics hardware. The proposed algorithms greatly exploit the hierarchical structure of the octree and the GPU parallel computation ability. The proposed octree data structure allows for fast traversal, and consequently optimizes the reconstruction performance. In the step of reconstruction update, the octree is traversed in breadth-first order, while in the step of surface prediction, the octree is traversed in a top-down order, which can skip a lot of

empty intermediate nodes. In experiments, the proposed method is much more efficient than the original KinectFusion in terms of memory consumption and running speed.

The main contribution of this paper is a memory-efficient and real-time scene scanning method, which consists of the following novel techniques:

- A memory-efficient and traversal-efficient octree data structure in GPU.
- Octree-based algorithms for reconstruction update and surface prediction.
- Several octree update algorithms for reconstructing dynamic scenes with moving objects.

## 2. Related Work

**Octree in GPU** There are several spatial data structures to organize 2D/3D data in computer graphics, e.g. kd-tree [2], BVH [3], and octree. Among these data structures, octree is used most widely, which appears in many applications, e.g. ray tracing [4], mesh simulation [5], mesh coding [6], and geometric modeling [7, 8, 9]. Octree adaptively splits the space of the scene according to the distribution of geometric primitives of the scene, and drops the empty cells to save the storage cost. As the development of programmable graphics hardware, octree can be built and traversed completely in GPU, and take advantage of the parallel computing feature of GPU to dramatically improve the performance of rendering and geometry processing. Compared with the traditional methods, we deal with time varying octrees.

**3D Reconstruction** There have been many kinds of systems for capturing 3D scenes based on stereo vision, structured lights, 3D laser scanner and depth camera. Stereo vision based methods infer the camera parameters from a set of images of the target scene, match the feature points across the images, and then compute the 3D positions of the matched feature points, recovering a sparse point cloud of the scene [10, 11]. A survey and comparison of stereo methods can be found in [12]. Stereo algorithms have been successfully applied to reconstruct large scale scenes and small scale objects, such as buildings, human body, and faces Stereo, but it is still challenging to handle varying lighting, non-diffuse surface. And the stereo methods are timing consuming and not suitable for real time applications. People in Robotic community focus on simultaneous localisation and mapping (SLAM) [13, 14], which can track the camera in real time and generate a sparse point cloud of the scene. In order to obtain dense reconstruction result in real time, people combined SLAM with multi-view stereo techniques [15, 16]. Recently, Newcombe et al. used iterative image alignment method to obtain the correspondence, instead of using the traditional image feature points, which can densely reconstruct a small scene (about $1m \times 1m \times 1m$) in real time [17].

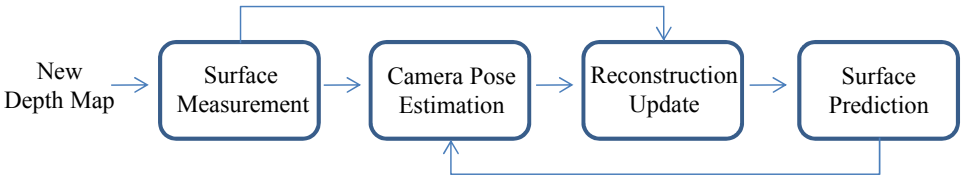
To meet the requirement of real time applications, depth cameras based on time-of-flight and structured light are used to capture a set of depth maps of the scene from different view points, which are then aligned together and fused into a complete surface representation. To align the depth maps, there are a number of registration methods [18, 19] based on the iterative closest point (ICP) method proposed by Chen and Medioni [20]. ICP method is originally developed for rigid objects, and can be extended to handle deformable objects [21, 22, 23, 24].

Recently, Newcombe et al. [1] developed a real time reconstruction method for in-door scenes, called KinectFusion, using a low-cost moving depth camera and commodity graphics hardware. The basic idea of KinectFusion is to maintain a volumetric scene data to align and merge the depth map streamed from the Kinect. KinectFusion can also be used for camera tracking and user interaction in augmented reality [25]. In order to handle unbounded scenes, Whelan et al. [26] extended KinectFusion to "Kintinuous" by shifting the volume and extracting meshes continuously.

Zeng et al. [27] introduced an octree structure for KinectFusion to save the memory cost of KinectFusion and achieve faster performance. Compared with "Kintinuous"[26], this method improves the core parts of the KinectFusion, which is memory-efficient and can be a building block of large scale scanning systems in "Kintinuous". In this paper, we extend the work of Zeng et al. [27] by (1) introducing node removal operation to handle dynamic scenes with moving objects; (2) adding a top layer to the octree for accelerating the traversal in the surface prediction step; (3) presenting detailed experimental results and analysis on the proposed method.

## 3. Overivew

KinectFusion captures the depth map of the scene using Kinect, a low-cost depth camera, and successively fuses the depth map into the reconstructed scene. It represents the scene as the zero valued surface of a signed distance function (SDF) and stores the SDF in an uniformly subdivided grid of voxels. It tracks the pose of the camera, so that the depth map can be correctly positioned in the volumetric grid.




(a) KinectFusion overview

Reconstruction Update using octree

1. Split nodes for storing the newly scanned part
2. Remove nodes if there is any moving object
3. Update the signed distance in the data layer
4. Update the top layer of the octree

Surface prediction using octree

1. Generate a ray toward each pixel and trace the ray in the leaf nodes of the octree
2. For the rays that hit the surface, estimate the position and normal of the intersection

(b) Octree based reconstruction update and surface prediction

Figure 1: KinectFusion overview and our octree based methods for reconstruction update and surface prediction.

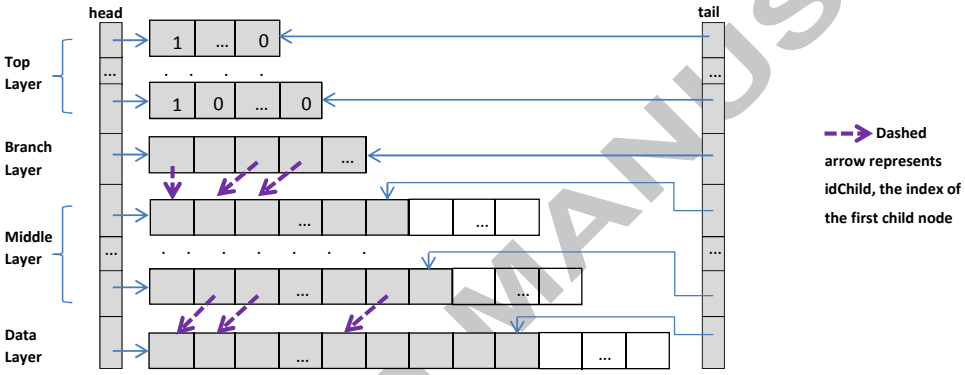



Figure 2: Layered structure of the octree.

As shown in Figure 1, KinectFusion consists of four main steps:

- **Surface Measurement** converts the depth map streamed from the depth camera into a point cloud and estimates the normal of each point.
- **Camera Pose Estimation** uses ICP method to estimate the rigid transformation between the point cloud and the current reconstructed scene surface, obtaining the pose of the depth camera.
- **Reconstruction Update** merges the depth map into the implicit surface of the scene by updating the signed distance function.
- **Surface Prediction** extracts the scene surface from the signed distance function under the current view of the depth camera, and estimates the normal of the surface.

