Patch Volumes: Segmentation-based Consistent Mapping with RGB-D Cameras

Peter Henry and Dieter Fox
University of Washington
Computer Science & Engineering
Seattle, Washington
peter@cs.washington.edu
fox@cs.washington.edu

Achintya Bhowmik and Rajiv Mongia
Intel Corporation
Santa Clara, California
achintya.k.bhowmik@intel.com
rajiv.k.mongia@intel.com

Abstract

Recent advances have allowed for the creation of dense, accurate 3D maps of indoor environments using RGB-D cameras. Some techniques are able to create large-scale maps, while others focus on accurate details using GPU-accelerated volumetric representations. In this work we describe patch volumes*, a novel multiple-volume representation which enables the creation of globally consistent maps of indoor environments beyond the capabilities of previous high-accuracy volumetric representations.*

1. Introduction

The availability of RGB-D cameras such as the Microsoft Kinect has opened up research on how to utilize these cameras for modeling environments and objects. The capability to create models with these inexpensive sensors has applications in robotics, augmented reality, localization, and 3D content creation. Several research group have developed techniques for generating consistent maps of large scale environments. These approaches typically extend visual odometry to incorporate depth information, and use feature matching techniques to detect loop closures followed by global alignment via graph-based optimization [11, 8, 12, 7]. In a separate line of work, Newcombe and colleagues introduced KinectFusion and showed how extremely accurate volumetric representations can be generated in real time taking advantage of GPU based optimizations [15]. Since this approach relies on keeping the full model in GPU memory, a straightforward implementation does not scale to larger environments. To overcome this limitation, Whelan and colleagues introduced Kintinuous, which keeps only a small, active part of the model in GPU memory [20]. However, the rigid, volumetric models underlying both KinectFusion and Kintinuous do not allow for global re-optimization after loop closure detection. They are thus not able to align data collected in very large environments.

In this paper we present a framework which we call *patch volumes*, which has the ability to create large globally consistent scene models with RGB-D cameras. Our approach combines the accuracy and efficiency of volumetric representations with the global consistency achieved with feature based approaches and graph optimization.

2. Related Work

The first paper describing a system for SLAM and dense reconstruction with RGB-D cameras is from Henry *et al*. [10], with an expanded treatment in [11]. This system performs frame-to-frame alignment with a combined optimization over sparse visual features and iterative closest point (ICP) [2] with a point-to-plane error metric [4]. Loop closures are suggested through place recognition, established with the same frame-to-frame alignment method, and optimized with pose graph optimization or bundle adjustment. This system was also extended to an interactive setting with real-time feedback and error recovery [6]. A similar system was described by Engelhard *et al*. [8], and their code was made available[1]. This system was evaluated on a publicly available dataset in [7].

Newcombe *et al*. provided an inspired approach to smaller scale high fidelity geometric reconstruction with KinectFusion [15]. This system utilizes a real-time GPU implementation of the range image fusion method of Curless and Levoy [5] to create a volumetric truncated signed distance function (TSDF) representation of the environment. Subsequent frames are aligned against this model with projective association point-to-plane ICP by performing parallel GPU ray-casting into the volume to generate a surface prediction. By performing frame-to-model alignment, drift is significantly reduced, eliminating the need for explicit loop closure detection or global optimization for the sequences on which it was demonstrated. The resulting models exhibit substantially higher depth accuracy than is present in an individual frame due to the effective noise reduction from TSDF fusion. KinectFusion inspired sev-

[1] http://www.ros.org/wiki/rgbdslam

eral widely used implementations such as Kinfu[2] and ReconstructMe[3].

One key limitation of KinectFusion is the restriction on model size imposed by the need to store a dense volumetric representation of the entire model in GPU memory. One method to alleviate this issue was originally described as Kintinuous [20]. In this work, the open source KinFu implementation was modified to allow a moving TSDF fusion volume. As the camera moves through the environment, slices of the volume which fall out of the view frustum are converted to a mesh. This GPU memory is efficiently reclaimed by treating the volume as a 3D circular buffer. While this allows the system to map effectively unbounded environments, no provision is made for reincorporating the mesh data into tracking or fusion when the camera returns to previously visited locations.

Other work avoids ICP-style shape alignment and uses the depth sensor of the RGB-D camera to enable 3D warping for dense color alignment [1, 18]. These methods do not volumetrically reconstruct the scene, but focus on the problem of frame-to-frame visual odometry, minimizing the error in projection of color from the previous frame onto the current color frame.

Recently, contemporaneous work by Whalen *et al.* [19] combines the moving volume and ICP alignment of Kintinuous with dense color frame-to-model registration on the GPU, as well as feature-based frame-to-frame alignment on the CPU. Their system runs in real time and produces high quality reconstructions. Notably, the camera trajectories for which they demonstrate results do not return to previously mapped areas, for as with their previous Kintinuous [20] work, they provide no method for loop closure or global consistency.

