TextureMe: High-Quality Textured Scene Reconstruction in Real Time

JUNGEON KIM, HYOMIN KIM, HYEONSEO NAM, JAESIK PARK, and SEUNGYONG LEE,
POSTECH, Korea

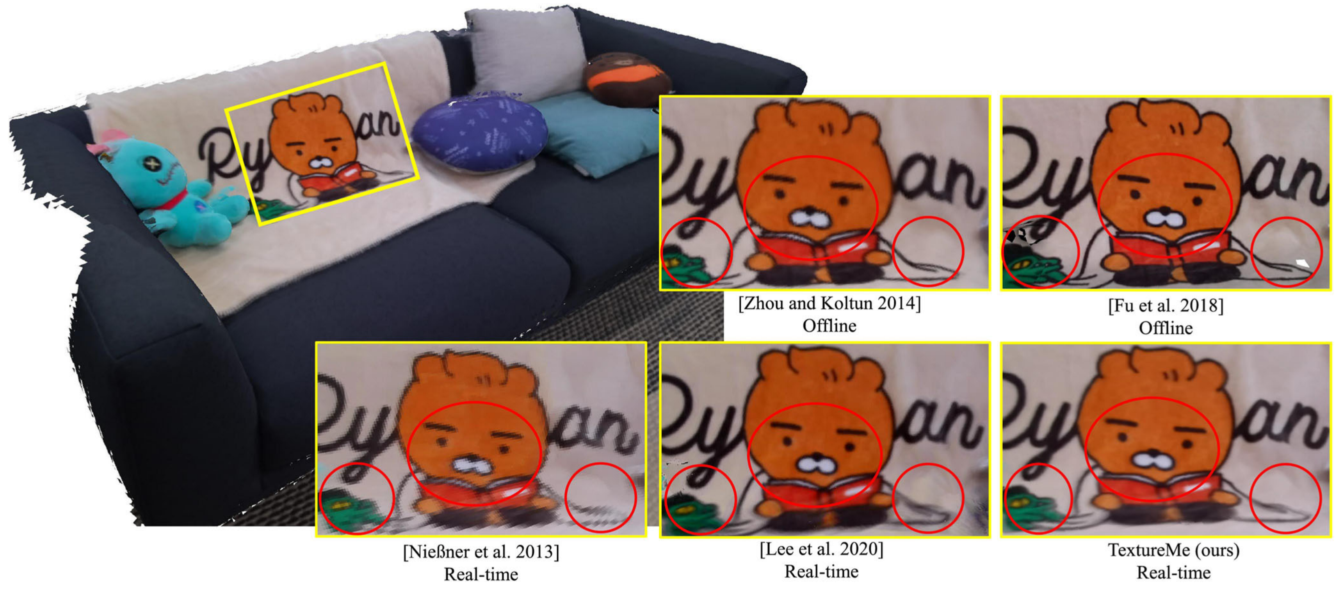

Fig. 1. Reconstruction result of *TextureMe* for 'Sofa' scene. *TextureMe* jointly recovers 3D geometry and high-quality texture in *real time*. The generated texture of the reconstructed model is much sharper than the result obtained from a per-voxel color fusion approach. Although our approach runs in real time, the reconstructed textures are even comparable to state-of-the-art texture mapping methods that run offline (see the supplementary video for a real-time demo).

Three-dimensional (3D) reconstruction using an RGB-D camera has been widely adopted for realistic content creation. However, high-quality texture mapping onto the reconstructed geometry is often treated as an offline step that should run after geometric reconstruction. In this article, we propose *TextureMe*, a novel approach that jointly recovers 3D surface geometry and high-quality texture in real time. The key idea is to create triangular texture patches that correspond to zero-crossing triangles of truncated signed distance function (TSDF) progressively in a global texture atlas. Our approach integrates color details into the texture patches in parallel with the depth map integration to a TSDF. It also actively updates a pool of texture patches to adapt TSDF changes and minimizes misalignment artifacts that occur due to camera drift and image distortion. Our global texture atlas representation is fully compatible with conventional texture mapping. As a result, our approach produces high-quality textures without utilizing additional texture map optimization, mesh parameterization, or heavy post-processing. High-quality scenes produced by our real-time approach are even comparable to the results from state-of-the-art methods that run offline.

CCS Concepts: • **Computing methodologies** → **Mesh models**; **Texturing;**

Additional Key Words and Phrases: Real-time, 3D reconstruction, texture mapping, single-view RGB-D

ACM Reference format:
Jungeon Kim, Hyomin Kim, Hyeonseo Nam, Jaesik Park, and Seungyong Lee. 2022. TextureMe: High-Quality Textured Scene Reconstruction in Real Time. *ACM Trans. Graph.* 41, 3, Article 24 (March 2022), 18 pages.
https://doi.org/10.1145/3503926

1 INTRODUCTION

Three-dimensional (3D) reconstruction has gathered widespread attention since the last decade. In particular, as RGB-D sensors and high-performance computing have become prevalent, effective algorithms for *real-time* reconstruction of 3D objects and scenes

This work was supported by the Ministry of Science and ICT, Korea, through IITP grants (SW Star Lab, 2015-0-00174; AI Innovation Hub, 2021-0-02068; Artificial Intelligence Graduate School Program (POSTECH), 2019-0-01906).
Authors' address: J. Kim, H. Kim, H. Nam, J. Park, and S. Lee, POSTECH, 77 Cheongam-Ro, Nam-Gu, Pohang, Gyeongbuk, Korea 37673; emails: {jungeonkim, min00001, hyeonseo.nam, jaesik.park, leesy}@postech.ac.kr.

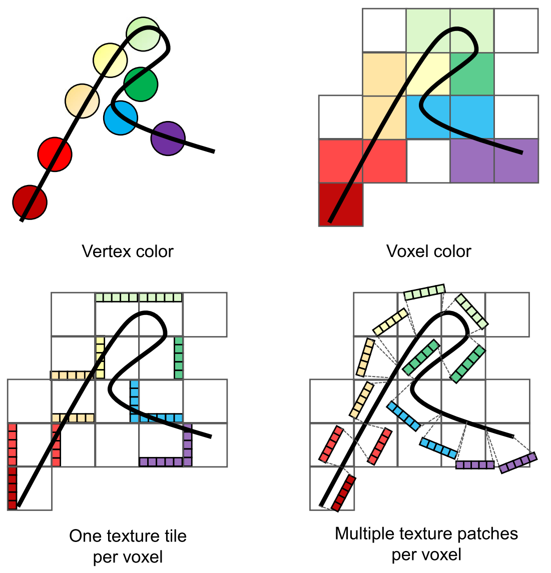
Vertex color
Voxel color
One texture tile per voxel
Multiple texture patches per voxel

Fig. 2. Illustration of different representations for coloring the geometry. The target surface is drawn as a black curve, and the stored color information of each representation is shown. Detailed color information can be lost with vertex or voxel color representation. TextureFusion [Lee et al. 2020] (bottom left) assigns a texture tile for a voxel. Our approach (bottom right) can assign multiple texture patches depending on the shape of the reconstructed surface.

have been developed. KinectFusion [Newcombe et al. 2011] is a representative real-time geometry reconstruction method. Follow-up studies extended the approach to be able to process large-scale scenes [Nießner et al. 2013; Whelan et al. 2012] and to reduce accumulated reconstruction errors [Dai et al. 2017b].

These reconstruction methods utilize volumetric representation, such as truncated signed distance function (TSDF), to update the recovered surface. In the reconstruction process, TSDF integrates noisy depth images and successfully produces a clean surface *geometry*. However, when surface *color* is required, the volumetric representation has a limited ability to restore detailed color information. In the conventional volumetric data structure, one voxel stores only one color value; representing a detailed color texture requires tiny voxels and would require significant memory.

To reconstruct a surface from a TSDF, the Marching Cubes algorithm [Lorensen and Cline 1987] has been widely adopted. It can produce multiple triangles from a single voxel. The triangles could represent detailed color, but conventional approaches encode the color by interpolating neighboring voxel colors. Consequently, the amount of surface color detail is constrained by the volume resolution. Modern depth cameras such as Kinect Azure [Microsoft 2020] can capture high-definition (HD) quality color images, but volumetric representation may miss useful color contents due to its insufficient spatial resolution.

Offline approaches based on texture mapping have been proposed to resolve the problem. In these approaches, a texture map is generated from high-quality color images after geometry reconstruction has been completed. Producing a global texture map by merging color images often induces misalignments due to inaccurate camera poses and image distortions. Prior methods attempted to solve the problem by optimizing camera poses [Zhou and Koltun 2014], texture map coordinates [Jeon et al. 2016], and image regions [Fu et al. 2018; Waechter et al. 2014]. These approaches generate high-quality texture maps, but they run *offline* with a substantial computational burden.

In this work, we propose a novel framework, which we call *TextureMe*, that jointly creates a high-quality texture map and the surface geometry *on the fly* from a sequence of RGB-D images in real time. The key idea is to maintain a special type of texture map, which we call a *global texture atlas*, together with a TSDF volume in the reconstruction process. A global texture atlas consists of right-angled triangles, which we call *texture patches*, that store color details of triangle faces of the reconstructed surface. We construct texture patches by mapping the triangles obtained from Marching Cubes onto the input color images. The texture patches are integrated on the global texture atlas to accumulate the surface color details.

TextureFusion [Lee et al. 2020] is a contemporary work to ours, and it achieves the same goal as our framework—building both geometry and texture map jointly in real time. TextureFusion associates a facet of each voxel to a single two-dimensional (2D) texture tile, and the texture of *only one* facet is maintained for each voxel (Figure 2). The method is fit for sufficiently small voxels. However, due to the diverse polygon configurations from Marching Cubes and improper assignment of a texture tile to a voxel facet, this scheme may yield inaccurate, blurry texture maps (see Figure 14). The issue would become more serious if the size of the voxel gets larger. On the other hand, our approach directly handles one or multiple triangles from a single voxel extracted by Marching Cubes. The triangles are seamlessly mapped to the global texture atlas, enabling our method to avoid the inaccurate texture map issue of TextureFusion.

Our approach is equipped with a few novel components to improve the visual quality of reconstructed colored scenes. We present a multi-scale optimization method for texture patch construction from the input color image to minimize misalignments among integrated texture patches. In addition, we introduce a hysteresis-based TSDF filter to prevent unnecessary connectivity changes of reconstructed triangles. By doing so, we can perform a stable texture patch update. We propose an effective resampling method to update the global texture atlas when the triangle connectivity changes. We also present a weighting scheme that takes into account effective information of new texture patches from the input color image. All of these elements are integrated to produce realistic colored scenes.

The contributions of this article are summarized as follows:

- We propose *TextureMe*, a novel real-time 3D reconstruction framework that generates a geometry model and a high-quality texture map simultaneously. With the framework, a texture-mapped scene can be immediately obtained on the fly without any post-processing, such as mesh parameterization or texture map optimization.
- Our TextureMe framework fully exploits the color information in the input color images by building and updating a *global texture atlas* during the reconstruction process.

- We introduce effective methods to handle misaligned texture patches, mesh connectivity changes, and texture patch updates. These methods help our framework to obtain a high-quality texture map.
- Our real-time system produces high-quality texture mapping results that are comparable to prior offline methods. We provide comprehensive comparisons with prior methods qualitatively and quantitatively.

2 RELATED WORK

Dense 3D surface reconstruction. In contrast to 3D reconstruction techniques that utilize complete 3D point clouds [Kazhdan and Hoppe 2013] or 2D images [Furukawa et al. 2010; Furukawa and Ponce 2010], online dense reconstruction methods integrate incoming depth frames incrementally to obtain complete models. The option of fusing depth maps in an explicit form such as a mesh [Turk and Levoy 1994] is not viable; thus, almost all online methods adopt volumetric representation [Curless and Levoy 1996]. KinectFusion [Newcombe et al. 2011] is the first real-time volumetric fusion system for 3D reconstruction. For each incoming depth map, it estimates the camera pose and updates TSDF on a regular volume grid. Kintinuous [Whelan et al. 2012] extended KinectFusion by conditionally moving the TSDF volume to recover larger scenes. Nießner et al. [2013] used voxel hashing to handle a large volume and relieve the computational burden of a hierarchical structure based on a GPU-based octree [Zhou et al. 2011].

As the scanning area gets larger, the accumulation error of estimated camera poses may increase. A representative offline method [Zhou et al. 2013] segments the input stream and reconstructs local fragments, which are then registered for loop closure [Choi et al. 2015]. Other recent methods perform loop closure online. ElasticFusion [Whelan et al. 2015] constructs a deformation graph and non-rigidly deforms surfel-based models. BundleFusion [Dai et al. 2017b] uses hierarchical pose optimization based on image features and dynamically updates the reconstructed scene on the fly.

Color mapping for 3D reconstruction. Several methods have been introduced to recover the color information of the reconstructed geometry. VoxelHashing [Nießner et al. 2013] provides colored scene reconstruction in real time. This approach assigns a single color to each voxel to reconstruct a colored model. Zhou and Koltun [2014] proposed an optimization-based color mapping method that corrects inaccurate camera poses by maximizing photometric consistencyand Maier et al. [2017] recover spatially varying spherical harmonics for colors, camera parameters, and camera poses together with geometry. These approaches utilize the vertices of a dense surface to represent color information, and the quality would be degraded for sparse models.