In this paper we follow the same steps of KinectFusion, but the reconstruction update step and the surface prediction step are improved with our octree data structure. In the rest of this paper, we will first introduce the octree structure in Section 4, the reconstruction update step and the surface prediction step in Section 5 and Section 6. Then we present the implementation details and the experimental results in Section 7, and conclude this paper in Section 8.

## 4. Octree Representation

Let $L$ denote the depth of our octree, and the root level is 0. We divide the levels of the octree into four layers: top layer, branch layer, middle layer, and data layer, as illustrated in Figure 2.

- **Top Layer** consists of several levels of nodes at the top part of the octree. This layer is designed for

accelerating ray casting procedure in the surface prediction step. Each node in the top layer store a boolean value to indicating if this node contains scene geometry.

As illustrated in 2, we discard some levels in the top part of the octree, because nodes in these level are too coarse. For convenience, we denote the number of levels in the top layer by $T$.

- **Branch Layer** consists of one level of nodes immediately below the top layer. The nodes in the branch layer can be regarded as the root of the corresponding subtrees. In each branch node, we store an index *idChild* addressing its first child node. If the node doesn't have child nodes, then *idChild* is set to $-1$.

  For convenience, we denote the depth of the branch layer by $B$.

- **Middle Layer** consists of the nodes in the levels below the branch layer except the bottom level.

  For each node in branch layer and the middle layer, we store the shuffled *xyz* key of the node center [9], and an index *idChild* addressing its first child node. The shuffled *xyz* key of a node at level $\ell$ is defined as a bit string: $x_1y_1z_1x_2y_2z_2...x_\ell y_\ell z_\ell$, where each 3-bit code $x_iy_iz_i$ encodes one of the eight subregions at level $i$. The shuffled key *xyz* is used to traverse the octree from the top to the bottom, and the index *idChild* are used to traverse the octree in the breadth-first order.

- **Data Layer:** consists of the nodes in the deepest level of the octree. In each data node, we store the shuffled *xyz* key of the node center, the signed distance of the node center to the scene, and the weight of the distance, which are defined as in [1].

We allocate a node buffer for each layer, as illustrated in Figure 2. The nodes in each layer is arranged in level by level, and two auxiliary arrays, **head** and **tail**, are used to store the addresses of the first and the last node of each level. Though the nodes in each layer is stored in an one dimensional buffer, we can use **oct**$[\ell][i]$ to represent the $i$-th node at level $\ell$ for convenience.

In the top layer and the branch layer, the node number is very small, so all nodes are stored in the corresponding array. However, in the middle layer and the data layer, some nodes are discarded, since their parents are not split during the reconstruction update step. The distribution of the nodes in these two layer is adaptive to scene surface.

**Algorithm 1** Split Nodes at Level $\ell$

```
1:  //step 1: predict which nodes need split
2:  splitFlag[] ← 0.
3:  for all node o_i at Level ℓ in parallel do
4:    if o_i is in the view frustum then
5:      c ← the center of node o_i
6:      p ← projection of c in the current image plane
7:      s ← query the surface point at pixel p
8:      sdf ← || p_camera − s || − || p_camera − c ||
9:      if o_i has no child and NeedSplit(o_i, sdf) then
10:       splitFlag[i] = 1
11:     end if
12:   end if
13: end for

14: //step 2: scan the split flag array
15: (shift, count) ← Scan(splitFlag, +)

16: //step 3: set child index and compute xyz key
17: for all node o_i at level ℓ in parallel do
18:   if splitFlag[i] == 1 then
19:     o_i.idChild ← tail[ℓ + 1] + (shift[i] ∗ 8)
20:     key ← o_i.xyzkey ∗ 8
21:     for k from 0 to 7 do
22:       addr ← o_i.idChild + k
23:       oct[ℓ + 1][addr].xyzkey ← key | k
24:     end for
25:   end if
26: end for

27: //step 4: update tail
28: tail[ℓ + 1] ← tail[ℓ + 1] + count ∗ 8
```

## 5. Reconstruction Update

### 5.1. Node Split

In order to fuse the point cloud of the current frame, we need split some nodes in the middle layer for the points that are not captured in the previous frames.

We traverse the octree from the branch layer in bread-first order, and adaptively split the nodes when needed. When a node is split, the id of its first child is computed and stored. Algorithm 1 shows the pseudo code of the node splitting procedure for level $\ell$.

In step 1, we predict whether a node needs further split in parallel. We first determine if it lies in the view frustum. If it is in, then we calculate the signed distance $d$ from its center to the current depth map (Line 5 ~ 8). In Line 7, the point surface **s** is queried in the surface measurement of the current depth map using pixel $p$. In Line 8, $\mathbf{p}_{camera}$ denotes the camera position of the

current frame. Then, we determine whether the node contains scene data and need split using function **NeedSplit**. We store an boolean value 1/0 in an auxiliary array $splitFlag[]$ to indicate whether the node need be split or not.

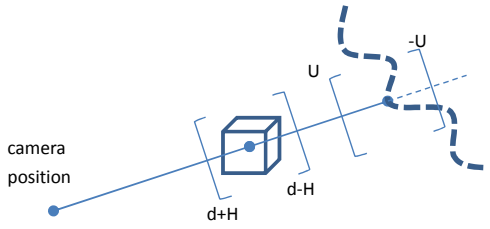



Figure 3: Node split prediction. $d$ is the signed distance from the node center to the estimated surface, $H$ is radius of the circumscribed sphere of the node, and $U$ is the maximum distance for distance truncation.

Function **NeedSplit** determines whether a node needs split. We assume the surface around the node is approximately the point **s**. As illustrated in Figure 3, let $d$ denote the distance between the node center and the estimated surface, and $H$ denote radius of the circumscribed sphere of the node. Then the distance from any point in the node to the surface can be estimated as interval $[d - H, d + H]$. Let $U$ be the maximum value to truncate the signed distance [1, 25], then the nodes with distance out of the range $[-U, U]$ is discarded, and doesn't need split. Therefore, **NeedSplit** function returns true if and only if intervals $[d - H, d + H]$ and $[-U, U]$ have overlaps.

Since the split prediction is based on the queried surface point **s** on the estimated surface, it will be not accurate enough when the node size becomes very large. Therefore, we start node split from the branch layer, instead of the top layer, and allocate all nodes in the top layer. The empty flag for the node in the top layer is determined according to if it has any split descendant in the branch layer.

In step 2, we adopt the parallel prefix scan primitive [28] with the add operation " + " on the $splitFlag[]$ array to compute an id for the child node, and store it in an array $shift[]$. Meanwhile, the number of split nodes is obtained and stored in a variable *count*.

In step 3, we assign index of its first child to the split node (Line 19), and compute the shuffled key *xyz* for each child node by the shuffled key of the split node (Line 21 ~ 24). Note that **tail**$[\ell + 1]$ stores the ending address of the nodes at level $\ell + 1$, and the newly generated child nodes are put at end of the buffer at level $\ell + 1$.