No previous system combines the real-time high accuracy models from GPU-based volumetric fusion with the global consistency of RGB-D slam systems. It is this combination of traits we wish to achieve in this work.

3. System Description

Our complete system applies volumetric fusion, ray-casting for surface prediction, dense frame-to-model alignment, and pose-graph inspired optimization for global consistency. Our system is implemented in C++. The normal computation, patch volume fusion, ray-casting, and alignment optimization are all implemented in OpenCL. We cover each of these components in detail in the following sections.

3.1. Patch Volume Representation

A patch volume (PV) is a dense volumetric representation of a region of space. Taking inspiration from KinectFusion [15] which uses the technique of [5], we represent the geometry as a TSDF, which is a dense voxel grid $\mathbf{F}$ where positive voxel values lie in front of the surface, and negative values lie behind the surface. The implicit surface is the zero crossing of the TSDF. We maintain a corresponding grid $\mathbf{W_F}$ of weights which allow weighted average fusion of new depth frames. To store color appearance information, we also store a grid of RGB values $\mathbf{C}$ and corresponding weight grid $\mathbf{W_C}$. A patch volume also has a pose $\mathbf{T}_{PV} \in \mathbb{SE}_3$ relative to the scene frame, which we define as the first camera pose. We also maintain a changing estimate $\mathbf{T}_G$ of the scene relative to the current camera. Thus, to transform PV-frame points to camera-frame points, we apply $\mathbf{T}_G \circ \mathbf{T}_{PV}$.

An entire scene is represented as potentially many patch volumes of arbitrary size and resolution. The original KinectFusion representation can be thought of as a single patch volume without color (only $\mathbf{F}$ and $\mathbf{W_F}$). As we wish to represent arbitrarily large scenes, patch volumes may be dynamically moved into and out of GPU memory based on availability. When a PV is moved out of GPU memory, we compress it using run length encoding to conserve system memory.

A prediction of scene geometry and color from any pose may be obtained through parallel ray-casting. We build up a depth image D_r and color image C_r for a virtual camera by rendering each patch volume sequentially, only overwriting depth and color for a pixel if the newer pixel is closer to the camera. We also produce a normal map N_r and a PV assignment map S_r. As the number of PVs falling into the viewing frustum may require more than the amount of GPU memory available, the system can bring PVs on and off the GPU individually as necessary to render all visible PVs while never exceeding GPU memory.

The method for parallel ray casting is substantially the same as that described in KinectFusion [15]. We make sure to only process the portion of a ray which intersects a given patch volume, which provides a notable speedup for volumes which occupy only a small fraction of the camera frustum. When a zero-crossing is located, the normal is computed as a finite difference approximation of the gradient of $\mathbf{F}$ and the color is trilinearly interpolated in $\mathbf{C}$. At any point the user may pause modeling and generate a colored mesh for each patch volume using the marching cubes algorithm [13]. Each mesh vertex is colored according to trilinear interpolation in $\mathbf{C}$.

3.2. Frame Normals

For a new frame F consisting of depths D_f and colors C_f, we also require an estimate of frame normals N_f for both projective association (section 3.3) and segmentation for PV fusion (section 3.4). To take into account the quadratically increasing noise in D_f values with increasing

[2] http://pointclouds.org/documentation/tutorials/using_kinfu_large_scale.php

[3] http://reconstructme.net/

depth, we use the empirically determined simple axial noise model of [16] (which implies $z_{min} = 0.4$):

$$\sigma_z(z) = 0.0012 + 0.0019(z - 0.4)^2 \quad (1)$$

We consider a neighboring point to be from the *same surface* as the point under consideration if the depth difference is less than $3\sigma_z$.

Similar to KinectFusion [15], a pixel (u, v) has a valid normal if all four neighboring pixels are from the same surface, providing two pixel-space axis-aligned 3D gradient vectors. A raw estimate of the normal $N_f^0(u, v)$ is computed using the cross product of these vectors. This raw estimate is refined with $k = 2$ iterations of a depth aware box filter, in which all valid normals from the same surface in a 3×3 neighborhood around $N_f^i(u, v)$ are averaged to produce the next value $N_f^{i+1}(u, v)$. The final value for each valid normal is $N_f(u, v) = N_f^k(u, v)$. All of these operations are implemented on the GPU.

3.3. Model To Frame Alignment

Given a patch volume scene and a new camera frame, we wish to find the pose of the scene relative to camera. Assuming a small amount of motion relative to the last frame, we can use a local alignment method with projective data association.

A full six degree of freedom 3D pose is defined by

$$\begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^\intercal & 1 \end{bmatrix} \in \mathbb{SE}_3 \quad (2)$$

with rotation matrix $\mathbf{R} \in \mathbb{SO}_3$ and translation vector $\mathbf{t} \in \mathbb{R}^3$. We parameterize the six degrees of freedom as $\mathbf{x} \in \mathbb{R}^6$, with $\mathbf{t} = (x_1, x_2, x_3)^\intercal$ representing translation and $\mathbf{q} = (x_4, x_5, x_6)^\intercal$ representing a unit quaternion as $((1 - ||\mathbf{q}||), x_4, x_5, x_6) \in \mathbb{R}^4$, which can be converted to rotation matrix $\mathbf{R}$ in the usual way.