As another line of work, texture mapping methods that are not restricted to the density of surface geometry have been proposed. In these methods, a texture map is generated by mosaicking subregions of color images, in which the subregions are selected by solving a labeling problem [Fu et al. 2018; Gal et al. 2010; Huang et al. 2017; Lempitsky and Ivanov 2007; Li et al. 2018; Waechter et al. 2014] or by using heuristics that consider view direction [Bernardini et al. 2001; Velho and Sossai Jr 2007]. The seams on image region boundaries are reduced by a few color correction methods, including gradient-domain image composition [Pérez et al. 2003]. Another approach utilizes texture blending. Jeon et al. [2016] proposed an optimization-based method that updates texture coordinates to reduce misalignments of subtextures to be merged. All of these methods operate offline; they first need to select high-quality keyframes from the input color image sequence and then perform substantial optimization.

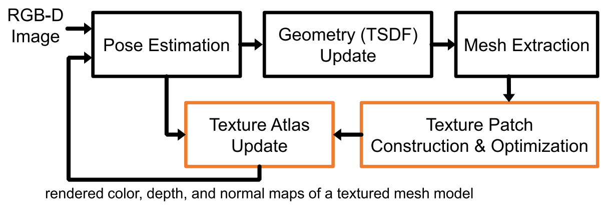
RGB-D Image
Pose Estimation
Geometry (TSDF) Update
Mesh Extraction
Texture Atlas Update
Texture Patch Construction & Optimization
rendered color, depth, and normal maps of a textured mesh model

Fig. 3. Overview of our TextureMe framework.

There exist a great deal of image-based rendering methods based on light fields [Buehler et al. 2001; Chen et al. 2002; Gortler et al. 1996; Levoy and Hanrahan 1996], projective texture mapping [Debevec et al. 1998], and patch match optimization [Bi et al. 2017]. They synthesize imagery from a novel viewpoint without generating a texture atlas, in which a proxy geometry can be used for better quality of synthesized images. Recently, deep-learning approaches have been introduced for novel view synthesis [Chen et al. 2018; Huang et al. 2020; Liu et al. 2020; Mildenhall et al. 2020; Thies et al. 2019]. In these approaches, production of high-fidelity imagery requires dense image samples. Xu et al. [2019] proposed a deep convolutional neural network that synthesizes a novel view from sparse, wide-baseline image samples. A few methods that use deep learning also achieve geometry reconstruction in addition to novel view synthesis from images [Liu et al. 2020; Mildenhall et al. 2020]. All of these methods need a reconstructed model that is used as a proxy geometry, or heavy computation for optimization or deep network training, making them run offline.

TextureFusion [Lee et al. 2020] is a contemporary work to ours, achieving the same goal of building both geometry and texture map jointly in real time. TextureFusion updates geometry and a texture map in real time without explicit mesh extraction. With the assumption that a voxel size is sufficiently small, texture information of one voxel facet is maintained for each voxel. To resolve color misalignment, they proposed spatially varying perspective warping. In contrast, our method constructs and updates triangular texture patches that correspond to triangle faces extracted from Marching Cubes. As a result, the produced texture map from our approach can be properly applied to the geometry regardless of voxel size (see Figure 14). To handle texture misalignment, we efficiently optimize the texture sampling coordinates in a multi-scaled image domain.

We present high-quality colored scene reconstructions using an HD quality RGB-D camera. The results are qualitatively and quantitatively evaluated using evaluation metrics designed for 3D reconstruction.

3 SYSTEM OVERVIEW

TextureMe (Figure 3) performs 3D geometry reconstruction and texture map generation simultaneously in real time.

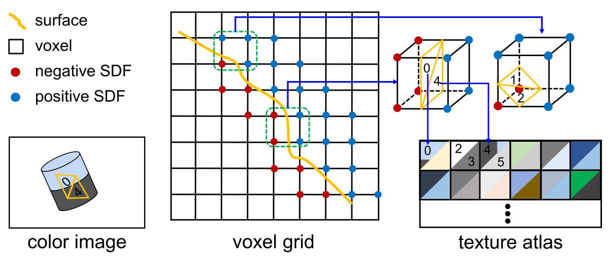

Fig. 4. Texture patch representation. When a depth image is integrated to update the TSDF in the voxel grid (middle), voxels containing the surface generate triangle faces using Marching Cubes. In our pipeline, each triangle face is mapped onto a texture patch in the global texture atlas (right). A texture patch takes rich color content from the color image (left) and puts it in the global texture atlas.

The geometry reconstruction module of TextureMe is built upon KinectFusion [Newcombe et al. 2011], which reconstructs a complete 3D model by integrating noisy input depth images. KinectFusion uses a variant of ICP [Low 2004], which takes only geometric alignment into account. However, we observed that using additional color information could relieve color misalignments. Similarly to Park et al. [2017] and Whelan et al. [2015], we adopt a Colored ICP that penalizes the intensity differences between a rendering of the reconstructed model and an input color image. For an RGB-D frame in the input sequence, the reconstruction module performs camera pose estimation to obtain the current camera pose $\mathbf{Q}_k$. The module then updates the TSDF volume for surface reconstruction using the input depth image $\mathbf{D}_k$ and the estimated pose $\mathbf{Q}_k$.

To construct texture patches, a mesh model of the reconstructed geometry is necessary. TSDF implicitly represents geometry during reconstruction; thus, we use the Marching Cubes algorithm [Lorensen and Cline 1987] to extract a mesh $\mathcal{M}_k$. After obtaining triangle patches from each voxel that contains zero-crossings, we map the triangles onto the input color image $\mathbf{C}_k$ using $\mathbf{Q}_k$ to obtain the corresponding color details, then construct texture patches for the current frame. We use the texture patches to update the global texture atlas maintained so far. The updates are performed by adaptively averaging the texture patches from the current frame with the corresponding texture patches in the global texture atlas.

The entire process of geometry and texture updates repeats for every frame to produce a fully textured mesh model. In our framework, the intermediate mesh model at each frame can be fully textured with the global texture atlas reconstructed until then, and the mesh model evolves in terms of coverage and quality through the reconstruction process.

The core of TextureMe is to effectively represent, store, construct, and update texture patches that constitute the global texture atlas. The representation and storage of texture patches are detailed in Section 4. The construction and update of texture patches, as well as handling of mesh connectivity changes, are covered in Section 5. Misalignments among texture patches can be caused by inaccurate pose $\mathbf{Q}_k$, rolling shutter, or imperfect synchronization of depth and color sensors, and should be resolved before texture patch update. We present an optimization method for the step in Section 6.

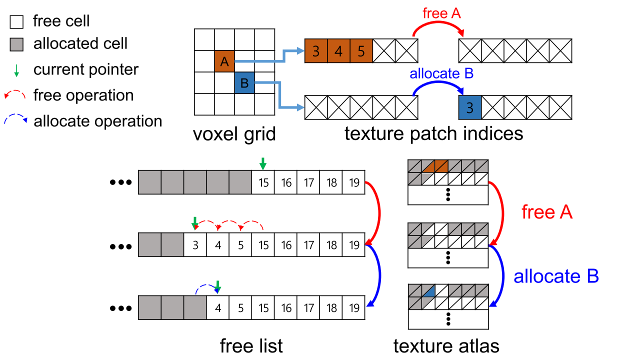

Fig. 5. Dynamic storage management for texture patches. Consider a case in which texture patches of voxel A are not needed anymore and voxel B needs to get one. Three texture patches (index: 3, 4, 5) for voxel A are released, and B receives one of the released texture patches (index: 3). The assigned texture patch indices are stored in the voxel grid. The free list shown below maintains ready-to-use texture patch indices. Once a texture patch index has been released, it is added to the free list (free operation). Then, it can be reallocated to another voxel when needed (allocate operation).

4 TEXTURE ATLAS REPRESENTATION

For real-time processing, all of the main components of TextureMe are performed on a GPU. This section presents an efficient data structure for our global texture atlas to maintain texture patches during the reconstruction process.

4.1 Texture Patch and Texture Atlas Representation

Texture patch. Our approach creates and progressively updates texture patches as the reconstruction procedure advances. A *texture patch* is a triangular region in the texture domain that stores color details for the corresponding triangle face of the reconstructed model. In our implementation, texture patches are maintained as triangles that reside in a 2D global texture atlas on the GPU side. They are stored as right-angled triangles for efficient memory usage.

Texture atlas. A texture atlas is a large image that stores a set of right-triangled texture patches (Figure 4). Before the reconstruction process begins, we preallocate memory on the GPU for a texture atlas. At the same time, by uniformly partitioning the texture atlas into right-angled triangles, we precalculate UV coordinates of vertices of each texture patch in the texture atlas. We predefine the minimal size of a texture patch that can store the possible maximal texture details from input color images. To be specific, we calculate the size of a right-angled triangle for a texture patch by projecting the size of a voxel onto the color image plane, where the voxel is distant from the RGB-D camera by the minimum depth that the camera can estimate. Each triangle face of the reconstructed final mesh $\mathcal{M}$ has a unique corresponding texture patch in the texture atlas.

4.2 Storing and Managing Texture Patches

We generate triangle faces from zero-crossing voxels and assign them unique indices of texture patches. The maximum number of triangles n_{tri} that can be extracted from a voxel is bounded due to the characteristics of Marching Cubes, where n_{tri} is five in our implementation. If n_v denotes the total number of voxels in the volumetric space used for reconstruction, a naïve implementation would be to allocate memory for a texture atlas to store $n_{tri} \times n_v$ texture patches. However, this approach wastes a massive amount of GPU memory because the portion of voxels crossing the reconstructed surface is quite small. In addition, most of them would generate fewer than n_{tri} triangles. Our goal is to manage the resources effectively so that memory space for the texture atlas fits in a budget during reconstruction.

We use a free list to manage storage for texture patches effectively. The free list is a stack that contains indices of unoccupied texture patches in the texture atlas. At the outset, we fill all possible texture patch indices ($[0, 1, \dots n-1]$) into the free list, where n is the maximum number of texture patches that can fit into the texture atlas. When the storage for a new texture patch is requested, we POP an index from the free list. When a texture patch no longer needs to be maintained in the texture atlas, we release the storage and PUSH its index to the free list. PUSH and POP operations are performed with an atomic GPU operation. Figure 5 illustrates the mechanism.

During the reconstruction process, new texture patches must be created when a voxel gets involved with zero-crossings the first time at a frame. Then, we create and store the texture patches in the texture atlas, where the texture patch indices are maintained in the voxel. On the other hand, texture patches are released from a voxel when the voxel is revisited with zero-crossings at a later frame but different numbers, types, or both of texture patches are needed for the voxel. Note that the numbers, types, or both of extracted triangles may change in Marching Cubes depending on the configurations of zero-crossings. In that case, we release old texture patches with the indices maintained in the voxel and create new texture patches based on the current zero-crossings. Our simple but effective resampling method from old to new texture patches will be explained in Section 5.4. If the types of texture patches needed for a revisited voxel remain the same as before, we keep and update the texture patches in the texture atlas.

5 TEXTURE PATCH FUSION

This section elaborates on how to construct and update the rich color contents of texture patches stored in the texture atlas using an input color image. We first construct new texture patches by mapping triangles extracted from the updated TSDF volume onto the color image. Then, we adaptively fuse the new texture patches from the current frame with the existing texture patches maintained in the texture atlas. We use a weighting scheme reflecting the blurriness of a color image for texture patch fusion. In addition, we introduce effective solutions to handle mesh connectivity changes that induce complications on texture patch fusion.

5.1 Constructing Texture Patches from the Current Frame

Let $\mathcal{M}_k$ be the mesh model extracted from the TSDF volume using Marching Cubes at the current frame k. We construct a texture patch $\mathbf{T}'_k$ for each triangle in $\mathcal{M}_k$. By projecting a triangle in $\mathcal{M}_k$ onto the current color image $\mathbf{C}_k$ using the estimated camera pose $\mathbf{Q}_k$ and camera calibration parameters, we can determine the corresponding triangle on $\mathbf{C}_k$ that contains the color information for texture patch $\mathbf{T}'_k$. We use barycentric coordinates to map the color details from $\mathbf{C}_k$ onto $\mathbf{T}'_k$. For a position inside $\mathbf{T}'_k$, we obtain the corresponding position on $\mathbf{C}_k$ by using the same barycentric coordinates.

5.2 Updating Texture Patches in the Texture Atlas

A new texture patch $\mathbf{T}'_k$ has a corresponding texture patch $\mathbf{T}_{k-1}$ in the global texture atlas if the number and types of texture patches for the voxel that contains $\mathbf{T}'_k$ remain the same at the current frame k. Otherwise, we resample the texture patch $\mathbf{T}_{k-1}$ corresponding to $\mathbf{T}'_k$ from the texture atlas, as will be described in Section 5.4. In both cases, texture patch $\mathbf{T}_{k-1}$ contains the color details accumulated up to the previous frame $k-1$. To smoothly fuse color information, we perform weighted averaging of $\mathbf{T}'_k$ and $\mathbf{T}_{k-1}$.

Texture patch update is defined as

$$\mathbf{T}_k = \frac{\mathbf{W}_{k-1} \odot \mathbf{T}_{k-1} + \mathbf{W}'_k \odot \mathbf{T}'_k}{\mathbf{W}_{k-1} + \mathbf{W}'_k}, \tag{1}$$

where, for simplicity, we omit texture coordinates and consider element-wise multiplication and division. $\mathbf{T}_k$ denotes the texture patch in the texture atlas after update. $\mathbf{W}_k$ is its weight map that contains a scalar weight for each texel in $\mathbf{T}_k$. $\mathbf{W}_k$ is defined as

$$\mathbf{W}_k = \min(\mathbf{W}_{k-1} + \mathbf{W}'_k, w_{max}), \tag{2}$$

where min is the element-wise minimum operator. w_{max} is a predefined maximum weight and $\mathbf{W}'_k$ is the weight map for $\mathbf{T}'_k$. We perform this update with each texture patch $\mathbf{T}'_k$ constructed from the current mesh $\mathcal{M}_k$.