Finally, we increase the ending address of level $\ell + 1$ by the count of split nodes multiplied by 8.

### 5.2. Node Removal

**Algorithm 2** Remove Nodes at Level $\ell$

```
1:  count ← tail[ℓ − 1] − head[ℓ − 1]
2:  mergeFlag[1..count] ← 0
3:  //step 1: determine whose child nodes to merge
4:  for all nodes o_i at level ℓ − 1 in parallel do
5:      if o_i.idChild ≠ −1 and NeedRemove(o_i) then
6:          mergeFlag[o_i.idChild/8] ← 1
7:          o_i.idChild ← −1
8:      end if
9:  end for

10: //step 2: scan the merge flag array
11: (shift, count) ← Scan(mergeFlag, +)

12: //step 3: update the child index
13: for all node o_i at level ℓ − 1 in parallel do
14:     idc ← o_i.idChild
15:     if idc ≠ −1 and mergeFlag[idc/8] == 0 then
16:         o_i.idChild ← idc − shift[idc/8] ∗ 8
17:     else
18:         o_i.idChild ← −1
19:     end if
20: end for

21: //Step 4: Compact the nodes
22: Compact(oct[ℓ], mergeFlag, shift)

23: //Step 5: decrease the ending address at level ℓ
24: tail[ℓ] ← tail[ℓ] − count ∗ 8
```

When the scene is dynamic with moving objects, it's necessary to remove nodes where the dynamic object moves. Otherwise, the corresponding memory is wasted, and the system may run out of memory when the moving objects traverse everywhere of the scene volume.

Node removal is a reverse procedure of node split. We traverse the octree from data layer to the branch layer. Algorithm 2 shows the pseudo code of the node removal method at level $\ell$. We need not only determine which nodes to remove, but also re-assign child indices for the nodes at the parent level $\ell - 1$.

In step 1, we visit all nodes at level $\ell - 1$ in parallel, and check if all of its eight child nodes (at level $\ell$) are empty by the function **NeedRemove**. *NeedRemove* returns true if and only if all of its eight child nodes are empty. A child node is empty if and only if its idChild is

$-1$. However, when the child node is in the data layer, it has no idChild field. In this case, we use the signed distance value of the child node to determine if it is empty. If the distance is out of the rang $[-U, U]$, then it is out of the fusion range and is regarded as empty.

When a node can be merged, the corresponding $mergeFlag[]$ is set as *true* (Line 6), and the child index, *idChild*, is set as -1 (Line 7).

In step 2, we use the **Scan** primitive [28] on array $mergeFlag[]$ with operation + to get the index shift of level $\ell$ after node removal, and the result is stored in array $shift[]$. In addition, the number of removed nodes is stored in variable *count*.

In step 3, we update the child index for each node at level $\ell - 1$ in parallel. The condition in Line 15 says that node $\mathbf{o}_i$ has child nodes that will not be removed, for which the *idChild* of node $\mathbf{o}_i$ is decreased by the correspond value in $shift[]$ multiplied by 8. Otherwise its *idchild* is set as $-1$ to indicate that $\mathbf{o}_i$ has no child node.

In step 4, we compact the nodes at level $\ell$ according to array $mergeFlag[]$ and $shift[]$ in parallel. We use the **Compact** primitive [28, 29] to remove the marked nodes, so that there is no gap among the remained nodes.

At last, we decrease the ending address of level $\ell$ by $count * 8$.

### 5.3. Top Layer Update

The top layer is designed to help the ray tracer to skip large empty cells. Therefore the boolean value, $emptyflag$, of the nodes in the top layer need be updated accordingly after node splitting and/or node removal.

The updating procedure is similar to the **NeedRemove** in the node removal procedure. For each node in the top layer, we check the *idChild* of its child nodes: if all *idChild* are $-1$, then we set the *emptyFlag* of the node as true. This procedure is repeated from the bottom to the top in the top layer.

### 5.4. SDF Update

Once the node structure of the octree is updated, we can merge the current depth map into the scene reconstruction, which requires to update the nodes in the data layer, since the other nodes don't record any distance value to the scene.

We adopt the same fusion algorithm in [25]. As shown in Algorithm 3, we compute the coordinates of the leaf nodes according to their shuffled *xyz* key (Line 5). Then we calculate the signed distance (Line 5 to 8). At last, we update the weight (Line 11 to 13).

**Algorithm 3** SDF Update

1: **for** all node $\mathbf{o}_i$ in the data layer in parallel **do**
2: **if** node $\mathbf{o}_i$ is not in the view frustum **then**
3: return
4: **end if**
5: $\mathbf{c} \leftarrow$ the center point of node $\mathbf{o}_i$
6: $p \leftarrow$ projection of $\mathbf{c}$ in the current image plane
7: $\mathbf{s} \leftarrow$ query the estimated surface point at pixel $p$
8: $sdf \leftarrow \| \mathbf{p}_{camera} - \mathbf{s} \| - \| \mathbf{p}_{camera} - \mathbf{c} \|$
9: //$U$ is maximal distance for truncation
10: **if** $sdf \geq -U$ **then**
11: $sdf \leftarrow$ truncate $sdf$ against $[-U, U]$
12: $\mathbf{o}_i.sdf \leftarrow (\mathbf{o}_i.sdf * \mathbf{o}_i.w + sdf)/(\mathbf{o}_i.w + 1)$
13: $\mathbf{o}_i.w \leftarrow \min(\mathbf{o}_i.w + 1, w_{max})$
14: **end if**
15: **end for**

## 6. Surface Prediction

After updating the octree, we develop an octree-based ray tracer in GPU to estimate the scene surface in the current camera view.

### 6.1. Ray Tracing

For each pixel on the imaging plane, a ray is cast from the camera center of the current frame. In order to compute the intersection of the ray with the scene, we need find the two nodes in the data layer where the distance values have different signs. Algorithm 4 shows how we marched the ray in the octree to find the intersection point. In this algorithm, $\vec{\mathbf{r}}(\mathbf{e}, \vec{\mathbf{d}})$ denotes a ray originated at $\mathbf{e}$ in direction $\vec{\mathbf{d}}$, $\mathbf{p}_{in}$ denotes the point that the ray enters into node $\mathbf{o}_{in}$, and $\mathbf{p}_{out}$ denotes point that the ray leave from node $\mathbf{o}_{in}$. Note that $\mathbf{p}_{out}$ is also the point that the ray enters into the next node denoted by $\mathbf{o}_{out}$. When the signs of the distance values in $\mathbf{o}_{in}$ and $\mathbf{o}_{out}$ are different, we can compute the intersection point between the ray and the scene. Note that only the nodes in the data layer record the distance function.

This algorithm calls two functions, **FindNode** and **Intersect**. Function **FindNode** finds the node where a point lies in, and returns the node and its level id. Function **Intersect** computes the intersection point of a ray with the bounding box of a node. Since $\mathbf{p}_{in}$ and $\mathbf{p}_{out}$ are at the bounding box's surface, function **FindNode** and **Intersect** need to deal with critical cases. In order to avoid the critical cases, we translate $\mathbf{p}_{in}$ and $\mathbf{p}_{out}$ by a small offset $\epsilon$ in the ray direction, as shown in Line 7, 9, and 10.