We render the scene from the previous camera pose as described in section 3.1. This produces a depth map D_r, normal map N_r, and color map C_r in the resolution of the camera. For each valid depth $D_r(u, v) = z$, and given camera focal lengths (f_x, f_y) and center (c_u, c_v) the 3D point is computed as

$$\mathbf{p} = \left(\frac{(u - c_u)z}{f_x}, \frac{(v - c_v)z}{f_y}, z \right)^\intercal \quad (3)$$

Naturally, this can be inverted to project a point $\mathbf{p} = (x, y, z)^\intercal$ to pixel space $(u_f, v_f)^\intercal \in \mathbb{R}^2$ as

$$u_f = x\frac{f_x}{z} + c_u \quad v_f = y\frac{f_y}{z} + c_v \quad (4)$$

The rendered normal map is already formed as unit length vectors $N_r(u, v) \in \mathbb{R}^3$.

We wish to take advantage of all depth and color data to make the alignment. We will compute a combined shape and color error term for each valid correspondence between rendered model point and frame point.

The desired 6DOF pose correction is initialized as $\mathbf{x} = \mathbf{0} \in \mathbb{R}^6$. To compute the error for parameters $\mathbf{x}$, we convert them to a 6DOF pose $\hat{\mathbf{T}}$ as in equation 2. An error term is obtained for each model rendered pixel (u, v) with valid rendered depth $d_r = D_r(u, v)$. This d_r is transformed into world coordinates with equation 3 and transformed by $\hat{\mathbf{T}}$ to obtain $\mathbf{p}_r$, which is then projected into the frame with equation 4 yielding frame pixel coordinates $(u_f, v_f)^\intercal \in \mathbb{R}^2$. We also transform the corresponding normal $N_r(u, v)$ with the rotation part of $\hat{\mathbf{T}}$ to obtain $\mathbf{n}_r$. We look up the nearest corresponding frame depth $d_f = D_f(\texttt{round}(u_f), \texttt{round}(v_f))$ which also gives us a frame point $\mathbf{p}_f$ via equation 3. We also look up the normal $\mathbf{n}_f = N_f(\texttt{round}(u_f), \texttt{round}(v_f))$. A rendered location (u, v) is compatible with the frame if all of $\mathbf{p}_r$, $\mathbf{n}_r$, $\mathbf{p}_f$, and $\mathbf{n}_f$ are valid, if $||\mathbf{p}_r - \mathbf{p}_f|| < \theta_d$, and if the angle between $\mathbf{n}_r$ and $\mathbf{n}_f$ is less than θ_n. We use $\theta_d = 0.05$ and $\theta_n = 45°$.

For compatible points, the geometric error is defined via the point-to-plane ICP error as

$$\epsilon_g = w_g(\mathbf{p}_f - \mathbf{p}_r) \cdot \mathbf{n}_r \quad (5)$$

As distant points are noisier, we downweight their contribution using equation 1 by letting

$$w_g = \frac{\sigma_z(z_{min})}{\sigma_z(\mathbf{p}_r.z)} \quad (6)$$

The color error can be defined in a variety of color spaces. While our system supports an arbitrary number of color channel errors, we found simple intensity error to be effective. With frame intensity image Y_f and rendered intensity image Y_r, the color error is simply

$$\epsilon_c = (Y_f(u_f, v_f) - Y_r(u, v)) \quad (7)$$

The overall residual for a point is a weighted combination of these two quantities:

$$\epsilon = \lambda\epsilon_g + \epsilon_c \quad (8)$$

As they are in different units, a simple empirically set balancing factor $\lambda = 10$ causes both functions to contribute roughly equally to the total error.

The goal is relative pose correction $\mathbf{T}^\star$ with

$$\mathbf{T}^\star = \underset{\hat{\mathbf{T}}}{\text{argmin}} \sum ||\epsilon||^2 \quad (9)$$

Assuming we are in the correct convergence basin, we proceed with Gauss-Newton iterative nonlinear least squares

minimization, for which we need to obtain and solve the normal equation

$$\mathbf{J}^\intercal \mathbf{J} \Delta \mathbf{x} = -\mathbf{J}\epsilon \tag{10}$$

to obtain parameter updates $\Delta\mathbf{x}$.

Giving $\mathbf{p}_r = (x_p, y_p, z_p)^\intercal$ and $\mathbf{n}_r = (x_n, y_n, z_n)^\intercal$, there is one row of the Jacobian for each geometric and each color error term for each valid correspondence.

