SLAMRecon: A Real-time 3D Dense Mapping System

Hao Li & Huayong Xu

Interdisciplinary Research Center in Shandong University

Contents

1 Introduction

SLAMRecon is a real-time 3D dense mapping system based on RGB-D camera. The output is a reconstructed indoor scene model. As we know, there are some previous 3D Dense Mapping Systems, like Kinect Fusion[Kinect Fusion], Voxel Hashing[Voxel Hashing] and InfiniTAM[InfiniTAM]. These works use ICP to do registration between adjacent two frames. And ICP will fail when scan some flat area. However, these works don't do any local or global optimization for camera poses. During scan, the reconstruction will drift because of accumulated errors. In our work, we used ORB-SLAM2[ORB-SLAM] to help us estimate camera pose of each frame. Because of orb-slam's local and global optimization, we can avoid obvious drift when scan an indoor scene. We use an integration and re-integration framework to handle changing camera poses. Once a camera pose changes because of optimization, the system will do re-integration process to remove data influenced by old camera pose from scene volume and integrate data based on new pose to the scene volume.

2 System Overview

In this section, we show an overview to introduce a basic pipeline of our system.

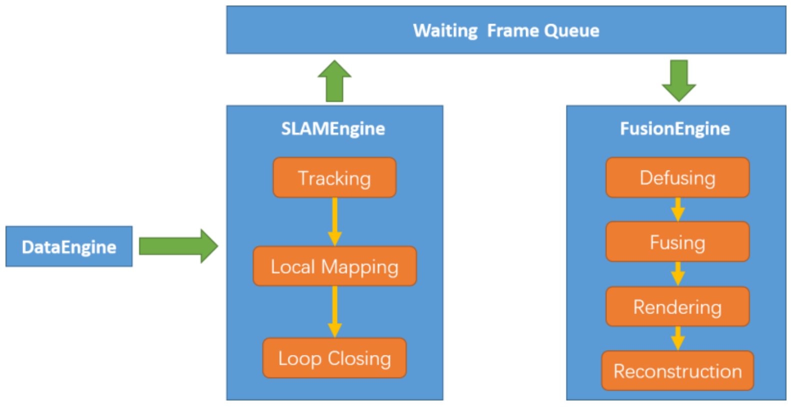


Fig.2-1. An overview of SLAMRecon

Our system, see an overview in Fig.2-1, incorporates three main modules: DataEngine, SLAMEngine and FusionEngine. More specifically, DataEngine takes charge of acquiring RGB-D images from different sources, such as RGB-D camera, RGB-D images stored on disk, a remote server which can capture a sequence of RGB-D images. Moreover, DataEngine stores each RGB-D frame in a pre-allocated memory block, which will be used in later process.

SLAMEngine processes new RGB-D frames from DataEngine to get associated camera pose related to world coordinate system of each frame. We use ORB-SLAM2[ORB-SLAM] in SLAMEngine, which is made up by three threads that run in parallel: tracking, local mapping and loop closing. It will be introduced detailedly in section 3.2. During process of SLAMEngine, the camera pose of each frame may update sometimes because of orb-slam's local and global optimization. Once the camera pose of special frame is updated, the frame with info about frame id, old camera pose (camera pose before updated), new camera pose (camera pose after updated) will be pushed into a waiting frame queue. In fact, the waiting frame queue is a map structure which can be retrieved by a key (frame id). In order to increase processing speed of the system, before a frame is pushed into the queue, the system first check if the frame exists in the queue, if so, replace the old info in the frame, if not, push the frame into the queue. And meanwhile, the new frame has priority over updated frame to provide better user's interaction. Because fusion and defusion will consuming a large amount of computing resources, in order to ensure real time, the system only do defusion for frames with large changes. If only have small changes of camera pose, the system will ignore these frames to save computing resources, but have no impact for user's interaction.

FusionEngine is mainly in charge of defusing and fusing data of frames in waiting frame queue one by one. Here, defusing means removing data influenced by old camera pose from a scene volume, while fusing means integrating data based on new pose to the scene volume. As introduced in [Voxel Hashing], the scene volume is organized by voxel hashing structure. If the picked frame from the queue is a new frame, FusionEngine only executes fusing process. In order to show quality of reconstruction, FusionEngine also renders surface of the scanned scene under the current camera view and a specified view by user's interaction. After finishing scanning, FusionEngine will re-fuse all frames with newest camera pose to get more accurate fusion result. and employ marching cubes algorithm to extract final surface.

3 Technical Details

3.1 DataEngine

We have briefly introduced DataEngine in system overview. In this section, we will show more details about this module.