While function **Intersect** is trivial, **FindNode** function requires to traverse the octree from the top to bottom, as shown in Algorithm 5.

The code between Line 2 ∼ 7 in this algorithm correspond to traverse above the branch layer, and the code between Line 8 ∼ 11 correspond to traverse below the branch layer.

**Algorithm 4** Surface Prediction

1: **for** each pixel $p$ in imaging plane in parallel **do**
2: $\quad \overrightarrow{\mathbf{r}}(\mathbf{e}, \overrightarrow{\mathbf{d}}) \leftarrow$ generate a ray toward pixel $p$
3: $\quad \mathbf{p}_{in} \leftarrow \mathbf{e}$
4: $\quad$ **if** $\mathbf{p}_{in}$ is not in the scene volume **then**
5: $\quad\quad \mathbf{p}_{in} \leftarrow$ **Intersect** $(\overrightarrow{\mathbf{r}}(\mathbf{e}, \overrightarrow{\mathbf{d}}), RootNode)$
6: $\quad$ **end if**
7: $\quad (\mathbf{o}_{in}, \ell_{in}) \leftarrow$ **FindNode**$(\mathbf{p}_{in} + \epsilon \cdot \overrightarrow{\mathbf{d}})$
8: $\quad$ **while** $\mathbf{p}_{in}$ is in the scene volume **do**
9: $\quad\quad \mathbf{p}_{out} \leftarrow$ **Intersect**$(\overrightarrow{\mathbf{r}}(\mathbf{p}_{in} + \epsilon \cdot \overrightarrow{\mathbf{d}}, \overrightarrow{\mathbf{d}}), \mathbf{o}_{in})$
10: $\quad\quad (\mathbf{o}_{out}, \ell_{out}) \leftarrow$ **FindNode**$(\mathbf{p}_{out} + \epsilon \cdot \overrightarrow{\mathbf{d}})$
11: $\quad\quad$ **if** $\ell_{in} = \ell_{out} = L$ **then**
12: $\quad\quad\quad$ // $L$ is the depth of the tree
13: $\quad\quad\quad$ **if** $\mathbf{o}_{in}.sdf \cdot \mathbf{o}_{out}.sdf < 0$ **then**
14: $\quad\quad\quad\quad$ // there is an intersection
15: $\quad\quad\quad\quad \mathbf{p} \leftarrow$ compute the intersection point
16: $\quad\quad\quad\quad \mathbf{n} \leftarrow$ estimate the normal at $\mathbf{p}$
17: $\quad\quad\quad\quad$ **break**
18: $\quad\quad\quad$ **end if**
19: $\quad\quad$ **end if**
20: $\quad\quad (\mathbf{p}_{in}, \mathbf{o}_{in}, \ell_{in}) \leftarrow (\mathbf{p}_{out}, \mathbf{o}_{out}, \ell_{out})$
21: $\quad$ **end while**
22: **end for**

**Algorithm 5** FindNode(p)

1: $x_1y_1z_1 \cdots x_Ly_Lz_L \leftarrow$ shuffled xyz key of point $\mathbf{p}$
2: **for** $\ell$ from $B - T$ to $B$ **do**
3: $\quad \mathbf{o}_\ell \leftarrow \mathbf{oct}[\ell][x_1y_1z_1 \cdots x_\ell y_\ell z_\ell]$
4: $\quad$ **if** node $\mathbf{o}_\ell$ is empty **then**
5: $\quad\quad$ return $(\mathbf{o}_\ell, \ell)$
6: $\quad$ **end if**
7: **end for**
8: **while** $\ell \neq L$ *and* node $\mathbf{o}_\ell$ has child node **do**
9: $\quad \mathbf{o}_\ell \leftarrow \mathbf{oct}[\ell][\mathbf{o}_\ell.idChild + x_\ell y_\ell z_\ell]$
10: $\quad \ell \leftarrow \ell + 1$
11: **end while**
12: return $(\mathbf{o}_\ell, \ell)$

### 6.2. Intersection Computing

Once we find two nodes with a positive distance and negative distance in the data layer, we compute the surface point using the linear interpolation method. As illustrated in Figure 4, $\mathbf{c}_1$ and $\mathbf{c}_2$ denotes the centers of these nodes. Let $d_1$ and $d_2$ denote that signed distance of these nodes, and $\mathbf{n}_1$ and $\mathbf{n}_2$ denote that normal function of these nodes (the normal at the node centers in the data layer can be obtained by computing the gradient of signed distance function). Then the surface point, $\mathbf{p}_s$, can be estimated as follows:

$$\mathbf{p}_s = \mathbf{c}_1 - \frac{d_1}{d_1 - d_2}(\mathbf{c}_1 - \mathbf{c}_2)$$

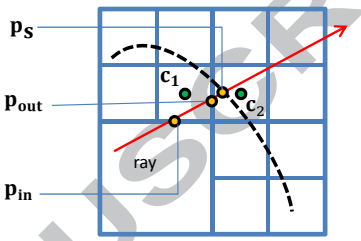



Figure 4: Intersection estimation by linear interpolation.

And the normal of the intersection point can be estimated similarly:

$$\mathbf{n}_s = Normalized\left[\mathbf{n}_1 - \frac{d_1}{d_1 - d_2}(\mathbf{n}_1 - \mathbf{n}_2)\right]$$

## 7. Experimental Results

We have implemented the proposed method in C++ with CUDA, and test it on an desktop computer with Intel Core2 Duo E7400 2.80GH CPU (we use only one core), 2GB RAM and a Nvidia GeForce GTX480 graphics card with 2GB GPU memory. And we adopt the **Scan** primitive in CUDPP [29] during octree generation. In our current implementation, the block size for parallelizing threads is 256.

All of the data, including the whole octree, captured depth maps, the predicted surface and the auxiliary arrays are stored in the global memory. For each layer in the octree, we pre-allocate a node buffer in the GPU memory. The buffer size should adapt to the depth of the layer in the octree, since the octree has more nodes in the lower level. For the octree at level $\ell$, we reserve the following number of nodes:

$$N_\ell = \begin{cases} 2^{3\ell}, & if \ \ell \leq B \\ \mu \cdot 2^{3\ell}, & if \ \ell > B \end{cases} \quad (1)$$

where $B$ is the level id of the branch layer, $\mu$ is user specified ratio. In our experiments, we set $\mu = 0.05$ for

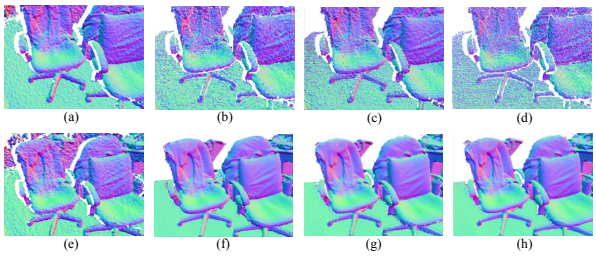



Figure 5: Left to right in each row: normal maps of raw data, $KF_{512}$'s result, $OF_9$'s result, and $OF_{10}$'s result, respectively. Top row are the normal maps of the first frame and bottom row are the normal maps of 346th frame.