The geometric rows are computed as follows:

$$\mathbf{J}_{3D} = \begin{bmatrix} 1 & 0 & 0 & 0 & 2z_n & -2y_n \\ 0 & 1 & 0 & -2z_n & 0 & 2x_n \\ 0 & 0 & 1 & 2y_n & -2x_n & 0 \end{bmatrix} \tag{11}$$

$$\mathbf{J}_{rot} = \begin{bmatrix} 0 & 0 & 0 & 0 & 2z_n & -2y_n \\ 0 & 0 & 0 & -2z_n & 0 & 2x_n \\ 0 & 0 & 0 & 2y_n & -2x_n & 0 \end{bmatrix} \tag{12}$$

The geometric Jacobian row $\mathbf{J}_g$ is then

$$\mathbf{J}_g = \lambda w_g \left(-\mathbf{n}_r^\intercal \mathbf{J}_{3D} + (\mathbf{p}_f - \mathbf{p}_r)^\intercal \mathbf{J}_{rot}\right) \tag{13}$$

To compute the Jacobian rows for the color error terms, we need to use the chain rule to compute how the image error changes with respect to the parameters $\mathbf{x}$. We take our inspiration from [18] and [1], but the use of Gauss-Newton for direct dense image alignment goes back to [14]. We first need how the error changes with respect to the projection $(u_f, v_f)^\intercal$ of $\mathbf{p}_r$. This is linearly approximated using the gradient images of Y_f, which we compute using a normalized Sobel filter of Y_f to obtain G_x and G_y. We then compute how the projection changes with respect to 3D position:

$$\mathbf{J}_{proj} = \begin{bmatrix} \frac{f_x}{z_r} & 0 & \frac{-x_r f_x}{z_r^2} \\ 0 & \frac{f_y}{z_r} & \frac{-y_r f_y}{z_r^2} \end{bmatrix} \tag{14}$$

Putting this together with equation 11, the color Jacobian row is

$$\mathbf{J}_c = \begin{bmatrix} G_x(u_f, v_f) & G_y(u_f, v_f) \end{bmatrix} \circ \mathbf{J}_{proj} \circ \mathbf{J}_{3D} \tag{15}$$

The final Jacobian matrix J is $2n \times 6$ with a geometric and color row for each of the n valid correspondences. We compute the quantities $\mathbf{J}^\intercal\mathbf{J}$ and $-\mathbf{J}^\intercal\epsilon$ needed by equation 10 in parallel on the GPU, including parallel tree reduction using local memory to avoid global GPU memory writes. We solve the resulting 6×6 linear equation using Cholesky decomposition on the CPU. We apply the parameter update as $\mathbf{x} + \Delta\mathbf{x}$, and iterate until $||\Delta\mathbf{x}||_1 < \theta_\Delta$, where we use $\theta_\Delta = 0.0001$.

3.4. Segmentation and Patch Volume Fusion

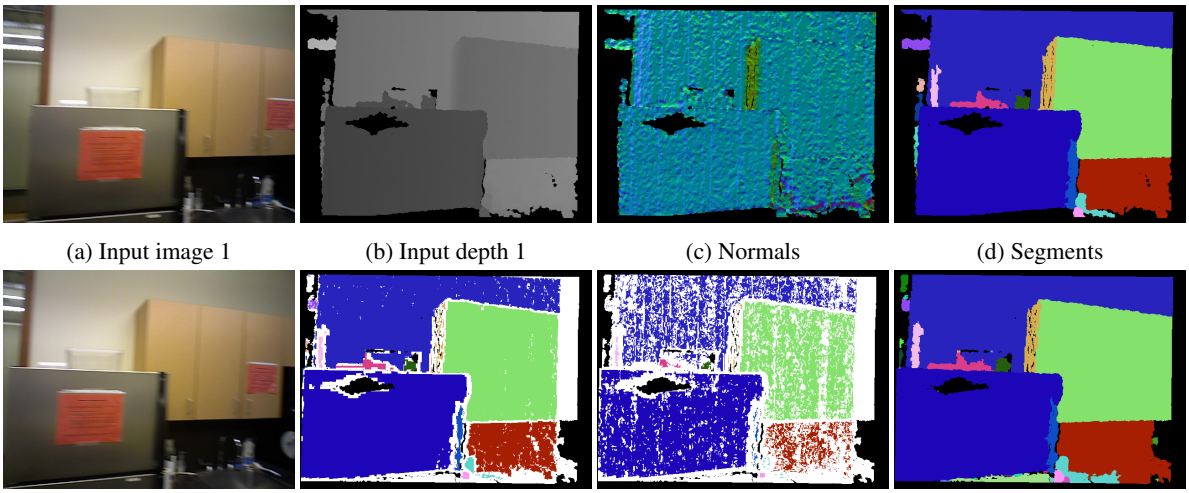
The patch volume framework allows for splitting up the scene in a wide variety of ways. We opt to use scene-dependent splitting based on locally planar geometric segmentation. This allows us to save memory by only allocating patch volumes for areas of the scene containing surfaces, and the patch volumes can be aligned with planar surfaces to avoid needless voxels in free space.