3.1.1 Architecture

DataEngine is split into two layers as showed in Fig.3-1. The topmost abstract layer is accessed by other modules and consists some blank interfaces and some common variables. The interfaces and members of abstract layer is listed here:

```
class DataEngine
{
public:
	typedef std::shared_ptr<DataEngine> Ptr;
	typedef std::shared_ptr<DataEngine const> ConstPtr;

	DataEngine();
	virtual ~DataEngine() {}

	virtual bool hasMoreImages(void) = 0;
	virtual bool getNewImages(void) = 0;
	UChar4Image* getCurrentRgbImage();
	ShortImage* getCurrentDepthImage();

	cv::Mat getCurrentMatRgbImage();
	cv::Mat getCurrentMatDepthImage();

	Vector2i getDepthImageSize();
	Vector2i getRGBImageSize();

	UChar4ImagesBlock *getRgbImagesBlock();
	ShortImagesBlock *getDepthImagesBlock();
```

```
    int getCurrentFrameId();

    void readCameraPoses(std::string filename);
    const std::vector<Matrix4f>& getAllCameraPoses() const;
    const std::vector<int>& getAllFlags() const;
    const Matrix4f getCameraPoses(int i) const;

protected:
    UChar4Image *rgbImage;
    ShortImage *rawDepthImage;
    UChar4ImagesBlock *rgbImagesBlock;
    ShortImagesBlock *depthImagesBlock;

    cv::Mat matRgbImage;
    cv::Mat matDepthImage;

    int image_width;
    int image_height;
    int curFrameId;

    std::vector<Matrix4f> allCameraPoses;
    std::vector<int> allFlags;
};
```

The blank interfaces are implemented in the Implementation Layer. We have implemented three subclass of DataEngine: OpenNIEngine, FileReaderEngine and RemoteDataEngine. OpenNIEngine is in charge of acquiring RGB-D images from a RGB-D camera that supports openNI library. FileReaderEngine can read RGB-D images stored on disk in a special format. RemoteDataEngine receives data by communicating eith a remote server which can capture a sequence of RGB-D images. For a new data source, it is easy to write a new subclass to implement blank interfaces in DataEngine.

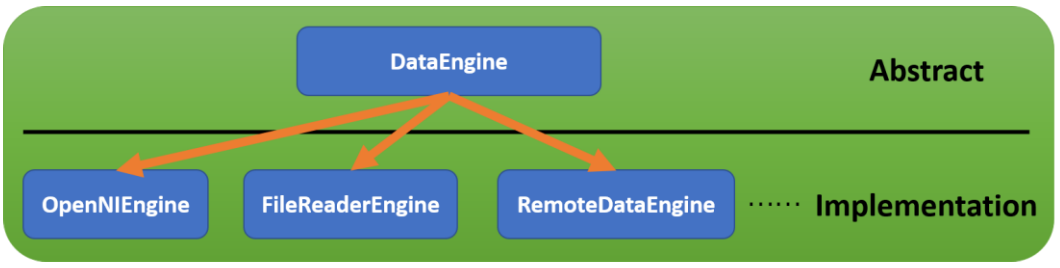


Fig.3-1. Architecture of dataEngine

3.1.2 Data Saving

For each frame, we store the depth data on the memory in order to do the defusion operation. All frames are stored on a linear memory block, the depth data is saved one by one according to frame id, and can be retrieved easily by frame index. See Fig.3-2.

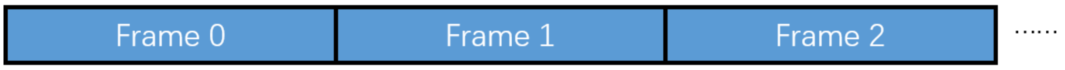


Fig.3-2. Depth image block

3.2 SLAMEngine

A general outline of SLAMEngine has already be given in section 2. In this section, we will introduce each processing stage in detail. It should be point out that SLAMEngine is developed based on ORB-SLAM[ORB-SLAM], more information about ORB-SLAM can be found on its project homepage: http://webdiis.unizar.es/~raulmur/orbslam/. Our system only could process RGB-D image data, which ORB-SLAM could process Monocular, Stereo and RGB-D image data.

Fig.3-3 shows the overview of SLAMEngine. There are three stages in SLAMEngine process: tracking, local mapping and loop closing.