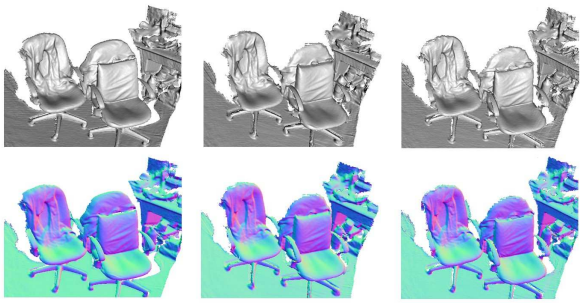


Figure 6: Reconstruction of the "chairs" scene. Top row are the Phong shading results and bottom are normal maps. The first column are $KF_{512}$, the middle column are $OF_9$ and the third column are $OF_{10}$.

Table 1: Memory consumption of the "chairs" scene (MB).

| $OF_9$ | $\ell$=7 | $\ell$=8 | $\ell$=9 | Aux | Total |
|---|---|---|---|---|---|
| PreAlloc | 8.0 | 6.4 | 51.2 | 16.0 | 81.6 |
| Used | 8.0 | 4.0 | 27.6 | 16.0 | 55.6 |

| $OF_{10}$ | $\ell$=7 | $\ell$=8 | $\ell$=9 | $\ell$=10 | Aux | Total |
|---|---|---|---|---|---|---|
| PreAlloc | 8.0 | 6.4 | 51.2 | 245.8 | 51.2 | 362.6 |
| Used | 8.0 | 3.0 | 19.8 | 145.4 | 51.2 | 227.4 |

$\ell \leq 9$, and $\mu = 0.03$ for $\ell > 9$, which work very well. When the pre-allocated buffer overflows, and there are remained memory, we can allocate another buffer, and regard them as a logically continuous memory.

In the next subsections, we present some results and comparison regarding the following aspects: memory consumption, reconstruction accuracy, computation time, and large scenes.

### *7.1. Memory Cost and Accuracy*

We first compare the memory cost and the accuracy between our method and KinectFusion. For KinectFusion, we set the grid resolution as $512 \times 512 \times 512$, and for our method, we set the depth of the octree as 9 and 10.

For convenience, we denote KinectFusion configuration with $M \times M \times M$ grid as $KF_M$, and denote our method with tree depth $L$ as $OF_L$.

Note that the effective voxel resolution $OF_9$ and $OF_{10}$ are $512 \times 512 \times 512$ and $1024 \times 1024 \times 1024$, which are equivalent to $KF_{512}$ and $KF_{1024}$.

Remember that memory for the branch layer is fully allocated, the level of the top layer should not be too great to avoid to much memory consumption. Meanwhile, we perform split prediction on the nodes in the branch layer, which require the size of the node be small enough to give accurate result. To compromise the two factors, we can choose the branch layer at level 5, 6, or 7. Since these levels all have small memory consumption and level 7 is more robust for split prediction, we set the branch layer at level 7 in all experiments.

As shown in Figure 5, we record a sequence of depth maps with 840 frames of a static scene "chairs", and compare the performance of $KF_{512}$, $OF_9$ and $OF_{10}$. The top row and the bottom row are the 1st and the 346-th frame. For the first frame, all of the reconstruction results are noisy since the normal estimation is not robust in the beginning of the scan. For the 346-th frame, all of these method achieved very good results.

The final reconstruction results are shown in Figure 6. The meshes are extracted by the marching cube method [30], which takes about 5s in GPU. Figure 7 (a) compares the reconstruction qualities by putting these meshes together. These result largely overlap in most regions, which shows that they have the same low frequency shape. Figure 7 (b,c,d) show some zoomed views of a detailed region. One can see that $OF_{10}$ achieves better results than $KF_{512}$ and $OF_9$, because it captures more rich details.

Figure 8 shows the memory consumptions of these methods. $KF_{512}$ takes 512 MB memory constantly, while $OF_9$ and $OF_{10}$ take more memory as more parts of the scene is scanned. At the beginning, $OF_9$ and $OF_{10}$ pre-allocate 81.6MB and 362.6MB, respectively. Eventually, $OF_9$ takes 55.6MB, and $OF_{10}$ takes 227.4MB. With $512^3$ resolution, memory consumption of $OF_9$ is $81.6/512 \approx 15.9\%$ of $KF_{512}$. If we set $1024^3$ resolution for KinectFusion, it would take 4GB memory, which is more than 10 times of $OF_{10}$. Table 1 shows the memory consumption in each level of the octree for $OF_9$ and $OF_{10}$.

Figure 9 shows the memory usage during scanning a dynamic scene with a moving chair. We empirically found that it is not necessary to do node removal at every

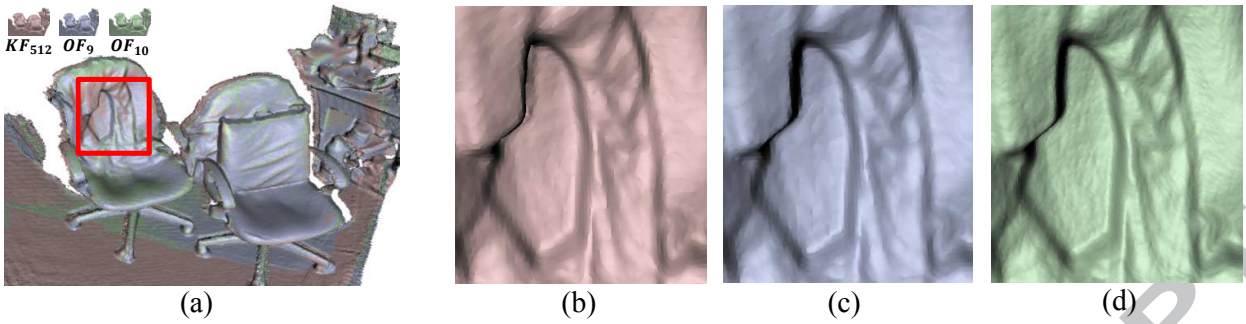



Figure 7: Results comparison between $KF_{512}$, $OF_9$ and $OF_{10}$. (a) compare the results by overlapping them together. (b) zoomed view by $KF_{512}$. (c) zoomed view by $OF_9$. (d) zoomed view by $OF_{10}$.

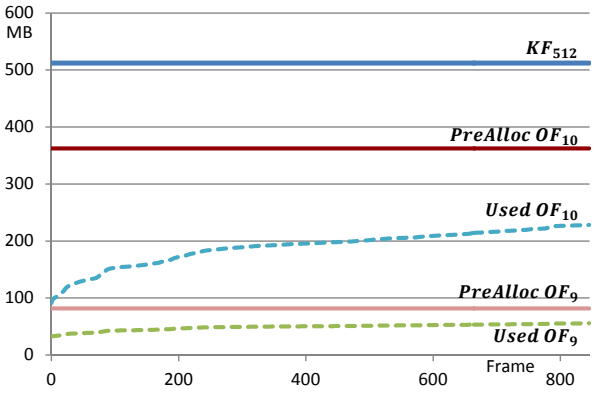



Figure 8: Memory comparison of the static scene "chairs".