Our segmentation algorithm is inspired by the Felzenszwalb-Huttenlocher algorithm [9]. We create edges between each pixel (u, v) in D_f and its eight neighbors (u', v') on the same surface (see section 3.2), with edge weight $(1 - N_f(u, v) \cdot N_f(u', v'))$. We sort the edges by weight, meaning that pixels with more similar normals are considered first. Each pixel is initially in its own component, and components are merged using an efficient union-find data structure. We also maintain a map from components to normals, so when two components are merged, we can set the merged component normal to a size-weighted average of the merged components' normals. When each edge is considered for merging, the two associated components are merged only if the angle between current component normals is less than a threshold $\theta_{seg} = 30°$. After all edges have been considered, we eliminate spurious small segments by making another pass over the edges in order and merge two segments if either has size less than 1000.

For frames after the first, we wish to consistently assign points to the corresponding existing patch volume. To accomplish this, following alignment (section 3.3) but before segmentation, we project the rendered points from D_r, normals from N_r, and PV assignments S_r into the new frame. We consider a projection onto (u, v) *consistent* if the angle between $N_f(u, v)$ and the projected normal from N_r is less than θ_{seg} and the depth difference is less than $3\sigma_z$. We initialize components for segmentation with these consistent projections, and all remaining pixels start in their own unary components. Thus new pixels can be merged with existing segments by following the same segmentation algorithm. New components following segmentation will establish new patch volumes. An example of this segmentation procedure is shown in figure 1.

Once we have segmented a new frame, we must fuse the new data into the existing patch volumes. For segment components not corresponding to an existing PV, we axis-align the segment in 3D using the segment normal, and initialize a new PV with size sufficient to contain the segment plus a small border (0.01m). For existing segments corresponding to existing PVs, we expand the PV in each axis direction as required to contain the new points. If expansion is required, we expand by an additional border (0.1m) in anticipation of further needed expansion in that direction on subsequent frames. Note that while this can cause PVs to overlap in 3D space, the consistent projection segmentation maps each input pixel into at most one PV and restricts each PV to represent a locally planar piece of the scene.

We now have a segmentation which maps input pixels to existing PVs that contain the input points in space. To fuse new geometric measurements into $\mathbf{F}$ and $\mathbf{W_F}$ for a PV, we follow the techniques of [15]. In parallel on the GPU, each voxel $v_F \in \mathbf{F}$ and $v_{W_F} \in \mathbf{W_F}$ is transformed



(a) Input image 1 (b) Input depth 1 (c) Normals (d) Segments

(e) Input image 2 (f) Rendered segments (g) Consistent segments (h) Result segments

Figure 1: The first row shows an example of our segmentation on an initial frame. The second row shows the transfer of consistent segment labels via rendering of existing patch volumes. White pixels are singleton segments.

into the PV via $\mathbf{T}_G \circ \mathbf{T}_{PV}$ to obtain $\mathbf{p}_f$ and projected into the input frame using equation 4 obtaining the closest pixel $(u, v) = (\texttt{round}(u_f), \texttt{round}(v_f))$. The current voxel values are $d_{old} \leftarrow v_F$ and $w^d_{old} \leftarrow v_{W_F}$. We define

$$d = D_f(u, v) - \mathbf{p}_f.z \quad (16)$$

$$d_{min} = -3\sigma_z\left(D_f(u, v)\right) \quad (17)$$

$$d_{max} = 3\sigma_z\left(D_f(u, v)\right) \quad (18)$$

$$d_{new} = \begin{cases} null & : \quad d < d_{min} \\ d & : \quad d_{min} \leq d \leq d_{max} \\ d_{max} & : \quad d > d_{max} \end{cases} \quad (19)$$

In this, $null$ values represent unseen voxels and d_{max} represents empty voxels. To compensate for the additional axial and lateral space covered by distance measurements, we modify the weighting function of [16]:

$$w^d_{new} = \frac{\sigma_z(z_{min})}{\sigma_z(D_f(u, v))} \frac{z^2_{min}}{D_f(u, v)^2} \quad (20)$$

If $d_{new} \neq null$, we update the voxel values

$$v_F \leftarrow \frac{w^d_{old} d_{old} + w^d_{new} d_{new}}{w^d_{old} + w^d_{new}} \quad (21)$$

$$v_{W_F} \leftarrow \min(w^d_{old} + w^d_{new}, w_{max}) \quad (22)$$

where $w_{max} = 100$.