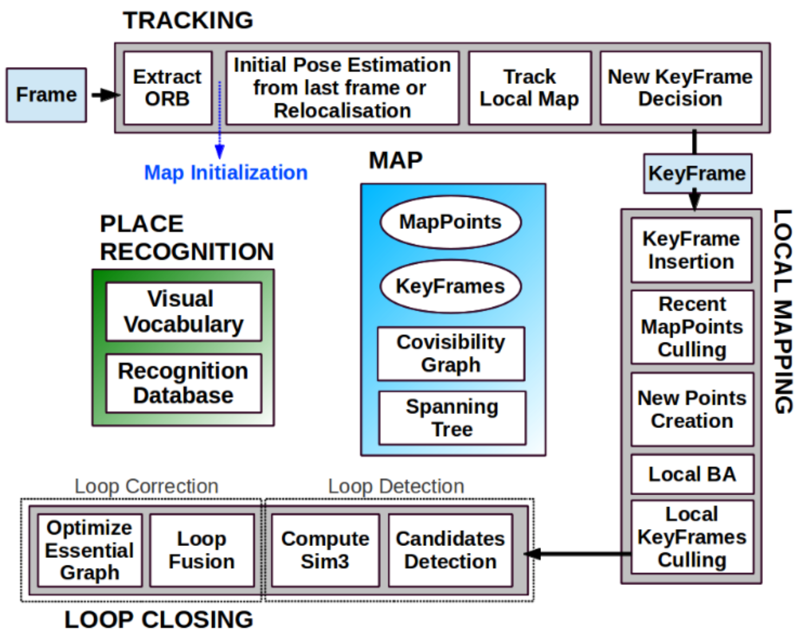


Fig.3-3. An overview of SLAMEngine

3.2.1 Tracking

There are five stages in tracking process: orb extraction, initial pose estimation from previous frame, initial pose estimation via global relocation, track local map and new keyframe decision. More details can be found in paper[ORB-SLAM].

3.2.2 Local Mapping

There are five stages in local mapping process: keyframe insertion, recent map points culling, new map point creation, local bundle adjustment and local keyframe culling. More details can be found in paper[ORB-SLAM].

3.2.3 Loop Closing

There are four stages in loop closing process: loop candidates detection, compute the similarity transformation, loop fusion and essential graph optimization. More details can be found in paper[ORB-SLAM].

3.2.4 Optimization

There are three types of optimization in SLAMEngine process: Bundle Adjustment(motion-only BA, local BA, global BA), Pose Graph Optimization and Relative Sim(3) Optimization. More details can be found in paper[ORB-SLAM].

motion-only BA

In pose optimization or motion-only BA process, all map points are fixed and only current frame’s camera pose is optimized. And pose optimization is used on every frame, and will be used many times to continuously optimize current frame’s camera pose. If map points are changed by other operations, the system will use pose optimization to optimize camera pose.

3.3 FusionEngine

A general outline of FusionEngine has already be given in section 2. In this section, we will introduce each processing stage in detail. It should be point out that FusionEngine is developed based on InfiniTAM[InfiniTAM], more information about InfiniTAM can be found on its project homepage: http://www.robots.ox.ac.uk/~victor/infinitam/. Before read following section, we suggest you to learn concept about TSDF by reading paper[Kinect Fusion].

3.3.1 Scene Volume Representation

We use a voxel hashing structure which is first proposed by Nießner et al. to represent the scene volume. We use the implementation of this method in InfiniTAM.

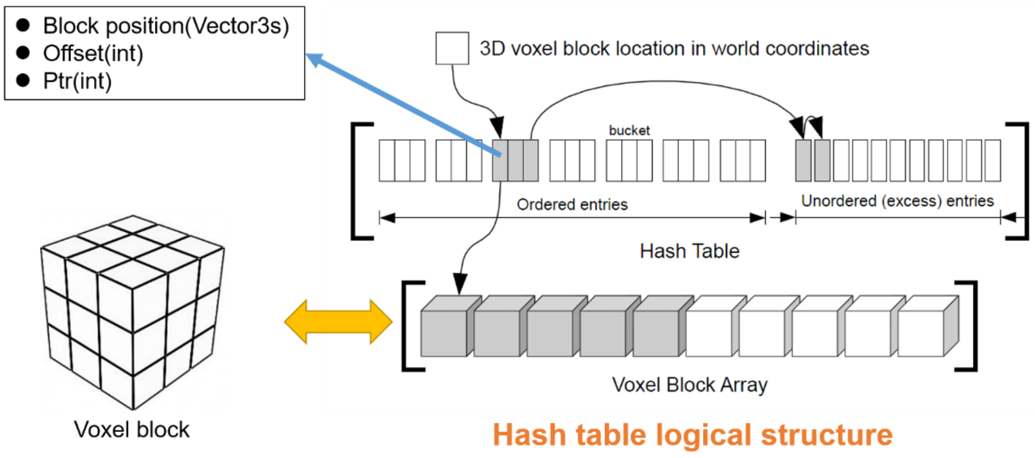

● Block position(Vector3s)
● Offset(int)
● Ptr(int)
3D voxel block location in world coordinates
bucket
Ordered entries
Unordered (excess) entries
Hash Table
Voxel Block Array
Voxel block
Hash table logical structure