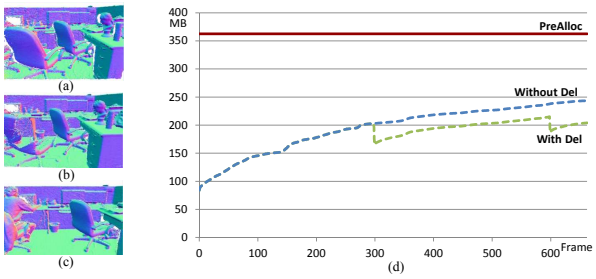



Figure 9: Memory consumption on a dynamic scene. (a)-(c): the 80-th, 175-th, and 663-th frame of the scene. A chair starts moving at the 175-th frame. (d): memory usage at each frame. At the 300-th and 600-th frame, node removal is performed.

frame. Instead, we perform it once for every 300 frames. The result shows that this method saves about 50MB memory, which effectively avoids memory overflow.

### *7.2. Performance Analysis*

Figure 10 shows a breakdown timing analysis for our method and KinectFusion, where the second subscript of our method denotes the number of levels in the top layer, and the method label with a star do node removal at every frame. In colored regions of the timing figure, we show the time cost in the surface measurement, camera pose estimation, reconstruction update and surface prediction from the bottom to the top.

From the results, we can see that

1. These methods spend the same time in the surface measurement step and the pose estimation step, since they use the same algorithms in these two steps.
2. Both $OF_9$ and $OF_{10}$ are faster than $KF_{512}$, which shows that our octree-based data structure is supe-

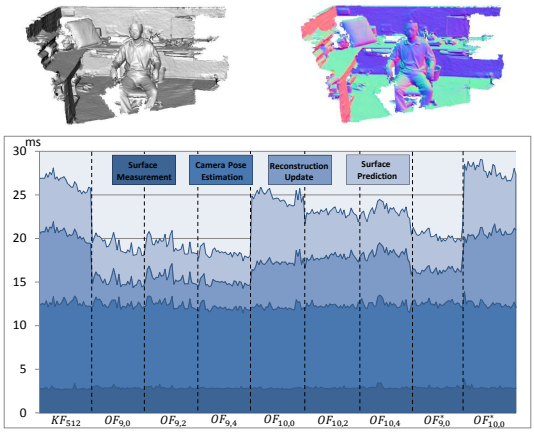
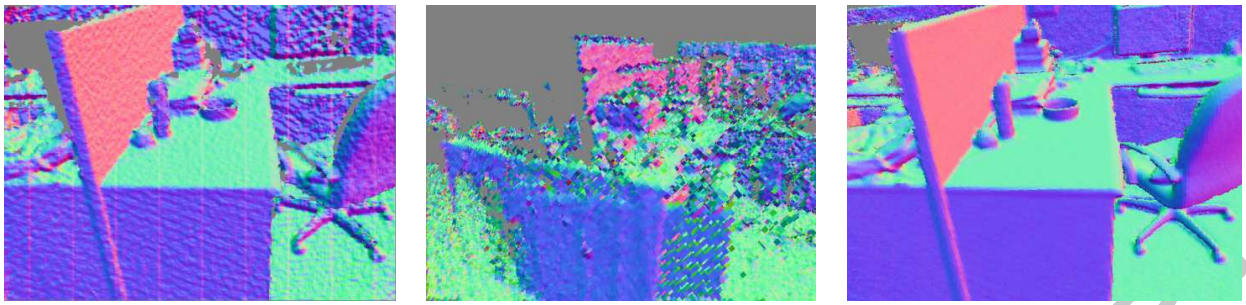



Figure 10: Processing time for a test scene. Top are the Phong-shaded rendering and the normal map of the test scene, respectively. Bottom is the processing time on each stage of different methods.



(a) captured depth map (b) $KF_{512}$ result (c) $OF_{10}$ result

Figure 11: The screenshots of reconstructing a scene in a $8m \times 8m \times 8m$ bounding box.

rior than the volumetric grid in KinectFusion. And $OF_9$ is faster than $OF_{10}$.

3. The top layer accelerates the performance of the ray-cast process by comparing the timings of $OF_{9,0}$ *vs* $OF_{9,2}$ *vs* $OF_{9,4}$ and $OF_{10,0}$ *vs* $OF_{10,2}$ *vs* $OF_{10,4}$.
4. The node removal operation will slightly slow down the performance, by comparing the reconstruction update timing of $OF_{9,0}$ *vs* $OF^*_{9,0}$ and $OF_{10,0}$ *vs* $OF^*_{10,0}$.

### *7.3. Large Scale Scene*

In Figure 11, we scanned a cubical office in an $8m \times 8m \times 8m$ bounding box using $OF_{10}$ and $KF_{512}$. Our method $OF_{10}$ spends about 2 minutes in processing 3800 frames of depth maps streamed from the depth camera, and takes about 299MB memory in a 363MB pre-allocated buffer.

Due to the memory limit, KinectFusion can only provide a $512 \times 512 \times 512$ resolution, which takes 512MB memory. Under this resolution, the size of the grid voxel is about 1.56cm. As shown in Figure 11 (b), this voxel size is too large for KinectFusion to correctly track the pose of the depth camera, and the reconstruction result is not acceptable.

The final extracted mesh contains about 6200K triangles and 3400K vertices. As shown in Figure 12, the final result has very rich surface details, including the folders of the pillow, the keys on the keyboard, the handlers of the drawer, and the small folders/belts of the bag.

## 8. Conclusions

We have presented a real time reconstruction method for dynamic scenes using a low cost depth camera and a commodity graphic hardware. The core of our method is an octree-based representation for storing the scene data, and we propose a set of algorithms to efficiently maintain and use the octree structure. Compared with original KinectFusion method, our method only requires about 10% memory and runs about 2× faster in the surface prediction step and the reconstruction update step. With the proposed method, we can scan much larger scenes using the same hardware setup.

There are several directions to improve this work. First, the current method pre-allocates the memory buffers according to Equation 1, where is the user need to determine the coefficient $\mu$. Since the optimal value for $\mu$ varies among different scenes, it is challenging for the user to determine the optimal coefficient. Second, we are interested in more efficient memory management solution to replace the pre-allocation method. Third, when the dynamic objects move fast enough, the camera tracking procedure may get lost, because the gap size between the adjacent frame is very large. In this case, we need find more additional information to correct the camera pose. At last, we can combine our method with the Kintinuous method [26], so that we further increase the scene scale. This combination will require inserting our octree structure into the Kintinuous method.

## Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments, and thank Bo Jiang, Zizhao Wu and Xuan Cheng for proof reading and video editing. This work was partially supported by NSFC (No. 60970074), China 973 Program (No. 2009CB320801), Fok Ying Tung Education Foundation and the Fundamental Research Funds for the Central Universities.