We also fuse new color information into $v_C \in \mathbf{C}$ and $v_{W_C} \in \mathbf{W_C}$ in a similar way. With color, we modify the fusion function to only update voxels near the surface and not in empty space. We also weight the color contribution based on the distance from the surface. We have previous values $c_{old} \leftarrow v_C$ and $w^c_{old} \leftarrow v_{W_C}$.

$$c_{new} = \begin{cases} null & : \quad |d| > d_{max} \\ C_f(u, v) & : \quad |d| \leq d_{max} \end{cases} \quad (23)$$

$$w^c_{new} = w^d_{new}\left(1 - \frac{|d|}{d_{max}}\right) \quad (24)$$

If $c_{new} \neq null$ (ensuring $w^c_{new} \geq 0$), update:

$$v_C \leftarrow \frac{w^c_{old} c_{old} + w^c_{new} c_{new}}{w^c_{old} + w^c_{new}} \quad (25)$$

$$v_{W_C} \leftarrow \min(w^c_{old} + w^c_{new}, w_{max}) \quad (26)$$

When performed for all PVs and their corresponding segments in the new frame, this completes the fusion of the new frame into the scene model.

One remaining issue is that patch volume expansion may cause PVs for large planar segments to grow to an unwieldy size, hampering dynamic GPU memory swapping. When a patch volume expands such that any dimension has voxel count larger than 256, we split the patch volume on that dimension.

3.5. Loop Closure and Global Optimization

A key issue with previous volumetric fusion mapping techniques is the inability to scale to larger environments. One issue is the large memory requirement of volumetric representations, which we address by representing the scene as patch volumes which can be moved in and out of GPU memory. Another issue, not addressed in previous work, is handling the inevitable drift that occurs during sequential

mapping, which becomes most apparent when when returning to a previously mapped areas of the scene after many intervening frames. This is the loop closure problem.

Our strategy is to divide patch volumes into two sets: $S_{current}$ and S_{old}. Only PVs in $S_{current}$ are used when performing standard alignment and fusion (sections 3.3 and 3.4). All PVs are initially in $S_{current}$. We track how many frames have passed since a PV was last in the render frustum for alignment. Once a PV has not been rendered for sequential alignment in over 50 frames, it is moved to S_{old}. Every 10 frames, after sequential alignment, we check for a loop closure by rendering all patch volumes in S_{old}. If the resulting rendering has valid points in more than half of the pixels, this is considered a loop closure detection, and we must ensure global consistency.

We might hope that the alignment procedure in section 3.3 would be sufficient to align our current camera pose with the old patch volumes, but the amount of drift accumulated since the PVs in S_{old} were last observed may well be large enough that we are not in the convergence basin of the local optimization. To mitigate this issue, we introduce the use of visual feature matching against keyframes to initialize the local alignment. As we perform sequential mapping, we cache features for a keyframe each time we accumulate a pose translational change of more than 0.5m or angular change of more than $30°$. We also store which PVs were in view of the keyframe, so we can map PVs back to keyframes which observed them. We compute FAST features [17] and BRIEF descriptors [3], associate them with their points from the depth image (or discard them if they lack valid depth), and save only this information for the keyframe to minimize memory usage.

When a loop closure is detected by rendering, but before alignment is run, we accumulate all keyframes that observed the PVs from S_{old} which fell into the render frustum. We filter these to only keyframes within 1.5m and $60°$ of the current pose estimate. Features and descriptors are generated for the current frame to be matched against these keyframes. Starting from the oldest such keyframe, we obtain purported matches for each feature in the current frame as the most similar descriptor in the keyframe. We then use RANSAC along with reprojection error to discard outliers and obtain a relative pose estimate relative to the keyframe. We accept the first relative pose obtained with at least 10 feature match inliers. Once we have an improved initial pose estimate relative to the PVs we rendered from S_{old}, we refine the camera pose relative to these PVs using the full dense alignment described in section 3.3. We have observed that the full dense alignment (given good initialization) provides better relative pose estimates than keyframe-based alignment alone.

Now that we have one belief about the camera pose relative to the sequentially rendered PVs from $S_{current}$ and another from the loop closure alignment against S_{old}, we must globally minimize the disagreement error. To do this, we use techniques from pose-graph optimization, utilizing the G2O library [12]. We maintain a graph with a vertex for each camera and a vertex for each PV. Each new frame, we add a relative pose edge between the new camera and each PV observed by that camera, enforcing their relative pose following alignment. Following a detected loop closure, relative pose edges are added from the new camera to the PVs from S_{old} enforcing their relative pose from loop closure alignment. Optimizing the G2O pose graph for a few iterations (5 in our implementation) converges, achieving a globally consistent scene model.

Because we wish to maintain the flexibility of adjusting PVs relative to each other, we do not currently merge PVs from S_{old} into their overlapping counterparts in $S_{current}$. Merging overlapping PVs can be accomplished through weighted addition of the underlying voxel values, but determining when to perform this operation is an open problem.

4. Results

We evaluate various components of our system on indoor sequences recorded with a hand-held Asus Xtion Pro Live camera. We use the OpenNI2 API, which allows for hardware time synchronization and depth-to-image registration for each frame, as well as the ability to disable automatic exposure and white balance. Though the goal is real-time live reconstruction, we currently process the files offline. We use only every third frame, corresponding to input at 10 frames per second. We use a voxel resolution of 1cm^3.