The validity of color information in $\mathbf{T}'_k$ is related to the normal direction, the depth from the camera, and the texture sharpness. To reflect these factors, we define the weight map $\mathbf{W}'_k$ for $\mathbf{T}'_k$ as

$$\mathbf{W}'_k(\mathbf{p}) = \mathbf{w}_n \cdot \mathbf{w}_d \cdot \mathbf{w}_b, \tag{3}$$

where $\mathbf{p}$ denotes a 2D pixel position inside $\mathbf{T}'_k$. Let $\mathbf{q}$ be the 3D position that corresponds to $\mathbf{p}$ within the triangle of $\mathcal{M}_k$ whose texture patch is $\mathbf{T}'_k$. Then, $\mathbf{w}_n$ is defined as $\max(\mathbf{n} \cdot \mathbf{v}, \epsilon_n)$, where $\mathbf{n}$ and $\mathbf{v}$ denote the normal vector and the view direction of 3D position $\mathbf{q}$, respectively. ϵ_n is a fixed small positive value to prevent unfilled texture patches caused by noisy normal vectors. $\mathbf{w}_d$ is inversely proportional to the depth of $\mathbf{q}$ from the camera, as $\mathbf{w}_d = exp(-\alpha d_n^2)$, where $d_n = \min(\max(\frac{d-d_{min}}{d_{max}-d_{min}}, 0), 1)$ and α is a fixed positive value. d is the depth of $\mathbf{q}$. d_{min} and d_{max} are constant values that normalize the depth range.

Camera motion blur frequently occurs in real-time scanning. If a blurry texture patch has a large weight in texture patch update using Equation (1), the updated texture patch would also become blurry. To avoid this artifact, we make the weight small for any texture patch extracted from a blurry color image. We compute the blurriness using a perceptual blur metric [Crete et al. 2007]. The value of blurriness b_k is between 0 and 1, and a smaller value means a sharper image. Then, $\mathbf{w}_b$ is defined as $\mathbf{w}_b = 1 - \frac{1}{1+exp(\gamma(\omega_k - b_k))}$, where ω_k is the reference blurriness. We use $\omega_k = \mu_k - \sigma_k$, where

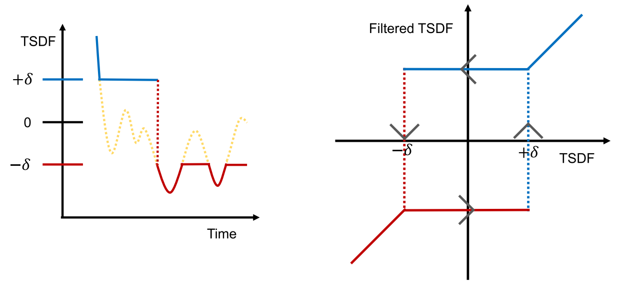

Fig. 6. Hysteresis-based TSDF filtering. (Left) Yellow dotted curve shows that an original TSDF value of a voxel fluctuates during geometry reconstruction. Blue and red curves indicate hysteresis-based filter output. The filter produces output according to whether and from which direction the original TSDF value has entered the wavering region ($|x| < \delta$). In this example, TSDF value enters the wavering region from above; thus, the output of our filter is $+\delta$. The output remains the same until the original TSDF value becomes equal to or less than $-\delta$. Once it has gone down into $x < -\delta$, the filter switches the output of the wavering region to $-\delta$. (Right) Visualization of the filter in terms of input and output.

μ_k and σ_k are the mean and the standard deviation of the blurriness values from the color images of previous frames. Weight $\mathbf{w}_b$ becomes 0.5 when b_k equals ω_k and rapidly decreases as b_k gets bigger. Parameter γ controls the decay speed of the weight; we use $\gamma = \frac{\beta}{\sigma_k}$ in our experiments, where β is a scalar constant.

5.3 Reducing Mesh Connectivity Changes

In practice, a depth image is often noisy, even if a denoise filter is applied. The depth noise can influence TSDF to be wavering. If the TSDF value of a voxel oscillates across zero value, the number and types of triangles extracted from Marching Cubes tend to frequently change as well. We design a hysteresis-based TSDF filter to stabilize the mesh connectivity variation. The filter behaves as an identity function when the absolute value of the input TSDF value is equal to or larger than δ. Otherwise, the filter outputs δ or $-\delta$ according to the history of the input TSDF values. Figure 6 illustrates the filter.

Specifically, let t_k and t_{k-1} be the original TSDF values of a voxel at the current k and the previous $k-1$ frames, respectively. Let t'_k and t'_{k-1} be the filter output for t_k and t_{k-1}, respectively. Then, t'_k is determined by

$$t'_k = \begin{cases} t_k & \text{if } |t_k| \geq \delta \\ +\delta & \text{if } |t_k| < \delta \text{ and } t_{k-1} \geq +\delta \\ -\delta & \text{if } |t_k| < \delta \text{ and } t_{k-1} \leq -\delta \\ t'_{k-1} & \text{otherwise } (|t_k| < \delta \text{ and } |t_{k-1}| < \delta) \end{cases} \quad (4)$$

5.4 Resampling Texture Patches

If the number, types, or both of triangles extracted from a voxel have changed, the set of texture patches maintained in the voxel needs to be replaced with a new set. We handle this case by resampling the color information of the new texture patches from the global texture atlas reconstructed so far. The key idea is rendering a snapshot of the current texture-mapped reconstructed model using the estimated camera pose $\mathbf{Q}_k$ of the current frame k. Then, color information for the new texture patches can be sampled from the rendered image in the same way as the texture patch construction described in Section 5.1 except that the rendered image replaces the input color image $\mathbf{C}_k$. We can also resample the weights of the new texture patches in a similar way by rendering texture patch weights stored in the global texture atlas as well as color information. In this manner, the mesh model with a changed set of surface triangles can inherit the same texture information from the previous frame.

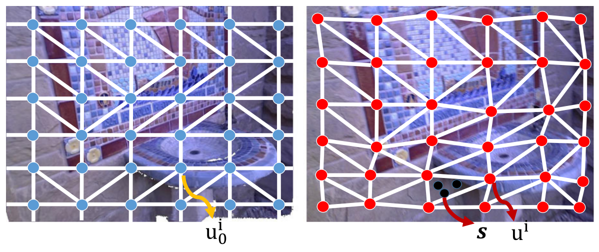

Fig. 7. Image warping for texture patch optimization. We create a regular grid on both a rendered color image $\mathbf{C}^r_k$ (left) and an input color image $\mathbf{C}_k$ (right), and generate a planar textured mesh by connecting grid points. We optimize the vertex positions of the planar mesh on $\mathbf{C}_k$ to warp $\mathbf{C}_k$ onto $\mathbf{C}^r_k$. Red circles show optimized vertex positions. Note that the mesh and the sample points $\{\mathbf{s}\}$ are shown for visualization only. In practice, we use much denser sample points for optimization.

6 TEXTURE PATCH OPTIMIZATION

In practice, camera pose and estimated geometry cannot be perfectly accurate. Such inaccuracy may incur misalignment of texture information in texture patch update using Equation (1). Input color images themselves also often suffer from lens distortions, rolling-shutter artifacts, and imperfect color and depth image synchronization. In this section, we introduce an optimization method to adjust the texture patches used for texture patch update to be more resilient to these challenges.

6.1 Image Warping Field

In the ideal case with perfect camera pose estimation and no imaging artifacts, an input color image $\mathbf{C}_k$ and the rendered image $\mathbf{C}^r_k$ of the textured reconstructed model should match each other in the overlapped regions. However, in practice, due to camera drifts and image distortions, $\mathbf{C}_k$ and $\mathbf{C}^r_k$ can become slightly misaligned. We obtain an image warping field that warps $\mathbf{C}_k$ onto $\mathbf{C}^r_k$ to resolve the misalignments and then use the warping field to determine optimized sampling positions of texture patches $\mathbf{T}'_k$ to be used for texture updates.

Similarly to Zhou and Koltun [2014], we use a regular grid to define a warping function. For efficient computation on GPU, we regard the input image $\mathbf{C}_k$ and the rendered image $\mathbf{C}^r_k$ as textured planar meshes, where updated vertex positions define a warping function. We make a virtual planar mesh by connecting grid points to make triangles as shown in Figure 7. Among the two possible options to form triangles on a grid cell, we select the one that minimizes the depth differences in the resulting triangles. Then, the variables for optimization are 2D vertex positions of the mesh overlaid on $\mathbf{C}_k$.

6.2 Objective

We formulate an objective that minimizes photometric discrepancy between a warped color image $\mathbf{C}_k$ and the rendered color image $\mathbf{C}_k^r$ of the reconstructed model. To calculate photometric errors between the two 2D textured meshes, inspired by the work of Jeon et al. [2016], we use intensities of $\mathbf{C}_k^r$ and $\mathbf{C}_k$ at the selected locations $\{\mathbf{s}\}$ on the image plane. A location $\mathbf{s}$ corresponds to predefined barycentric coordinates in a 2D mesh triangle.

The objective for image warping is defined as

$$E_{tex}(\mathbf{U}) = E_{data}(\mathbf{U}) + \lambda_{reg} E_{reg}(\mathbf{U}), \tag{5}$$

where $E_{data}(\mathbf{U})$ evaluates photometric errors, and $E_{reg}(\mathbf{U})$ preserves edge lengths in a 2D mesh. Optimization translates 2D original vertex positions $\mathbf{U}_0 = \{u_0^i\}$ of the mesh overlaid on $\mathbf{C}_k$ to optimized ones $\mathbf{U} = \{u^i\}$. The data term is written as

$$E_{data}(\mathbf{U}) = \sum_{v \in \mathbf{V}_M} \sum_{f \in \mathbf{F}_v} \sum_{\mathbf{b} \in \mathbf{B}_f} \left(\mathbf{C}_k \left(\mathbf{s}(\mathbf{b}, \{u^f\}) \right) - \mathbf{C}_k^r \left(\mathbf{s}(\mathbf{b}, \{u_0^f\}) \right) \right)^2, \tag{6}$$

where $\mathbf{V}_M$ is the set of vertices of the planar mesh on $\mathbf{C}_k$, $\mathbf{F}_v$ is the set of faces that contain vertex v, $\mathbf{B}_f$ is the predefined set of barycentric coordinates to densely sample intensities in the images, $\{u^f\}$ is the set of vertex positions of a triangle f, and $\mathbf{s}(\mathbf{b}, \{u^f\})$ is a sampled position in an image corresponding to the barycentric coordinates $\mathbf{b}$ in f. u_0^f indicates initial positions before optimization. The regularization term is defined as

$$E_{reg}(\mathbf{U})) = \sum_{v \in \mathbf{V}_M} \sum_{v' \in \mathbf{N}_1(v)} \left(l(u^v, u^{v'}) - l(u_0^v, u_0^{v'}) \right)^2, \tag{7}$$

where v' is a vertex in the 1-ring neighborhood $\mathbf{N}_1(v)$ of v, $l(u^v, u^{v'})$ is the Euclidean distance between u^v and $u^{v'}$, and $l(u_0^v, u_0^{v'})$ is the distance between the two vertices before optimization.

6.3 Efficient Optimization

The objective is formulated as non-linear least squares; thus, we apply the Gauss-Newton method to optimize $\mathbf{U}$. In practice, solving a large linear system of Jacobian matrix requires a highly efficient linear system solver. Instead of constructing Jacobian using Equation (5), similarly to Jeon et al. [2016], we exploit the locality of the problem that can be readily parallelized on GPU.

To be specific, we optimize $u^i \in \mathbf{U}$ of a vertex v_i independently from other parameters $u^j \in \mathbf{U} \setminus u^i$. When optimizing u^i of vertex v_i using Equation (5), we regard parameters u^j of the 1-ring neighborhood of v_i as constant. Therefore, for each vertex v_i, only $n \times 2$ Jacobian matrix and $n \times 1$ residual need be calculated, where n is the number of sample points $\{\mathbf{s}\}$ inside the 1-ring faces of vertex v_i. Then, we obtain a step vector Δu^i for updating u^i by solving the linear system from Jacobian and the residual.

We update $u^i \in \mathbf{U}$ of every vertex v_i in parallel on the GPU. We iterate this update process until it approaches the predefined maximum iteration number or the energy of the objective converges. To achieve real-time performance, we use CUDA to fully parallelize computation of Jacobian and residual, as well as linear system solving, for each vertex.

6.4 Multiscale Optimization

The computation time of texture patch optimization depends on the size of the regular grid laid on $\mathbf{C}_k$, because the size determines the number of unknowns u^i in $\mathbf{U}$. The time is also affected by the amounts of texture misalignments as $\mathbf{U}$ are incrementally updated through iterations. If texture misalignments are large relative to the grid cell size on $\mathbf{C}_k$, the number of iterations would increase. However, if we use a large grid cell to reduce the iterations, small texture misalignments could not be resolved. To handle this trade-off, we introduce a multiscale approach to our texture patch optimization.

We construct (1) Gaussian pyramids of the input color image $\mathbf{C}_k$ and the rendered image $\mathbf{C}_k^r$ and (2) a hierarchy of regular grid meshes corresponding to the image pyramids. The same triangle size is used for the grid mesh at each level of the pyramids. In this way, a triangle at a coarser level covers a larger region on the original image.

With this configuration, we optimize the objective (Section 6.2) from coarse to fine levels progressively. For each level l of the pyramid except for the coarsest level, we set initial vertex positions $\mathbf{U}_{0,l}$ of the grid mesh by interpolating the previously optimized ones $\mathbf{U}_{l-1}^{opt}$ at level $l-1$. Starting from these refined vertex positions, $\mathbf{U}_l$ at level l can be optimized with only a few iterations.