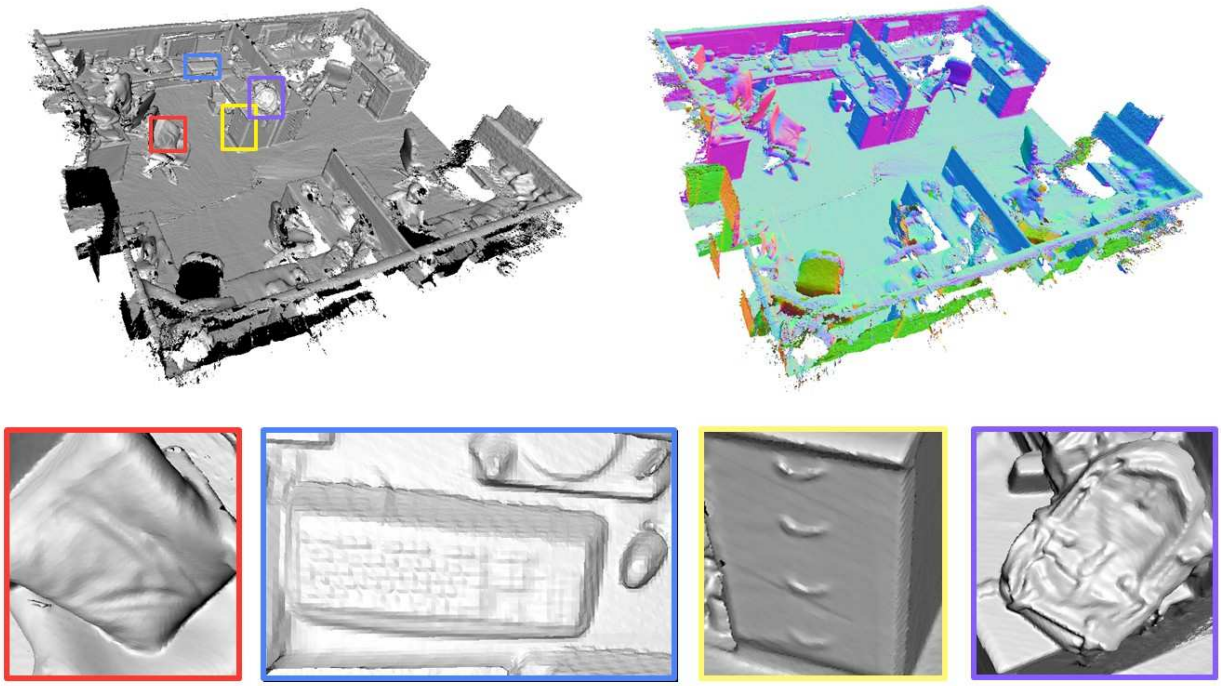


Figure 12: Reconstruction result of a large scene with an $8m \times 8m \times 8m$ bounding box. Top: a Phong-shaded rendering and a false color visualized normal map of the reconstructed result. Bottom: four zoomed views.

## References


[1] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohli, J. Shotton, S. Hodges, A. Fitzgibbon, Kinectfusion: Real-time dense surface mapping and tracking, in: Procedings of IEEE / ACM International Symposium on Mixed and Augmented Reality, pp. 127–136.

[2] K. Xu, Y. Li, T. Ju, S.-M. Hu, T.-Q. Liu, Efficient affinity-based edit propagation using k-d tree, ACM Trans. Graph. 28 (2009) 118:1–118:6.

[3] C. Lauterbach, M. Garl, S. Sengupta, D. Luebke, D. Manocha, Fast bvh construction on gpus, Computer Graphics Forum 28 (2009) 375–384.

[4] X. Sun, K. Zhou, E. Stollnitz, J. Shi, B. Guo, Interactive relighting of dynamic refractive objects, ACM Trans. Graph. 27 (2008) 35:1–35:9.

[5] M. Seiler, D. Steinemann, J. Spillmann, M. Harders, Robust interactive cutting based on an adaptive octree simulation mesh, Vis. Comput. 27 (2011) 519–529.

[6] J. Tian, W. Jiang, T. Luo, K. Cai, J. Peng, W. Wang, Adaptive coding of generic 3d triangular meshes based on octree decomposition, Vis. Comput. 28 (2012) 819–827.

[7] S. F. Frisken, R. N. Perry, A. P. Rockwood, T. R. Jones, Adaptively sampled distance fields: a general representation of shape for computer graphics, in: Proceedings of the 27th annual conference on Computer graphics and interactive techniques, SIGGRAPH '00, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000, pp. 249–254.

[8] C.-H. Shen, G.-X. Zhang, Y.-K. Lai, S.-M. Hu, R. R. Martin, Harmonic field based volume model construction from triangle soup, J. Comput. Sci. Technol. 25 (2010) 562–571.

[9] K. Zhou, M. Gong, X. Huang, B. Guo, Data-parallel octrees for surface reconstruction, IEEE Transactions on Visualization and Computer Graphics (TVCG) 17 (2011) 669–681.

[10] A. W. Fitzgibbon, A. Zisserman, Automatic camera recovery for closed or open image sequences, in: European Conference on Computer Vision, Springer-Verlag, 1998, pp. 311–326.

[11] M. Pollefeys, L. V. Gool, M. Vergauwen, F. Verbiest, K. Cornelis, J. Tops, R. Koch, Visual modeling with a hand-held camera, International Journal of Computer Vision 59 (2004) 207–232.

[12] D. Scharstein, R. Szeliski, A taxonomy and evaluation of dense two-frame stereo correspondence algorithms, INTERNATIONAL JOURNAL OF COMPUTER VISION 47 (2001) 7–42.

[13] A. J. Davison, Real-time simultaneous localisation and mapping with a single camera, in: IEEE International Conference on In Computer Vision, volume 2, pp. 1403–1410.

[14] G. Klein, D. Murray, Parallel tracking and mapping for small ar workspaces, in: Proceedings of the 2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality, ISMAR '07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 1–10.

[15] A. J. D. Richard A. Newcombe, Live dense reconstruction with a single moving camera, in: IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR), pp. 1498–1505.

[16] J. Stühmer, S. Gumhold, D. Cremers, Real-time dense geometry from a handheld camera, in: Proceedings of the 32nd DAGM conference on Pattern recognition, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 11–20.

[17] R. A. Newcombe, S. Lovegrove, A. J. Davison, Dtam: Dense

tracking and mapping in real-time, in: ICCV, pp. 2320–2327.

[18] S. Rusinkiewicz, O. Hall-Holt, M. Levoy, Real-time 3d model acquisition, ACM Trans. Graph. 21 (2002) 438–446.

[19] N. Gelfand, N. J. Mitra, L. J. Guibas, H. Pottmann, Robust global registration, in: Proceedings of the third Eurographics symposium on Geometry processing, SGP '05, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2005, pp. 197–206.

[20] Y. Chen, G. Medioni, Object modeling by registration of multiple range images, Image and Vision Computing (IVC) 10 (1992) 145–155.

[21] H. Li, R. W. Sumner, M. Pauly, Global correspondence optimization for non-rigid registration of depth scans, in: Proceedings of the Symposium on Geometry Processing, SGP '08, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2008, pp. 1421–1430.

[22] H. Li, B. Adams, L. J. Guibas, M. Pauly, Robust single-view geometry and motion reconstruction, ACM Trans. Graph. 28 (2009) 175:1–175:10.

[23] T. Weise, H. Li, L. Van Gool, M. Pauly, Face/off: live facial puppetry, in: Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, SCA '09, ACM, New York, NY, USA, 2009, pp. 7–16.