Our test system is an Intel Xeon E5530 4-core 2.4 GHz machine with 12GB of RAM and an NVidia GTX 560 Ti graphics card with 1GB video RAM.

4.1. Geometry and Color

Our first result shows that both the depth and color error terms are needed to achieve the best alignment. We use a single PV with 256 voxels per side with no loop closure for this example to highlight the properties of the alignment algorithm. The sequence consists of 157 processed frames of a painting on a wall. The running time averaged 122ms per frame on the GPU. As a comparison, disabling the GPU and running the OpenCL code on the CPU yielded a running time of 1100ms per frame.

We ran our algorithm using just the geometric term (equation 5). As the scene consists primarily of a flat wall which does not sufficiently constrain point-to-plane ICP, this failed within the first few frames, and is not shown. Next we ran our algorithm using only the color error term (equation 7). This also performed poorly, as the lack of geometric constraint caused some poor alignments. Once the geometry of the model is inconsistent, subsequent results are even worse as the model is projecting incorrect colors. In comparison, our full alignment produces a visually accu-

(a) Color Only (b) Full Alignment View 1 (c) Full Alignment View 2

Figure 2: This example demonstrates the need for both color and depth in alignment. Figure (a) shows the failure when only the color term is used. Using only the geometric term failed immediately and is not shown. Figures (b) and (c) show two views of the model using our full alignment.

rate model with no obvious alignment failures through the entire sequence. See figure 2 for a comparison of these results.

4.2. Global Consistency

Our next result showcases the need for global consistency through loop closure, and how our patch volume representation allows us to achieve this. We recorded a sequence in a medium sized room, performing roughly one full rotation from within the center of the room. This sequence consists of 248 processed frames. The final model consists of 141 patch volumes requiring 952MB of memory, which did not fit simultaneously in (free) GPU memory, requiring our dynamic patch volume swapping. The total time per frame averaged 616ms, including the loop closure checks in the same thread every 10 frames. Approximately a third of this time is spent on patch volume fusion, where the segmentation on the CPU and periodic patch volume expansion and splitting are both computationally expensive. We are confident that with additional optimization and placing loop closure in a parallel thread, real-time performance can be achieved.

In figure 3, we compare the result in the overlapping portion of the sequence. Note that without loop closure, the alignment does not fail catastrophically, because the sequential alignment is carried out only over frames in $S_{current}$. However, there is a clear global inconsistency, with duplicate instances of pictures on the wall. In comparison, detecting loop closures and performing graph optimization over patch volumes produces a consistent model. Please see an overview of the final model in figure 4, where one can observe that the overall model is not adversely affected by the graph optimization.

A video showing these results is present in the supplementary material and online[4].

[4] http://youtu.be/hEUDRTTlCxM

5. Conclusion

We have presented a system based on a novel multiple volume representation called patch volumes which combines the advantages of volumetric fusion with the ability to generate larger-scale globally consistent models.

In future work, we will optimize towards true real-time operation over yet larger settings. We will explore multi-scale representations for memory efficiency, texture mapping for visual accuracy, and modifications to the framework to handle objects in dynamic environments.

Acknowledgements

We are grateful to Richard Newcombe for his guidance during the development of our system. This work was funded in part by an Intel grant and by the Intel Science and Technology Center for Pervasive Computing (ISTC-PC).

References

[1] C. Audras, A. I. Comport, M. Meilland, and P. Rives. Real-time dense appearance-based SLAM for RGB-D sensors. *Australian Conference on Robotics and Automation*, 2011. 2, 4

[2] P. J. Besl and N. D. McKay. A Method for Registration of 3-D Shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 14(2), 1992. 1

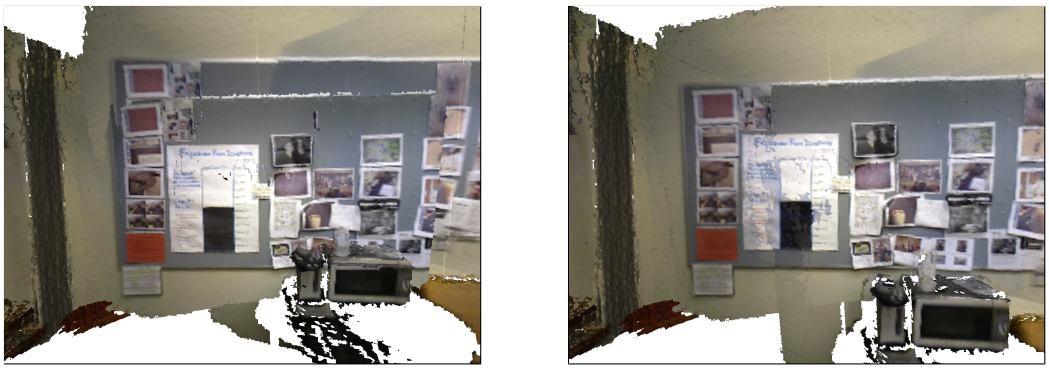
[3] M. Calonder, V. Lepetit, and C. Strecha. BRIEF : Binary Robust Independent Elementary Features. *European Conference on Computer Vision (ECCV)*, 2010. 6