6.5 Optimized Texture Patch Sampling

We use the computed warping field that aligns $\mathbf{C}_k$ onto $\mathbf{C}_k^r$ for optimized sampling of new texture patch $\mathbf{T}_k'$ in Section 5.1. Initially, the sampling positions for $\mathbf{T}_k'$ on $\mathbf{C}_k$ are determined by projecting the corresponding triangle face in the current 3D mesh $\mathcal{M}_k$. Let x_0 be the initial 2D position of a projected vertex v on $\mathbf{C}_k$. We have a 2D warping vector $u_{opt}^i - u_0^i$ for each vertex of the 2D grid mesh used for image warping. Then, the updated position x of the projected vertex v is determined using the 2D warping vectors. Specifically, let f be the triangle face of the 2D grid mesh that contains x_0. We calculate the barycentric coordinates of x_0 inside face f and use the barycentric coordinates as the weights for interpolating the 2D warping vectors of three vertices of f. The interpolated 2D warping vector is then added to x_0 to obtain the updated position x.

Finally, the updated vertex positions of the projected triangle are used to sample color information for $\mathbf{T}_k'$ from $\mathbf{C}_k$. As a result, misalignments are reduced between texture patches $\mathbf{T}_k'$ and $\mathbf{T}_{k-1}$ that are used for texture update in Section 5.2.

7 EXPERIMENTS

7.1 Dataset and Implementation Details

Dataset. To evaluate our TextureMe system, we recorded several medium-sized indoor and outdoor scenes by using an Azure Kinect DK [Microsoft 2020]. Most existing datasets have color and depth image resolutions of the same size, whereas the Azure Kinect RGB-D camera supports various color and depth image resolutions. Among them, we utilized 1,080 p for color images and 576 p for depth images to capture high-quality color images. The numbers of frames in our scene data range from 600 to 2,400 (30 fps). We perform additional experiments for several scenes

Table 1. Timing of the Components in Our TextureMe System on Some Scenes in Different Datasets (Ours, 3D Scene [Zhou and Koltun 2014], and ScanNet [Dai et al. 2017a]), and with Different Voxel Sizes (1 cm, 2 cm, and 4 cm)

Unit: milliseconds

Scene	Image resolution		Geometry module		Texture module			Others	Total
	RGB	Depth	Pose	TSDF	Construction	Optimization	Update		
Our dataset, Sofa	1920 × 1080	640 × 576	4.7 ± 0.8	3.9 ± 0.9	8.6 ± 1.6	9.1 ± 0.8	9.2 ± 1.9	4.1 ± 0.4	39.6 ± 4.6
3D Scene, Fountain	640 × 480	640 × 480	4.3 ± 0.7	2.0 ± 0.4	5.4 ± 0.8	7.6 ± 0.7	3.4 ± 0.7	2.6 ± 0.4	25.2 ± 1.9
ScanNet, Scene0261_02	1296 × 968	640 × 480	4.6 ± 0.7	2.3 ± 0.8	5.7 ± 1.4	7.9 ± 0.8	6.9 ± 1.7	4.5 ± 0.9	32.1 ± 4.1

Scene	Voxel size	Geometry module		Texture module			Others	Total
		Pose	TSDF	Construction	Optimization	Update		
Our dataset, Sofa	1 cm	4.7 ± 0.8	3.9 ± 0.9	8.6 ± 1.6	9.1 ± 0.8	9.2 ± 1.9	4.1 ± 0.4	39.6 ± 4.6
Our dataset, Sofa	2 cm	4.3 ± 0.9	1.6 ± 0.4	3.0 ± 0.7	8.2 ± 0.7	6.0 ± 1.5	3.1 ± 0.5	26.1 ± 2.7
Our dataset, Sofa	4 cm	4.3 ± 0.8	0.9 ± 0.2	1.6 ± 0.3	8.2 ± 0.7	3.3 ± 0.7	3.0 ± 0.4	21.3 ± 1.4

Image sizes (in pixels) for datasets are also presented. Pose and TSDF denote Colored ICP and TSDF update on the volume (Section 3), respectively. Construction in the texture module includes mesh extraction using Marching Cubes and building connections between mesh triangles and texture patches in the global texture atlas (Section 4). Optimization indicates texture patch optimization (Section 6). Update denotes texture patch update (Section 5). Others include extra computations such as blurriness score calculation, image pyramid generation for texture patch optimization, and rendering vertex and normal maps. The numbers indicate the average times in milliseconds for handling a single RGB-D frame.

Table 2. Timing of Other Online Systems ([Nießner et al. 2013] and [Lee et al. 2020]) on Some Scenes in Different Datasets (Refer to Table 1 for the Scene Information), with Different Voxel Sizes (1 mm, 4 mm, 1 cm, 4 cm)

Unit: milliseconds

Scene	[Nießner et al. 2013]	[Lee et al. 2020]
Sofa	7.1 ± 0.8	60.8 ± 3.9
Fountain	4.9 ± 0.7	28.2 ± 4.3
Scene0261_02	5.4 ± 0.7	30.1 ± 3.2

Scene	Voxel size	[Nießner et al. 2013]	Voxel size	[Lee et al. 2020]
Sofa	1 mm	42.3 ± 2.3	4 mm	60.8 ± 3.9
	4 mm	7.1 ± 0.8	1 cm	36.4 ± 4.2
	1 cm	6.4 ± 0.9	4 cm	53.7 ± 2.3

The numbers indicate the average times in milliseconds for handling a single RGB-D frame. All numbers in the left sub-table are calculated using the 4 mm voxel size, except that the number of 'Scene0261_02' for Lee et al. [2020] is evaluated with the 1cm voxel size due to GPU VRAM limitation. The right sub-table shows the average times with varying voxel size in the 'Sofa' scene. Nießner et al. [2013] always met real-time speed with the 4 mm voxel size, whereas Lee et al. [2020] slowed when using small voxels (4 mm) in scenes with high-resolution color images, such as 'Sofa.'

from public datasets such as the 3D Scene dataset [Zhou and Koltun 2013, 2014] and ScanNet [Dai et al. 2017a].

Parameters. We used the following parameters for all of the results in this article. For texture patch update, $w_{max} = 5$, $\epsilon_n = 0.0001$, $\alpha = 3$, $\beta = 3$, $d_{min} = 0.35$, and $d_{max} = 3.4$. For hysteresis-based TSDF filter, $\delta = 0.1$cm. For texture patch optimization, we set the width of a regular grid cell to 50 and determine the height by following the aspect ratio of the color image. The number of levels in the Gaussian image pyramid $L = 3$, and the numbers of the max iteration for the Gauss-Newton optimization are set to 5, 4, 4 from the coarsest to the finest levels. The sample points $\{s\}$ inside triangles that constitute the regular grid cells are sampled every two pixels horizontally and vertically. Last, $\lambda_{reg} = 0.001$.

Seamless rendering. In our framework, the spatial adjacency in the generated texture atlas does not indicate spatial adjacency on the reconstructed 3D model. Therefore, adjacent texture patches may contain different contexts. If we render a reconstructed model with its texture atlas by a typical texture mapping method, bleeding artifacts [Carr and Hart 2002] may be observed on the boundaries of texture patches. To resolve the problem, we introduce a gutter space on the boundary of each texture patch. This space can be filled by fetching a slightly larger triangular patch than the real projected one from a color image. When the reconstructed model is texture-mapped, the gutter space prevents blending of adjacent texture patches in the texture atlas.

Timing. We used a PC equipped with an Intel i7-8700K 3.7GHz CPU, 64 GB RAM, and a single NVIDIA Titan RTX GPU having 24 GB VRAM. We fully utilize both NVIDIA CUDA API and OpenGL API to implement our TextureMe system. With our implementation, the average processing time for geometry reconstruction is 6~9 ms per frame. Our texture fusion module has three components: surface triangle and texture patch construction, texture patch optimization, and texture patch update in the texture atlas. The computation time of each component is proportional to the number of faces extracted at the current viewpoint, the frequency of mesh connectivity change, the size of a texture patch, and the color image resolution. For surface triangle construction, we perform Marching Cubes. To reduce the computation for Marching Cubes, our system extracts mesh triangles from zero-crossing voxels only within the current view frustum. In Table 1, we report the computation time of the respective component according to different types of datasets, where 1-cm voxel size is used. The table also shows the computation time changes of each component according to different voxel sizes. Among the aforementioned main factors of texture module computation time, three (number of extracted faces, frequency of mesh connectivity change, and size of

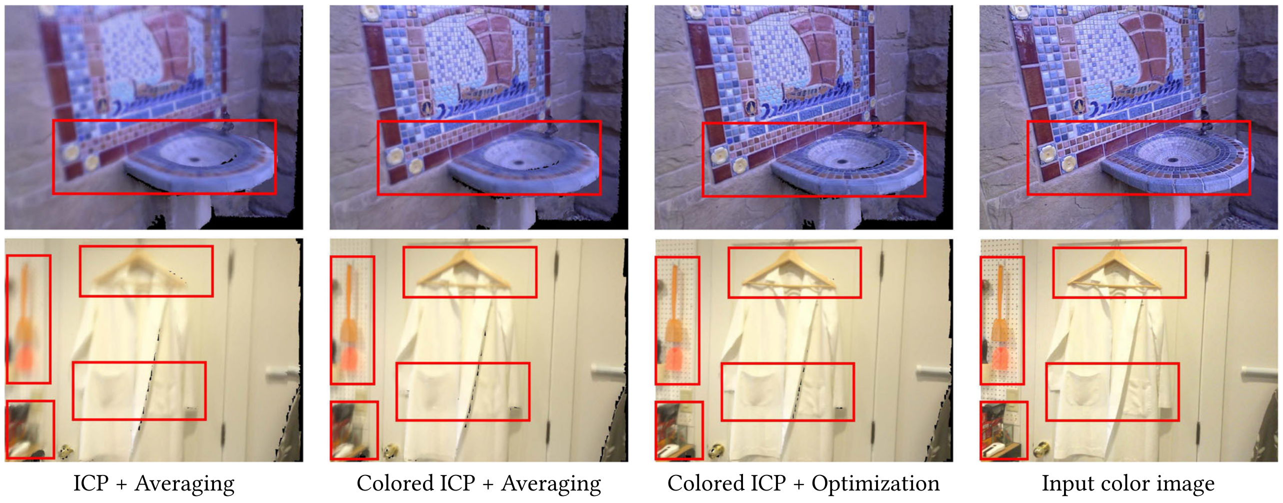

Fig. 8. Ablation study on our system with different pose estimation schemes (ICP and Colored ICP) and texture update methods (Averaging and Optimization). The images are rendered from textured geometries obtained by three approaches (ICP + Averaging, Colored ICP + Averaging, and Colored ICP + Optimization). ICP + Averaging produces blurry textures. Colored ICP resolves global camera drifts to some extent, but it cannot relieve spatially varying misalignments. Our approach indicated as Colored ICP + Optimization produces better results. For these results, we use fountain in the 3D Scene dataset [Zhou and Koltun 2014] and scene0261_02 in the ScanNet dataset [Dai et al. 2017a].

a texture patch) are strongly correlated with voxel size. The computation time of texture patch construction increases as the voxel size decreases because the number of managed voxel-to-texture-patch mappings increases. Using larger voxels increases memory locality of generated texture patches and decreases the total number of updated texels in the global texture atlas during scanning, reducing the computing time for texture patch update. In contrast, texture patch optimization is less affected by voxel size variation, because the optimization is performed in the image domain.

7.2 Analysis

Effect of Colored ICP. We use Colored ICP to estimate the camera pose that minimizes the point-to-plane errors and the photometric errors simultaneously (Section 3). Global color misalignments between an input image and the rendered image are considerably reduced by utilizing color information (Figure 8).

Effect of texture patch optimization. Colored ICP effectively reduces global color misalignments, but spatially varying misalignments remain due to various distortions (Figure 8). We handle the problem by computing a warping function to align an input color image to the rendered color image of the reconstructed model textured with the global atlas. Figure 9 demonstrates that our multi-scale texture patch optimization can reduce large and small misalignments between the input color image and the rendered color image with only a few iterations of optimization. Figure 9 also shows the effect of the multi-scale approach compared with the single-scale one. To obtain the optimization results (Figures 9(c), 9(d)), we used the same iteration number for both approaches. Computation of the single-scale optimization took about 1.7 ms longer than the multi-scale approach. Some iterations of the multi-scale approach run at coarse scales and take fewer computations than in single-scale optimization. With our multi-scale approach, we can effectively reduce misalignments between texture patches involved in texture fusion, which is hardly resolved even with globally consistent pose estimation approaches, such as BundleFusion [Dai et al. 2017b]. Qualitative comparison with BundleFusion is shown in Figure 10. As a result of texture patch optimization, the textured model keeps sharper and clearer color information.

Resilience to fast camera motion. To test the robustness of our system in fast camera motion, we generated two differently sampled sequences from the 'Fountain' scene, in which the sequences consist of RGB-D images sampled every 4 or 16 frames, respectively. As shown in the left image of Figure 11, our system can recover a sharp texture well in four-times-fast camera motion. However, when estimating entirely wrong camera poses due to too-fast camera motion, our system fails, as in the right image of Figure 11. Overall, our system works stably and reconstructs sharp textures in various scenes under modest camera pose accuracy (Figures 15, 16, and 18).

Effect of hysteresis-based TSDF filtering. Our hysteresis-based TSDF filter is designed to reduce mesh connectivity changes. It helps stable accumulation of color information onto texture patches by reducing the number of texture patch resamplings. However, an excessively high δ could cause a non-smooth reconstructed mesh as a side effect. Based on this observation, we selected $\delta = 0.1$ cm to balance the qualities of texture and geometry. Figure 12 demonstrates the impact of the filtering.