[24] T. Weise, S. Bouaziz, H. Li, M. Pauly, Realtime performance-based facial animation, ACM Trans. Graph. 30 (2011) 77:1–77:10.

[25] S. Izadi, D. Kim, O. Hilliges, D. Molyneaux, R. Newcombe, P. Kohli, J. Shotton, S. Hodges, D. Freeman, A. Davison, A. Fitzgibbon, Kinectfusion: real-time 3d reconstruction and interaction using a moving depth camera, in: Proceedings of the 24th annual ACM symposium on User interface software and technology, UIST '11, ACM, New York, NY, USA, 2011, pp. 559–568.

[26] T. Whelan, J. McDonald, M. Kaess, M. Fallon, H. Johannsson, J. J. Leonard, Kintinuous: Spatially extended kinectfusion, in: RSS Workshop on RGB-D: Advanced Reasoning with Depth Cameras.

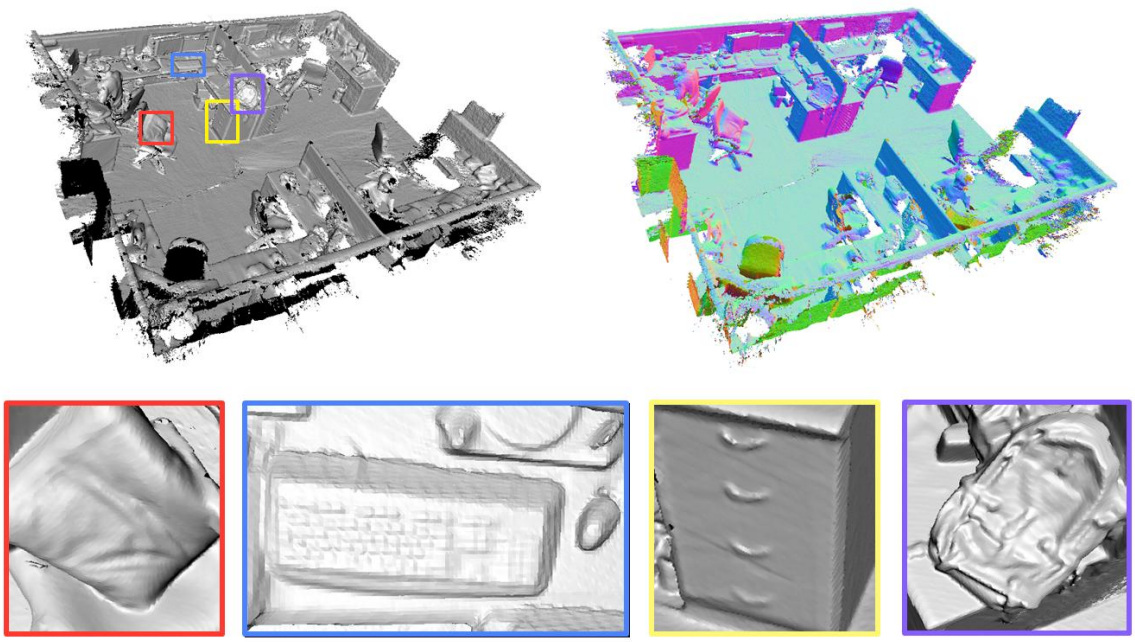
[27] M. Zeng, F. Zhao, J. Zheng, X. Liu, A memory-efficient kinectfusion using octree, in: Proceedings of Computational Visual Media 2012, p. to appear.

[28] M. Harris, S. Sengupta, J. D. Owens, Parallel prefix sum (scan) with CUDA, Addison Wesley, 2007. Chapter 39.

[29] M. Harris, J. D. Owens, S. Sengupta, Y. Zhang, A. Davidson, Cudpp homepage, 2007. Http://gpgpu.org/developer/cudpp.

[30] W. E. Lorensen, H. E. Cline, Marching cubes: A high resolution 3d surface construction algorithm, COMPUTER GRAPHICS 21 (1987) 163–169.

Graphical Abstract

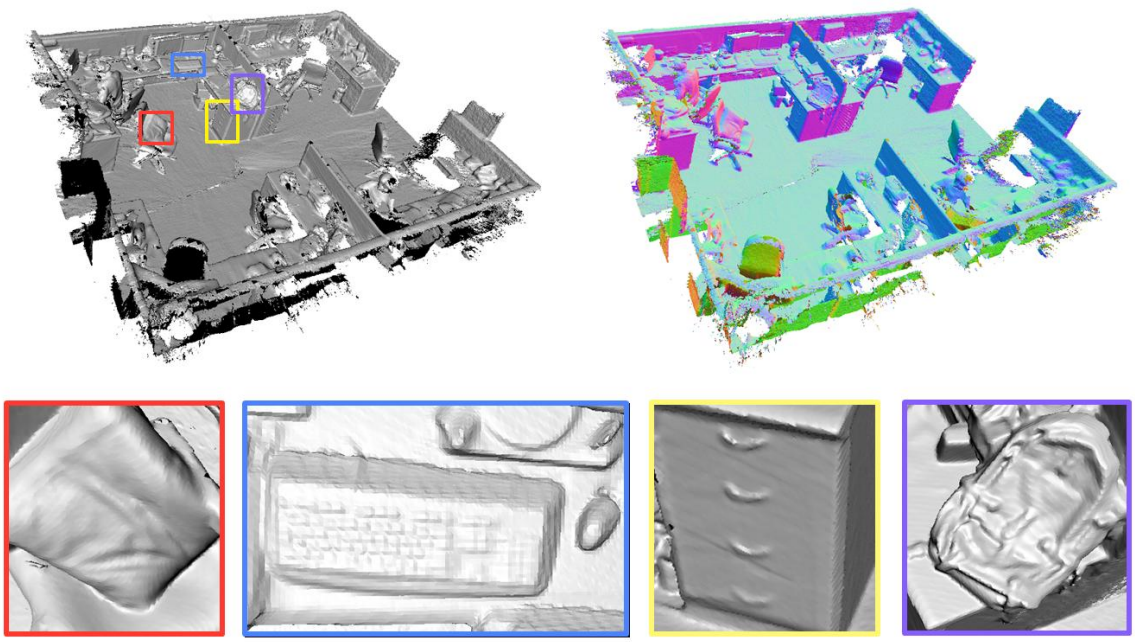


**Reconstruction result of a large scene, and four zoom-in parts.**

Research Highlights

1. We propose an octree based realtime 3D reconstruction algorithm using a consumer depth camera.
2. Our method is memory-efficient thanks to the octree based data structure.
3. Our method is fast due to both GPU parallelism and hierarchical structure.
4. We introduce complete operations on dynamically updating octree structure on GPUs, and octree based surface prediction.

Graphical Abstract



**Reconstruction result of a large scene, and four zoom-in parts.**

Research Highlights

1. We propose an octree based realtime 3D reconstruction algorithm using a consumer depth camera.
2. Our method is memory-efficient thanks to the octree based data structure.
3. Our method is fast due to both GPU parallelism and hierarchical structure.
4. We introduce complete operations on dynamically updating octree structure on GPUs, and octree based surface prediction.