[4] Y. Chen and G. Medioni. Object Modeling by registration of multiple range images. In *International Conference on Robotics and Automation (ICRA)*, 1991. 1

[5] B. Curless and M. Levoy. A volumetric method for building complex models from range images. *SIGGRAPH*, 1996. 1, 2

[6] H. Du, P. Henry, X. Ren, M. Cheng, D. B. Goldman, S. M. Seitz, and D. Fox. Interactive 3D Modeling of Indoor Environments with a Consumer Depth Camera. *UbiComp*, 2011. 1

[7] F. Endres, J. Hess, N. Engelhard, J. Sturm, D. Cremers, and W. Burgard. An Evaluation of the RGB-D SLAM System.



(a) No Loop Closure (b) With Loop Closure

Figure 3: This example shows the need for loop closure to achieve global consistency. Figure (a) shows the overlapping region of the sequence if no loop closure is performed. Note the drift has caused serious misalignment. In comparison, figure (b) shows the globally consistent result when we perform graph optimization over the patch volumes.

Figure 4: An overview of the final result and the patch volumes used.

IEEE International Conference on Robotics and Automation (ICRA), 3(c):1691–1696, May 2012. 1

[8] N. Engelhard, F. Endres, J. Hess, J. Sturm, and W. Burgard. Real-time 3D Visual SLAM with a Hand-held RGB-D Camera. *Proceedings of the RGB-D Workshop on 3D Perception in Robotics at the European Robotics Forum*, 2011. 1

[9] P. F. Felzenszwalb and D. P. Huttenlocher. Efficient Graph-based Image Segmentation. *International Journal of Computer Vision (IJCV)*, 59(2):167–181, Sept. 2004. 4

[10] P. Henry, M. Krainin, E. Herbst, X. Ren, and D. Fox. RGB-D Mapping: Using Depth Cameras for Dense 3D Modeling of Indoor Environments. *International Symposium on Experimental Robotics (ISER)*, 2010. 1

[11] P. Henry, M. Krainin, E. Herbst, X. Ren, and D. Fox. RGB-D mapping: Using Kinect-style depth cameras for dense 3D modeling of indoor environments. *The International Journal of Robotics Research (IJRR)*, 31(5):647–663, Feb. 2012. 1

[12] R. Kuemmerle, G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard. g2o: A General Framework for Graph Optimization. *International Conference on Robotics and Automation (ICRA)*, 2011. 1, 6

[13] W. E. Lorensen and H. E. Cline. Marching Cubes: A High Resolution 3D Surface Construciton Algorithm. *SIGGRAPH*, 21(4):163–169, 1987. 2

[14] B. D. Lucas and T. Kanade. An Iterative Image Registration Technique with an Application to Stereo Vision. *International Conference on Artificial Intelligence (IJCAI)*, 1981. 4

[15] R. A. Newcombe, D. Molyneaux, D. Kim, A. J. Davison, J. Shotton, S. Hodges, and A. Fitzgibbon. KinectFusion: Real-Time Dense Surface Mapping and Tracking. *International Symposium on Mixed and Augmented Reality (ISMAR)*, 2011. 1, 2, 3, 4

[16] C. V. Nguyen, S. Izadi, and D. Lovell. Modeling Kinect Sensor Noise for Improved 3D Reconstruction and Tracking. *2012 Second International Conference on 3D Imaging, Modeling, Processing, Visualization & Transmission (3DIM/3DPVT)*, pages 524–530, Oct. 2012. 3, 5

[17] E. Rosten and T. Drummond. Machine Learning for High-speed Corner Detection. *European Conference on Computer Vision (ECCV)*, pages 430–443, 2006. 6

[18] F. Steinbrücker, J. Sturm, and D. Cremers. Real-Time Visual Odometry from Dense RGB-D Images. *Workshop on Live Dense Reconstruction with Moving Cameras at the International Conference on Computer Vision (ICCV)*, 2011. 2, 4

[19] T. Whelan, H. Johannsson, M. Kaess, J. J. Leonard, and J. Mcdonald. Robust Real-Time Visual Odometry for Dense RGB-D Mapping. *International Conference on Robotics and Automation (ICRA)*, 2013. 2

[20] T. Whelan, J. McDonald, M. Kaess, M. Fallon, H. Johannsson, and J. J. Leonard. Kintinuous: Spatially Extended KinectFusion. *3rd RSS Workshop on RGB-D: Advanced Reasoning with Depth Cameras*, 2012. 1, 2