Less sensitivity to voxel size. Compared with the volumetric color fusion approach, the obvious benefit of our approach is that the texture mapping result is less affected by the voxel size, as shown in Figure 13. This is a natural consequence of our pipeline in that texture patches are maintained for voxels while volumetric color fusion stores single color values. Since VoxelHashing [Nießner et al. 2013] still has spare computational resources for real-time processing when using the 4-mm voxel size, we performed an

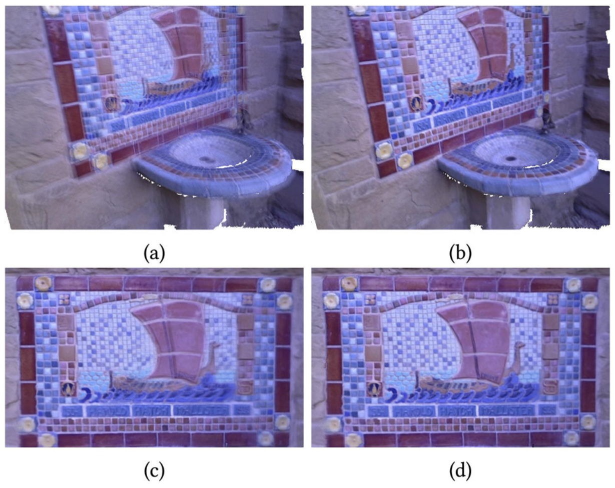

Fig. 9. Texture patch optimization. Given an input image C_k and the rendered image C_k^r of the reconstructed mesh, let C_k^w be the warped input image obtained by our multi-scale texture patch optimization. (a) is $(C_k + C_k^r)/2$ and shows that only improving the camera pose using color information cannot handle spatially varying distortions. (b) is $(C_k^w + C_k^r)/2$ and shows that our texture patch optimization helps avoiding a blurry texture map. (b) looks sharper as the input image is properly aligned with the rendered image. (c) and (d) show the final textured models obtained using the single-scale and multi-scale approaches in texture patch optimization, respectively. The multi-scale approach (d) corrects various texture patch misalignments more effectively than the single-scale one (c).

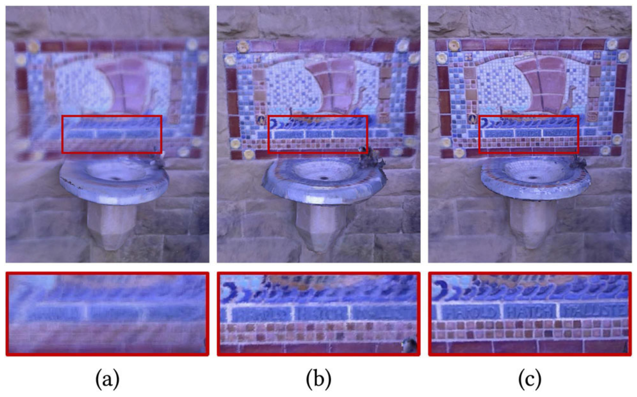

Fig. 10. Final reconstructed models of (a) VoxelHashing [Nießner et al. 2013] and (b) BundleFusion [Dai et al. 2017b] using 4-mm voxels. (c) Our TextureMe result using 1-cm voxels. Although BundleFusion provides more accurate camera poses than VoxelHashing, there are slight color misalignments caused by various image distortions. On the other hand, our TextureMe can effectively relieve the misalignments, resulting in a better result than BundleFusion.

additional experiment using the 1-mm voxel size, at which the average per-frame computation time of VoxelHashing is about 42 ms on the 'Sofa' scene (Table 2). As shown in Figure 13, the reconstructed model of VoxelHashing using 1-mm voxels shows sharper and clearer colors than using 4-mm voxels. However, these small voxels generate a highly dense reconstructed model that may be unsuitable for real-time applications. Furthermore, as

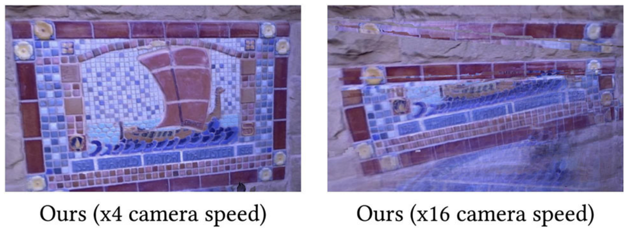

Fig. 11. Resilience of our TextureMe to fast camera motions. To obtain the reconstruction results, we used two differently sampled sequences from the 'Fountain' scene. While our optimization scheme can successfully recover a sharp texture in a four-times accelerated camera motion (left), it fails when camera motion is accelerated by 16 times (right), which causes erroneous camera-pose estimates.

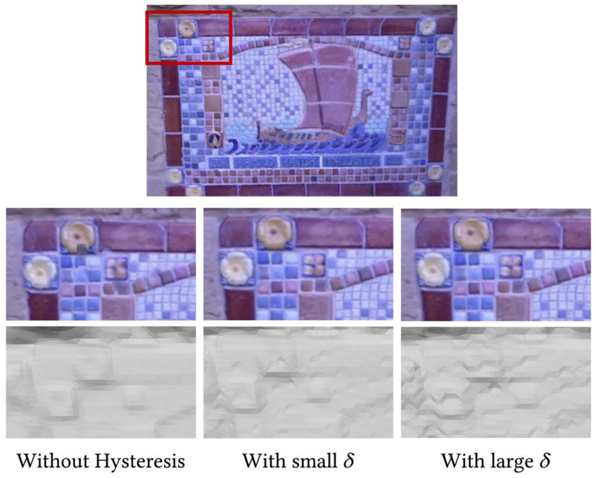

Fig. 12. Effect of hysteresis-based TSDF filter. (Left) Reconstruction without hysteresis. Some texture patches undergo frequent resampling and become incoherent with neighbors. (Middle, right) Reconstructions with hysteresis, where δ is set to 0.1 and 0.2 cm, respectively. Our TSDF filter reduces Marching Cubes' triangle configuration changes and, thereby, provides stable texture patch updates. (Bottom) However, as a side effect, the geometry of the reconstructed model could be less smooth when a big δ is used.

VoxelHashing does not deal with color misalignments at all, the result is less sharp than our reconstructed model (Figure 13).

Effect of different texture representations. We compare our texture representation with TextureFusion [Lee et al. 2020], analyzing its effect on the resulting appearance as the voxel size is increased (Figure 14). To obtain the result of TextureFusion at 4-cm voxels, we set the texture tile size to 16×16. For other voxel sizes, we follow the original paper setting [Lee et al. 2020]. Both approaches of ours and Lee et al. [2020] manage and update a texture map to store detailed color information for single voxels regardless of voxel size but use different texture representations. TextureFusion assigns a single texture tile on a side of a zero-crossing voxel (see Figure 2), then obtains textures of faces of the final mesh model by performing orthographic projection onto the texture tile stored at a voxel side. Therefore, if the reconstructed mesh faces within a voxel are orthogonal to the texture tile on the voxel side, color information

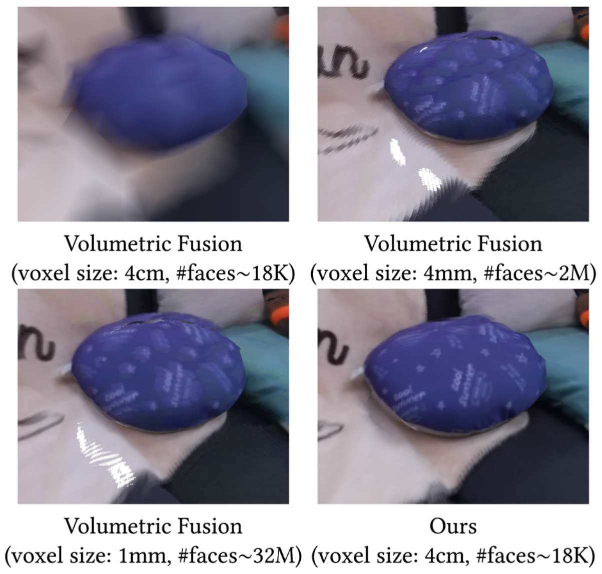
Volumetric Fusion
(voxel size: 4cm, #faces~18K)
Volumetric Fusion
(voxel size: 4mm, #faces~2M)
Volumetric Fusion
(voxel size: 1mm, #faces~32M)
Ours
(voxel size: 4cm, #faces~18K)

Fig. 13. Renderings of the reconstructed models using different voxel sizes for the 'Sofa' scene. Large voxel size severely degrades the result from volumetric color fusion. Using a smaller voxel size at the cost of highly detailed geometry and increased computation time still produces a model that has an unsatisfactory, blurry appearance. In contrast, our method reconstructs a high-quality texture even with 4-cm voxels.

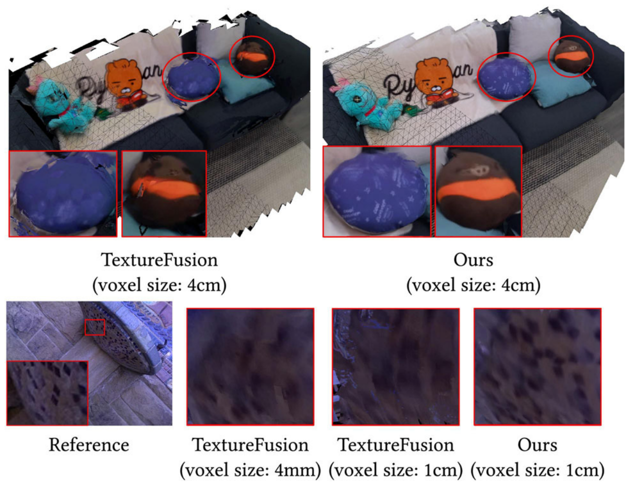

Fig. 14. Effect of different texture representations on the final reconstructed models. We compare our texture representation with the one-texture-tile-per-voxel scheme of TextureFusion [Lee et al. 2020]. (Refer to Figure 2 for visualization of respective representations.) Our texture representation is more flexible in that plentiful color information is retained regardless of voxel size variation. However, the texture expressive power of TextureFusion drastically decreases as the voxel size increases and texture tiles are assigned to improper voxel sides. As shown in the 'Fountain' scene (bottom row), albeit using even 4-mm voxels, TextureFusion lost color details due to its inflexible texture representation.

that can be obtained from the texture tile for the mesh faces is reduced as shown in Figure 14. On the other hand, as our texture representation directly maps texture patches to the corresponding reconstructed mesh faces, it does not suffer from the problem.

7.3 Comparisons

Baselines. We compare our system with a real-time volumetric fusion algorithm [Nießner et al. 2013] and two offline color refinement algorithms [Fu et al. 2018; Zhou and Koltun 2014]. We also compare it with TextureFusion [Lee et al. 2020], which reconstructs geometry and its texture map in real time, similarly to our method. We obtained all results using the original codes provided by the authors.

Although comparison of our real-time approach with offline methods may not be fair, we provide it to show the effectiveness of our approach. The offline methods require the reconstructed model, camera poses, and keyframes. Thus, to test these approaches, we use VoxelHashing [Nießner et al. 2013] to compute camera poses and to obtain the reconstructed model. For these approaches, we also select sharp color images as the keyframes using the heuristics proposed in Zhou and Koltun [2014].

For all results obtained by VoxelHashing [Nießner et al. 2013], the size of a voxel was set to 4 mm, which would be a reasonable setting for room-sized test scenes. We also set the voxel size to 4 mm to obtain reconstruction results of TextureFusion [Lee et al. 2020] on all test scenes except for a few scenes (Scene0000_00, Scene0261_02, Lounge, and Stonewall), for which the voxel size is set to 1 cm due to GPU VRAM limitation. The texture tile size was set to 4×4 for 4-mm voxels and to 8×8 for 1-cm voxels as suggested in Lee et al. [2020]. To reproduce the results of Zhou and Koltun [2014], the reconstructed model by VoxelHashing was directly used. In this approach, the quality of the colored geometry profoundly depends on the vertex density of the model. On the other hand, to reproduce the results of Fu et al. [2018], we decimated the mesh faces to retain 3% of the original. The method takes excessive time to handle a large number of faces. Our system used a voxel size of 1 cm for all results reported here.

Computation time comparison. For timing comparison to real-time methods (VoxelHashing [Nießner et al. 2013] and TextureFusion [Lee et al. 2020]), we measured the average time to process a single frame on some scenes of different datasets (Tables 1 and 2). We also measured the dependence of computation time on voxel size. VoxelHashing is the fastest since it does not have any additional computation burdens to prevent color from being blurry, unlike our TextureMe or TextureFusion [Lee et al. 2020]. Thus, it shows the poor color quality of reconstructions (Figures 15, 16, and 18). While TextureFusion with the 4-mm voxel size needs a much larger computation time (about 20 ms) than ours in the 'Sofa' scene with high-resolution color images, the visual quality of the scene reconstruction is rather lower than ours (Figure 1, supplementary video). The visual quality superiority of our system to TextureFusion is overall maintained in other scenes in our dataset (Figure 17).

Qualitative comparison. As can be observed in Figures 15, 16, and 18, our results show better quality than the real-time volumetric fusion method [Nießner et al. 2013]. Volumetric fusion has a limited capacity since a single color value is stored at each voxel. Moreover, it does not resolve color misalignments, resulting in blurry color information. In contrast, our framework allows each voxel

Table 3. Quantitative Comparison with Previous Methods on Various Datasets (Ours, 3D Scene [Zhou and Koltun 2013, 2014], and ScanNet [Dai et al. 2017a])

	[Fu et al. 2018] Offline (1 hr + α)			[Nießner et al. 2013] Real-time			[Zhou and Koltun 2014] Offline (18 minutes + α)			[Lee et al. 2020] Real-time			TextureMe (ours) Real-time		
Ours	PSNR	SSIM	$\lvert C_b\rvert + \lvert C_r\rvert$	PSNR	SSIM	$\lvert C_b\rvert + \lvert C_r\rvert$	PSNR	SSIM	$\lvert C_b\rvert + \lvert C_r\rvert$	PSNR	SSIM	$\lvert C_b\rvert + \lvert C_r\rvert$	PSNR	SSIM	$\lvert C_b\rvert + \lvert C_r\rvert$
Cuboid	21.334	0.642	1.998	19.806	0.619	2.151	21.759	0.674	1.968	13.760	0.585	2.168	**24.032**	**0.751**	**1.741**
Einstein	19.683	0.472	2.715	21.026	0.471	2.738	21.505	0.491	3.210	17.151	0.476	3.434	**23.384**	**0.565**	**2.245**
LabShelf	18.627	0.695	3.773	19.415	0.729	3.674	19.749	0.742	3.519	12.801	0.573	4.811	**21.212**	**0.770**	**2.987**
Sofa	19.435	0.627	2.376	18.472	0.615	3.405	20.256	0.679	**2.322**	15.853	0.540	3.013	**21.244**	**0.691**	2.383
Bookshelf 1	15.451	0.351	3.820	15.834	0.355	4.079	16.080	0.386	4.003	13.901	0.304	4.542	**18.314**	**0.488**	**2.965**
Bookshelf 2	14.590	0.350	7.118	14.348	0.313	7.953	14.735	0.358	6.997	11.944	0.204	10.306	**16.359**	**0.413**	**5.631**
Puppet	15.083	0.263	3.168	15.185	0.287	3.906	16.076	0.317	3.490	12.892	0.257	4.003	**18.160**	**0.371**	**2.642**
Room 1	22.822	0.754	2.492	21.937	0.781	2.535	23.108	**0.814**	2.285	13.836	0.638	4.988	**23.731**	0.796	**2.173**
Room 2	21.063	0.747	2.905	20.955	0.760	3.312	21.755	**0.793**	**2.751**	12.444	0.643	4.735	**21.918**	0.775	3.043
3D Scene	PSNR	SSIM	$\lvert C_b\rvert + \lvert C_r\rvert$	PSNR	SSIM	$\lvert C_b\rvert + \lvert C_r\rvert$	PSNR	SSIM	$\lvert C_b\rvert + \lvert C_r\rvert$	PSNR	SSIM	$\lvert C_b\rvert + \lvert C_r\rvert$	PSNR	SSIM	$\lvert C_b\rvert + \lvert C_r\rvert$
Fountain	18.060	0.318	3.926	17.939	0.293	4.893	**19.410**	**0.382**	**3.457**	18.186	0.358	3.919	18.428	0.295	4.511
Lounge	12.935	0.354	6.373	13.180	0.402	6.688	**14.834**	**0.504**	**5.059**	12.543	0.458	5.682	12.588	0.423	6.649
Totempole	16.793	0.404	3.136	16.056	0.342	4.176	**18.521**	**0.486**	**2.733**	15.563	0.395	3.618	16.579	0.384	3.661
Stonewall	19.876	0.451	3.134	18.103	0.416	4.261	**21.014**	**0.518**	**2.607**	17.298	0.467	3.235	18.914	0.398	4.118
ScanNet	PSNR	SSIM	$\lvert C_b\rvert + \lvert C_r\rvert$	PSNR	SSIM	$\lvert C_b\rvert + \lvert C_r\rvert$	PSNR	SSIM	$\lvert C_b\rvert + \lvert C_r\rvert$	PSNR	SSIM	$\lvert C_b\rvert + \lvert C_r\rvert$	PSNR	SSIM	$\lvert C_b\rvert + \lvert C_r\rvert$
Scene0000_00	15.399	0.572	5.590	16.769	0.682	5.242	**17.741**	**0.719**	4.138	14.810	0.638	5.347	17.466	0.711	**4.137**
Scene0261_02	16.424	0.633	4.238	20.303	0.733	3.382	22.028	**0.783**	2.604	15.394	0.685	4.161	**22.294**	0.761	**2.584**

Note that our real-time approach is even comparable to the offline approaches. Offline approaches we compare with need additional preprocessing time for 3D model reconstruction, keyframe sampling, and mesh decimation. α denotes the additional computation time for such preprocessing.

to store and update *texture patches* on the global texture atlas, and keeps the color information being reconstructed as sharp as possible by our texture patch optimization. As a result, we obtain much sharper and cleaner textures for the reconstructed model.

Models reconstructed by TextureFusion [Lee et al. 2020] occasionally show appearances that are inferior even to models from VoxelHashing [Nießner et al. 2013] that do not include any color alignment algorithm, when its perspective warping produces erroneous alignments (Figures 17 and 18). In contrast, our texture patch optimization works more stably and robustly on several test scenes. In addition, TextureFusion reconstruction results may suffer from black-colored regions, which are caused by the strong constraint on texture tile update and are frequently observed in the results (Figure 17). Each texel in a texture tile is updated only if its occlusion weight is larger than the prescribed scalar value, so there could exist texels that have not been updated at all and remain in black. On the other hand, our approach does not have such a problem since our method instantly assigns and updates texture patches for reconstructed mesh faces. Overall, our reconstruction results show better visual quality than TextureFusion (Figures 17 and 18).

Our results are on par with the offline texture mapping results of Fu et al. [2018] and Zhou and Koltun [2014]. It is worth noting that our approach does not need to extract keyframes, unlike offline techniques. The computation times of these offline methods depend on the number of vertices and the faces of the model. Specifically, Zhou and Koltun [2014] spend about 18 minutes for a model with about 2.5 M vertices, and Fu et al. [2018] spend about one hour for a model with 0.1 M faces. Fu et al. [2018] select a single frame for texturing each face so that they could avoid the blurry artifact. However, their method still leaves seams on vastlymisaligned regions. Moreover, as the offline approach suffers from shading variations among keyframes, color correction is necessary at post-processing. On the other hand, our method reduces the artifact by blending all incoming frames and needs no post-processing.

Quantitative comparison. Evaluation of the textured geometry is not straightforward to devise, since it is hard to obtain the precise ground truth geometry with textures. In this article, we adopt the evaluation scheme proposed by Waechter et al. [2017], which is based on novel view predictions. Under the assumption that camera poses (ground truths or estimates) of real images are known, the approach obtains an image of a novel view using the standard rendering pipeline and compares the rendered image with the real captured image. It evaluates the rendered image in terms of completeness and accuracy. Completeness is the measure that depends on the geometry. Since all of the approaches, including ours, are based on the same geometry reconstruction pipeline of KinectFusion [Newcombe et al. 2011], we aim at comparing the accuracy.

The accuracy relates to the consistency between the reconstruction and the input image. We render the textured reconstructed model using the camera poses without any shading. The rendered images are compared with the input color images using three metrics — PSNR, SSIM, and $|C_b| + |C_r|$. $|C_b| + |C_r|$ —proposed by Waechter et al. [2017] that measure the sum of absolute differences of the two chrominance channels in the YC_bC_r color space. We did not take holes into account when calculating the metrics.

Two offline algorithms [Fu et al. 2018; Zhou and Koltun 2014] use 3D models reconstructed by VoxelHashing [Nießner et al. 2013] as the input. Thus, we used the camera poses estimated by VoxelHashing during reconstruction to produce rendered images. For our method and TextureFusion [Lee et al. 2020], we used the camera poses estimated during the reconstruction process.

We compute the three measures for every frame in a test scene and calculate the average values. The comparison with the four representative approaches [Fu et al. 2018; Lee et al. 2020; Nießner et al. 2013; Zhou and Koltun 2014] are shown in Table 3.

Our TextureMe shows better quantitative results than the volumetric fusion [Nießner et al. 2013] for 20 test scenes except for the lounge scene. With our dataset (denoted as ours), TextureMe produces better images than the other two offline methods and

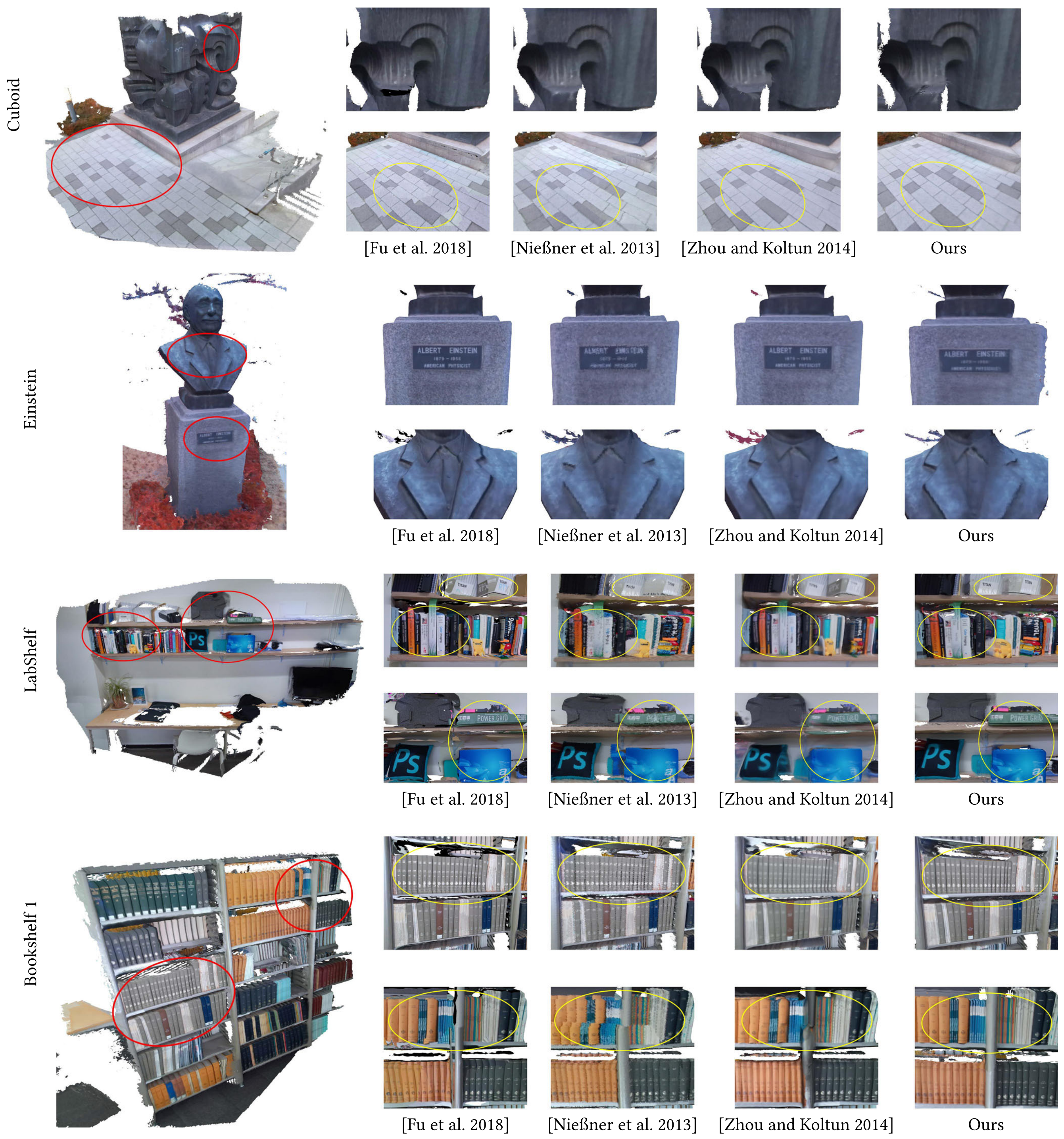

Fig. 15. Visual comparison with previous methods on various scenes of our dataset. We compare our approach against one real-time volumetric fusion system [Nießner et al. 2013] and two offline algorithms [Fu et al. 2018; Zhou and Koltun 2014]. The leftmost image of each row shows the reconstructed model by our method, and we list closeup views for models reconstructed by respective methods. Although our system operates in real time, it obtains high-quality textured meshes comparable to two offline algorithms, or preferably better in some scenes.

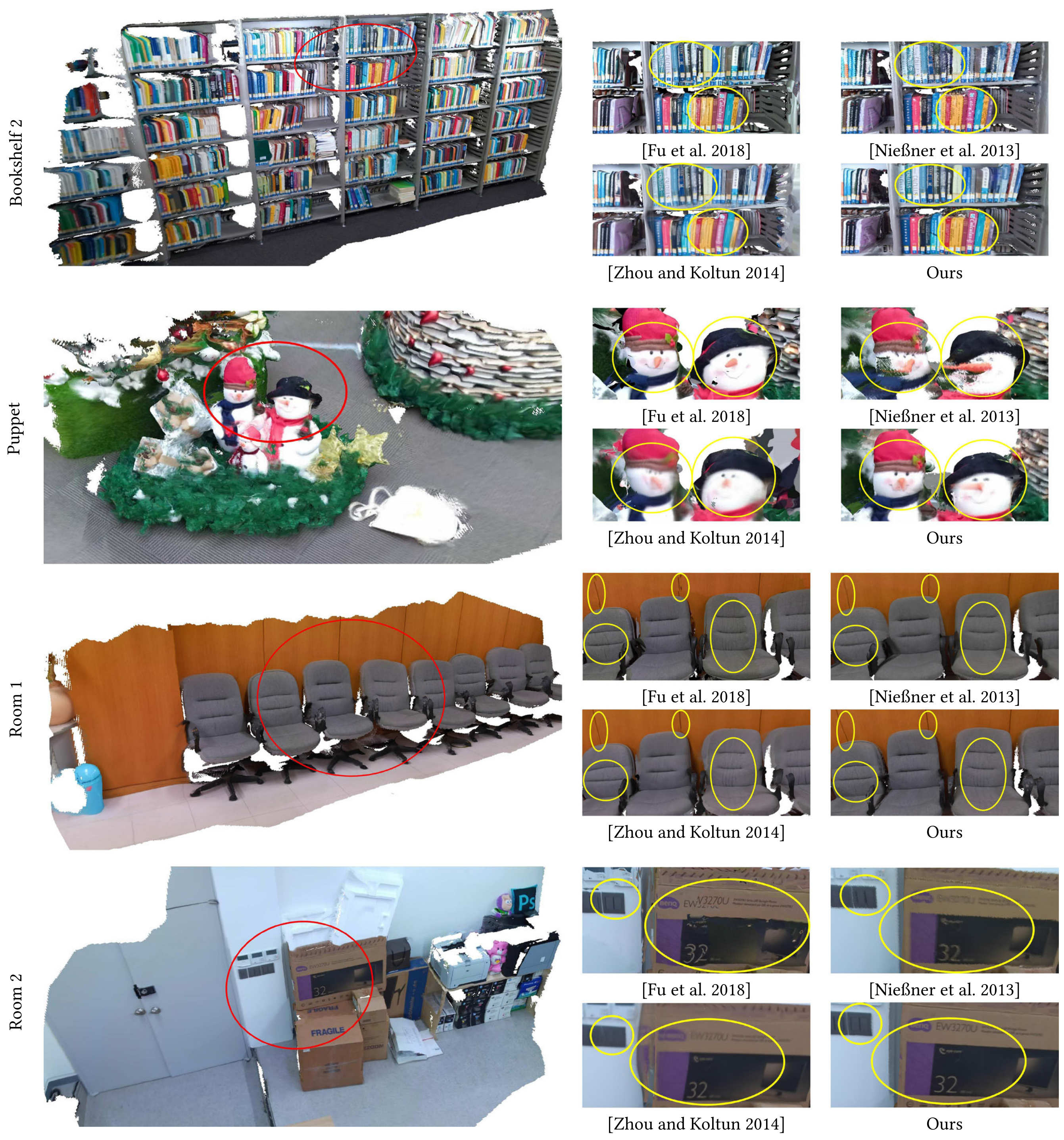

Fig. 16. Visual comparison with previous methods of other scenes of our dataset. We compare our approach against one real-time volumetric fusion system [Nießner et al. 2013] and two offline algorithms [Fu et al. 2018; Zhou and Koltun 2014]. The leftmost image of each row shows the reconstructed model by our method, and we list closeup views for models reconstructed by respective methods. Although our system operates in real time, it obtains high-quality textured meshes comparable to two offline algorithms, or preferably better in some scenes.

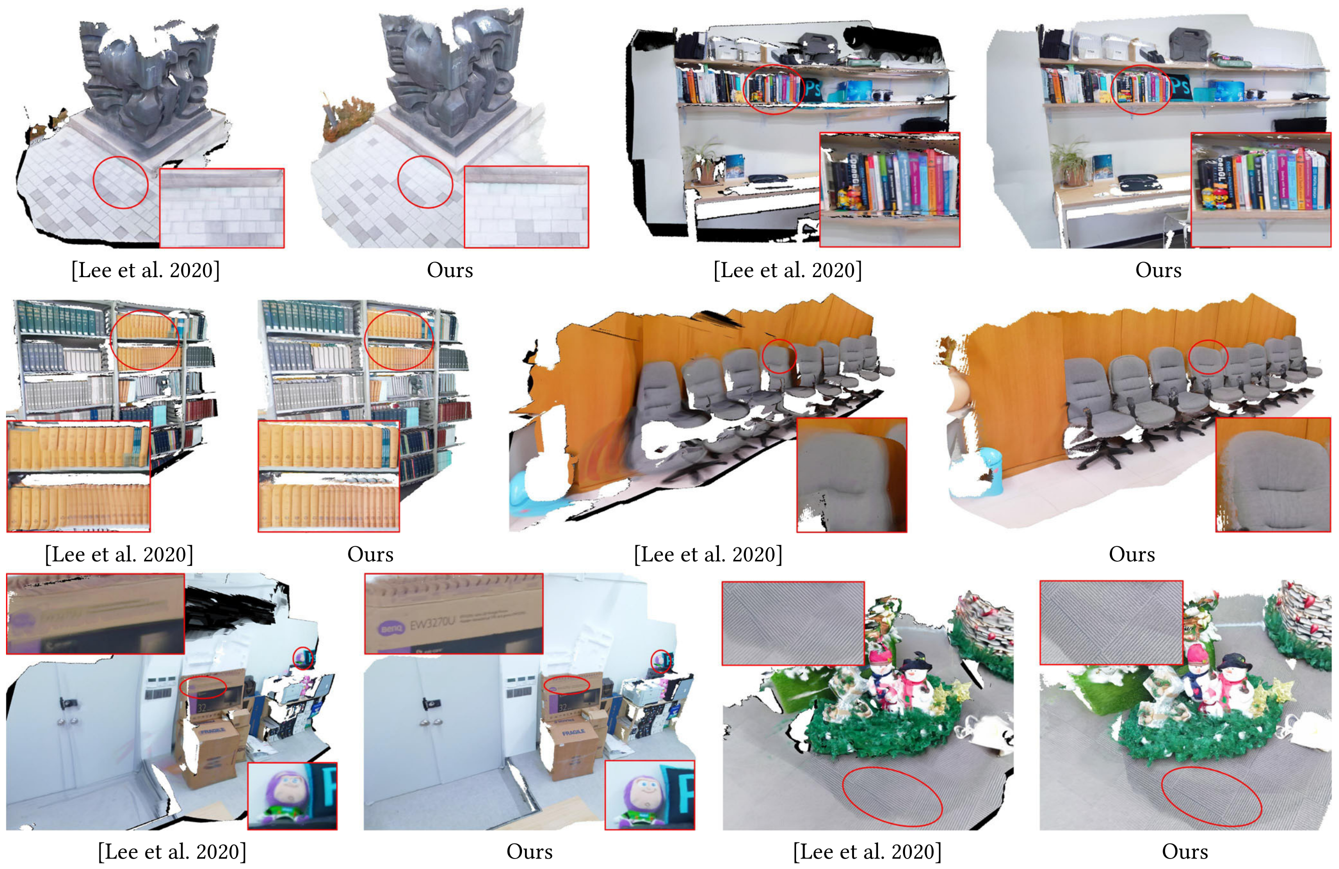

Fig. 17. Visual comparisons with TextureFusion [Lee et al. 2020] of scenes of our dataset. We show the reconstructed models of ours and TextureFusion at both holistic and closeup views. Overall, the models reconstructed by our method have better visual quality than ones constructed by TextureFusion.

TextureFusion. Our approach would be a good fit for HD high-quality color images, since our texture patch representation can easily capture details in the color images. The plentiful color information also has a positive influence on our texture patch optimization since the optimization is based on the photometric error.

The color images of 3D Scene and ScanNet datasets are low resolution and blurry. Even with these datasets, our approach produces comparable results to offline methods. Note that the offline methods spend a substantial amount of time obtaining the optimized colors of the reconstructed model. Additional time is necessary for preprocessing, including 3D mesh model reconstruction, keyframe sampling, and mesh decimation.

Discussion. Our global texture atlas is highly fragmented, thereby making it inconvenient for a user to edit the texture atlas directly. However, we think indirect texture map editing would be possible by manipulating the texture on the 3D model surface and projecting the manipulation result onto the texture atlas.

If texture patches in the global texture atlas are blurred and distorted due to accumulated color misalignments, texture patch optimization may not work appropriately because it cannot find proper gradients to decrease the objective function of the optimization. In that case, texture patches would be fused without effective texture patch optimization. On the other hand, since weights of texture patches in the global texture atlas cannot be larger than the predefined maximum, the accumulated texture patches are mainly affected by recently observed color images. As a result, texture patches in the global atlas are not distorted more than a certain level, as observed in various texture reconstruction results (Figures 15, 16, and 18).

We consider geometry and its texture reconstruction for static scenes in this paper. We believe that with some adaptation, the texture module of our system can be applied to volumetric fusion methods for dynamic objects, such as DynamicFusion [Newcombe et al. 2015]. To be specific, such a method generally retains a volume space in the canonical frame to update the geometry and color of the target model. For geometry and color update, the method deforms the model being reconstructed in the canonical frame to fit the current frame, and then update TSDF and color values stored at voxels using the current depth and color images. We could handle dynamic objects by managing and updating voxels and their texture patches for the volume maintained in the canonical frame.

Limitations. Even though our approach can create a high-quality texture map in real time, its result can be degraded by non-uniform lighting conditions (Figure 19). In this case, during the capture, the brightness of a part of the scene changes abruptly with camera motion due to strong specular lighting. Then, textures

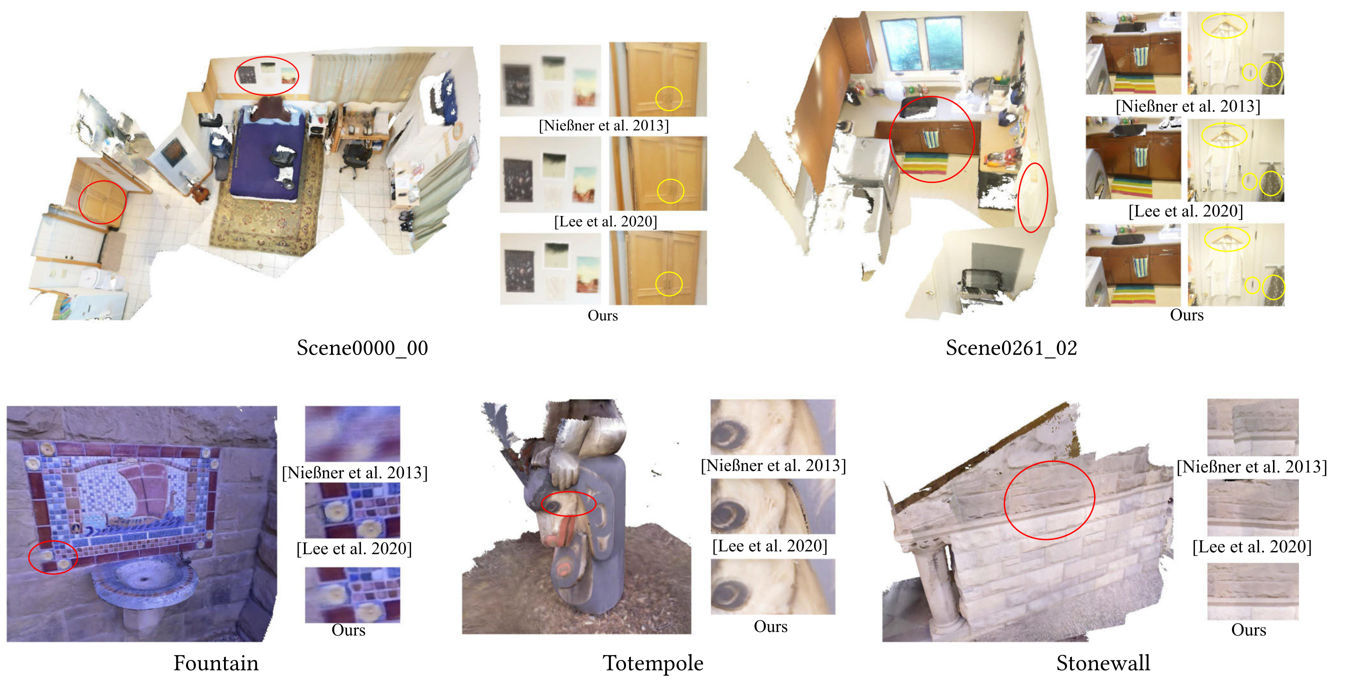

Fig. 18. Additional visual comparison with VoxelHashing [Nießner et al. 2013] and TextureFusion [Lee et al. 2020] of scenes of public datasets (3D Scene [Zhou and Koltun 2013, 2014] and ScanNet [Dai et al. 2017a] datasets). The left figure of each scene shows the reconstructed model by our method. As seen in closeup images, our reconstructed models show sharper appearances than results by VoxelHashing on all scenes. Compared with TextureFusion, our results show comparable or sharper visual quality except in the 'Fountain' scene.

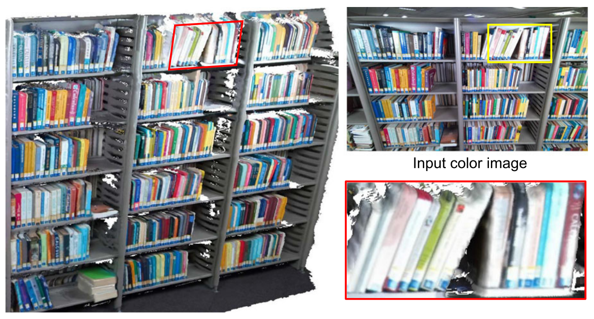

Fig. 19. Failure case I. Our approach suffers from non-uniform lighting conditions of input color images. In 'Bookshelf2' scene, the brightness of a part of the scene severely changes during the capture due to strong specular lighting. As a result, the fused texture of that part shows saturated brighter colors compared with other parts.

with different brightness could be merged on that part, resulting in a much brighter texture than other parts. We think solving for coherent scene radiance may be possible to resolve the artifact.

Our weight computation for texture patch update includes a blurriness score computed using a perceptual blur metric [Crete et al. 2007]. However, the metric is not always accurate and may regard a blurry frame as a sharp one, resulting in a less sharp texture map. Our system may obtain performance improvement with a more reliable method for blurriness estimation.

Since our texture patch optimization is based on the photometric error and assumes a moderately accurate camera pose and a

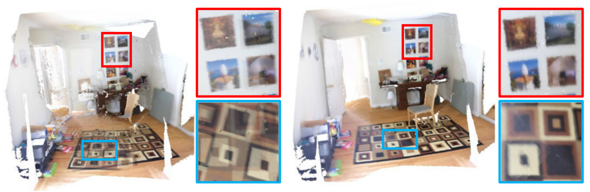

Fig. 20. Failure case II. The left image shows a tracking failure case on a challenging indoor scene in ScanNet [Dai et al. 2017a]. The underlying reconstruction pipeline can be improved with a globally consistent pose estimation pipeline, such as BundleFusion [Dai et al. 2017b]. The right image shows the reconstruction result using the camera poses obtained by BundleFusion. As shown in the red and blue boxes, the textures are improved with the camera poses.

proper voxel size, it will not work (1) when an input color image changes abruptly locally or globally due to specular lighting or auto-white balancing, resulting in large discrepancy from the color of the current reconstructed model, (2) if camera pose tracking fails or a large camera pose error occurs (Figure 20), and (3) when too-large voxels are used to make the geometry oversimplified, incurring mismatches between input color images and rendered images (Figure 21).

If the camera pose error is too large to be recovered by our texture patch optimization, the remedy would be to adopt high-level image features and to detect loop closure to relieve pose error accumulation. Figure 20 shows an example of this idea. We used BundleFusion [Dai et al. 2017b] to obtain more reliable camera poses while keeping our texture mapping pipeline

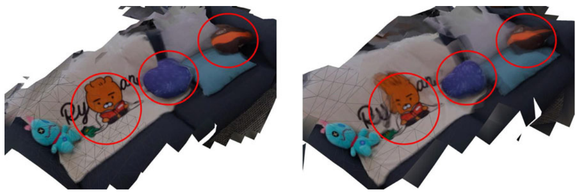

Fig. 21. Failure case III. In our TextureMe reconstructions with large voxels (8 cm and 16 cm), as the voxel size increases, the reconstructed geometry becomes simplified, consequently, intensifying the discrepancy between the input color image and the rendered one. Our texture patch optimization may not be able to resolve such a large discrepancy.

unchanged. The reconstruction subsequently improved, and the texture mapping result was also improved with the camera poses. Note that the color images in this dataset exhibit severe motions blur due to rapid camera motions.

8 CONCLUSIONS

In this article, we proposed a novel approach to reconstruct a high-quality texture map as well as 3D geometry using RGB-D images in real time. The key to our approach is to create and manage a dynamic global texture atlas that can be naturally adapted to the complex geometry being evolved through reconstruction.

With our texture patch update scheme, we produce a high-quality texture-mapped mesh. Our texture patch optimization module effectively and efficiently handles large and small color misalignments that frequently occur in typical camera tracking. Our approach does not require any post-processing, such as texture map optimization or mesh parameterization.

To verify the performance, we compared our system with real-time and offline methods, both qualitatively and quantitatively, on various indoor and outdoor scenes. Our results are superior to voxel-based color blending schemes and comparable to or better in some scenes than state-of-the-art color optimization approaches that run offline.

Future work includes proposing scalable texture maps for large scenes. Currently, our approach assumes that the scene to be recovered is bounded in a certain range; thus, a fixed size of the texture atlas is used. As volumetric fusion systems have been extended to cover larger scenes, our texture atlas could be extended depending on the size of the scene as well. Any large-scale volumetric fusion system, such as BundleFusion [Dai et al. 2017b], can be a viable option to integrate our texture module because our framework depends on a volumetric fusion approach for geometry reconstruction. Another research direction is to design texture optimization robust to local and global color changes.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their constructive comments.

REFERENCES

Fausto Bernardini, Ioana M. Martin, and Holly Rushmeier. 2001. High-quality texture reconstruction from multiple scans. *IEEE Transactions on Visualization and Computer Graphics* 7, 4 (2001), 318–332.

Sai Bi, Nima Khademi Kalantari, and Ravi Ramamoorthi. 2017. Patch-based optimization for image-based texture mapping. *ACM Transactions on Graphics* 36, 4 (2017), 106–101.

Chris Buehler, Michael Bosse, Leonard McMillan, Steven Gortler, and Michael Cohen. 2001. Unstructured lumigraph rendering. In *Proceedings of SIGGRAPH Conference*. 425–432.

Nathan A. Carr and John C. Hart. 2002. Meshed atlases for real-time procedural solid texturing. *ACM Transactions on Graphics* 21, 2 (April 2002), 106–131.

Anpei Chen, Minye Wu, Yingliang Zhang, Nianyi Li, Jie Lu, Shenghua Gao, and Jingyi Yu. 2018. Deep surface light fields. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 1, 1 (2018), 1–17.

Wei-Chao Chen, Jean-Yves Bouguet, Michael H. Chu, and Radek Grzeszczuk. 2002. Light field mapping: Efficient representation and hardware rendering of surface light fields. *ACM Transactions on Graphics* 21, 3 (2002), 447–456.

Sungjoon Choi, Qian-Yi Zhou, and Vladlen Koltun. 2015. Robust reconstruction of indoor scenes. In *Proceedings of CVPR*. 5556–5565.

Frederique Crete, Thierry Dolmiere, Patricia Ladret, and Marina Nicolas. 2007. The blur effect: Perception and estimation with a new no-reference perceptual blur metric. In *Human Vision and Electronic Imaging XII*, Vol. 6492. International Society for Optics and Photonics, 64920I.

Brian Curless and Marc Levoy. 1996. A volumetric method for building complex models from range images. In *Proceedings of SIGGRAPH Conference*. 303–312.

Angela Dai, Angel X. Chang, Manolis Savva, Maciej Halber, Thomas Funkhouser, and Matthias Nießner. 2017a. ScanNet: Richly-annotated 3D reconstructions of indoor scenes. In *Proceedings of CVPR*. 5828–5839.

Angela Dai, Matthias Nießner, Michael Zollhöfer, Shahram Izadi, and Christian Theobalt. 2017b. Bundlefusion: Real-time globally consistent 3D reconstruction using on-the-fly surface reintegration. *ACM Transactions on Graphics* 36, 3 (2017), 24.

Paul Debevec, Yizhou Yu, and George Borshukov. 1998. Efficient view-dependent image-based rendering with projective texture-mapping. In *Eurographics Workshop on Rendering Techniques*. 105–116.

Yanping Fu, Qingan Yan, Long Yang, Jie Liao, and Chunxia Xiao. 2018. Texture mapping for 3D reconstruction with RGB-D sensor. In *Proceedings of CVPR*. 4645–4653.

Y. Furukawa, B. Curless, S. M. Seitz, and R. Szeliski. 2010. Towards Internet-scale multi-view stereo. In *Proceedings of CVPR*. 1434–1441.

Y. Furukawa and J. Ponce. 2010. Accurate, dense, and robust multiview stereopsis. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 32, 8 (Aug 2010), 1362–1376.

Ran Gal, Yonatan Wexler, Eyal Ofek, Hugues Hoppe, and Daniel Cohen-Or. 2010. Seamless montage for texturing models. In *Computer Graphics Forum*, Vol. 29. 479–486.

Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. 1996. The lumigraph. In *Proceedings of SIGGRAPH Conference*. 43–54.

Jingwei Huang, Angela Dai, Leonidas J. Guibas, and Matthias Nießner. 2017. 3Dlite: Towards commodity 3D scanning for content creation. *ACM Transactions on Graphics* 36, 6 (2017), 203–1.

Jingwei Huang, Justus Thies, Angela Dai, Abhijit Kundu, Chiyu Jiang, Leonidas J. Guibas, Matthias Nießner, Thomas Funkhouser, et al. 2020. Adversarial texture optimization from RGB-D scans. In *Proceedings of CVPR*. 1559–1568.

Junho Jeon, Yeongyu Jung, Haejoon Kim, and Seungyong Lee. 2016. Texture map generation for 3D reconstructed scenes. *The Visual Computer* 32, 6–8 (2016), 955–965.

Michael Kazhdan and Hugues Hoppe. 2013. Screened Poisson surface reconstruction. *ACM Transactions on Graphics* 32, 3, Article 29 (July 2013), 13 pages.

Joo Ho Lee, Hyunho Ha, Yue Dong, Xin Tong, and Min H. Kim. 2020. TextureFusion: High-quality texture acquisition for real-time RGB-D scanning. In *Proceedings of CVPR*. 1269–1277.

Victor Lempitsky and Denis Ivanov. 2007. Seamless mosaicing of image-based texture maps. In *Proceedings of CVPR*. 1–6.

Marc Levoy and Pat Hanrahan. 1996. Light field rendering. In *Proceedings of SIGGRAPH Conference*. 31–42.

Wei Li, Huajun Gong, and Ruigang Yang. 2018. Fast texture mapping adjustment via local/global optimization. *IEEE Transactions on Visualization and Computer Graphics* 25, 6 (2018), 2296–2303.

Lingjie Liu, Jiatao Gu, Kyaw Zaw Lin, Tat-Seng Chua, and Christian Theobalt. 2020. Neural sparse voxel fields. In *Proceedings of NeurIPS*.

William E. Lorensen and Harvey E. Cline. 1987. Marching cubes: A high resolution 3D surface construction algorithm. In *Proceedings of SIGGRAPH Conference*. 163–169.

Kok-Lim Low. 2004. Linear least-squares optimization for point-to-plane ICP surface registration. *Technical Report TR04-004, Chapel Hill, University of North Carolina*, 1–3.

Robert Maier, Kihwan Kim, Daniel Cremers, Jan Kautz, and Matthias Nießner. 2017. Intrinsic3D: High-quality 3D reconstruction by joint appearance and geometry optimization with spatially-varying lighting. In *Proceedings of ICCV*. 3114–3122.

Microsoft. 2020. Azure Kinect DK. Retrieved from https://azure.microsoft.com/en-us/services/kinect-dk/ Online; accessed 19 Jan 2020.

Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. 2020. NeRF: Representing scenes as neural radiance fields for view synthesis. In *Proceedings of ECCV*. 405–421.

Richard A. Newcombe, Dieter Fox, and Steven M. Seitz. 2015. DynamicFusion: Reconstruction and tracking of non-rigid scenes in real-time. In *Proceedings of CVPR*. 343–352.

Richard A. Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J. Davison, Pushmeet Kohi, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. 2011. KinectFusion: Real-time dense surface mapping and tracking. In *Proceedings of ISMAR*. 127–136.

Matthias Nießner, Michael Zollhöfer, Shahram Izadi, and Marc Stamminger. 2013. Real-time 3D reconstruction at scale using voxel hashing. *ACM Transactions on Graphics* 32, 6 (2013), 169.

J. Park, Q. Zhou, and V. Koltun. 2017. Colored point cloud registration revisited. In *Proceedings of ICCV*. 143–152.

Patrick Pérez, Michel Gangnet, and Andrew Blake. 2003. Poisson image editing. In *Proceedings of SIGGRAPH Conference*. 313–318.

Justus Thies, Michael Zollhöfer, and Matthias Nießner. 2019. Deferred neural rendering: Image synthesis using neural textures. *ACM Transactions on Graphics* 38, 4 (2019), 1–12.

Greg Turk and Marc Levoy. 1994. Zippered polygon meshes from range images. In *Proceedings of SIGGRAPH Conference*. 311–318.

Luiz Velho and Jonas Sossai Jr. 2007. Projective texture atlas construction for 3D photography. *The Visual Computer* 23, 9 (2007), 621–629.

Michael Waechter, Mate Beljan, Simon Fuhrmann, Nils Moehrle, Johannes Kopf, and Michael Goesele. 2017. Virtual rephotography: Novel view prediction error for 3D reconstruction. *ACM Transactions on Graphics* 36, 1, Article 8 (Jan. 2017), 11 pages.

Michael Waechter, Nils Moehrle, and Michael Goesele. 2014. Let there be color! Large-scale texturing of 3D reconstructions. In *Proceedings of ECCV*. 836–850.

Thomas Whelan, Stefan Leutenegger, R. Salas-Moreno, Ben Glocker, and Andrew Davison. 2015. ElasticFusion: Dense SLAM without a pose graph. Robotics: Science and Systems. In *Proceedings of RSS*.

Thomas Whelan, John McDonald, Michael Kaess, Maurice Fallon, Hordur Johannsson, and John J. Leonard. 2012. Kintinuous: Spatially extended KinectFusion. In *Proceedings of RSS'12 Workshop on RGB-D: Advanced Reasoning with Depth Cameras*.

Zexiang Xu, Sai Bi, Kalyan Sunkavalli, Sunil Hadap, Hao Su, and Ravi Ramamoorthi. 2019. Deep view synthesis from sparse photometric images. *ACM Transactions on Graphics* 38, 4 (2019), 1–13.

K. Zhou, M. Gong, X. Huang, and B. Guo. 2011. Data-parallel octrees for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics* 17, 5 (May 2011), 669–681.

Qian-Yi Zhou and Vladlen Koltun. 2013. Dense scene reconstruction with points of interest. *ACM Transactions on Graphics* 32, 4 (2013), 1–8.

Qian-Yi Zhou and Vladlen Koltun. 2014. Color map optimization for 3D reconstruction with consumer depth cameras. *ACM Transactions on Graphics* 33, 4 (2014), 155.

Qian-Yi Zhou, Stephen Miller, and Vladlen Koltun. 2013. Elastic fragments for dense scene reconstruction. In *Proceedings of ICCV*. 473–480.

Received November 2020; revised November 2021; accepted December 2021