Fig.3-4. Voxel Block Retrieval & Insertion by Hash Table

Fig.3-4 shows a basic voxel hashing structure. It incorporates a hash table and voxel block array, in which voxel block array is a sequential memory block and each bucket in hash table may have several entries to access a voxel block in the memory. The hash table consists ordered entries and unordered entries to avoid the entries conflict. Each voxel block has N*N*N voxels, which are stored on memory linearly. Here is the structure of each voxel:

```
struct FEVoxel_s
{
    short sdf; //Value of the truncated signed distance.
    short w_depth; //Number of fused observations that make up sdf.
};
```

The structure of hash entry is listed here:

```
struct FEHashEntry
```

```
{
        Vector3s pos; // Position of the corner of the NxNxN volume.
        int offset; // Offset in the excess list.
        /** Pointer to the voxel block array.
                - >= 0 identifies an actual allocated entry in the voxel block array
                - -1 identifies an entry that has been removed (swapped out)
                - <-1 identifies an unallocated block
        */
        int ptr;
};
```

The main operation of hash table are retrieval and insertion. We will describe these operations next.

Insertion

To insert new hash entries, we first evaluate the hash function and determine the target bucket. We then iterate over all bucket elements including possible lists attached to the last entry. If we find an element with the same world space position we can immediately return a reference. Otherwise, we look for the first empty position within the bucket. If a position in the bucket is available, we insert the new hash entry. If the bucket is full, we append an element to unordered entries list (see Fig. 3-4).

Retrieval

To read the hash entry for a query position, we compute the hash value and perform a linear search within the corresponding bucket. If no entry is found, and the bucket has an unordered entries list associated (the offset value of the last entry is set), we also have to traverse this list. The retrieval operation stops until the searched hash entry's offset value is less than -1 (see Fig. 3-4).

More details about voxel hashing can be got from these related papers: [Voxel Hashing][InfiniTAM].

3.3.2 Fusion

There are two stages in fusion process: allocation and integration. In allocation stage, new voxel blocks are allocated as required and a list of all visible voxel blocks is built based on current depth image. In integration stage, the current depth frame is integrated into the scene volume. Next, we will introduce these two stages in detail.

Allocation

1. Back projection. We cast rays from the optical center through each pixel in current depth image. According to camera's intrinsic calibration parameters, extrinsic calibration parameters and the depth value, we can compute the associated 3D point for each depth pixel. We can get another two 3D points along each casting ray within a distance threshold μ relative to computed depth-associated 3D point. See Fig.3-5.

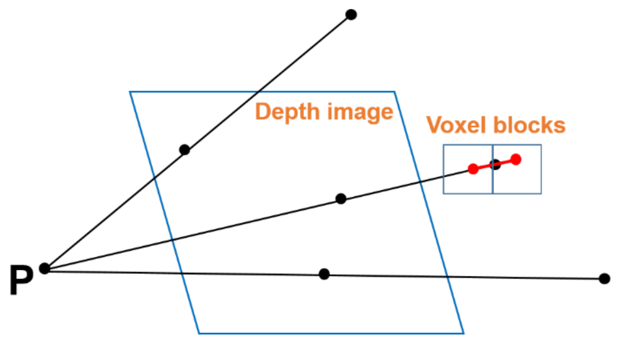
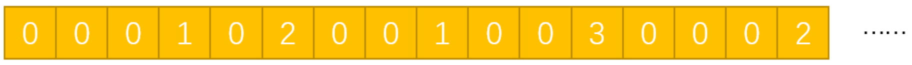


Fig.3-5. Back projection

2. For each 3D point computed in step1, we compute the associated intersection block. For each intersection block, retrieving it from hash table, if it doesn't exist, insert a new entry to hash table.
3. Build visible table. The number of elements in visible table is equal to total number of entries in hash table and is one-to-one. Each element mark if the voxel block that its associated entry point to is visible in current view. To build this visible table, we consider three kinds of situations. For blocks that are visible both in last view and current view, we set the elements value in visible table as 3. For intersection blocks that haven't been allocated before or have already been allocated but haven't been removed, we set the elements value in visible table as 1. For intersection blocks that have already been allocated and have been removed, we set the elements value in visible table as 2. For other invisible blocks, we set the elements value in visible table as 0. See Fig.3-6.

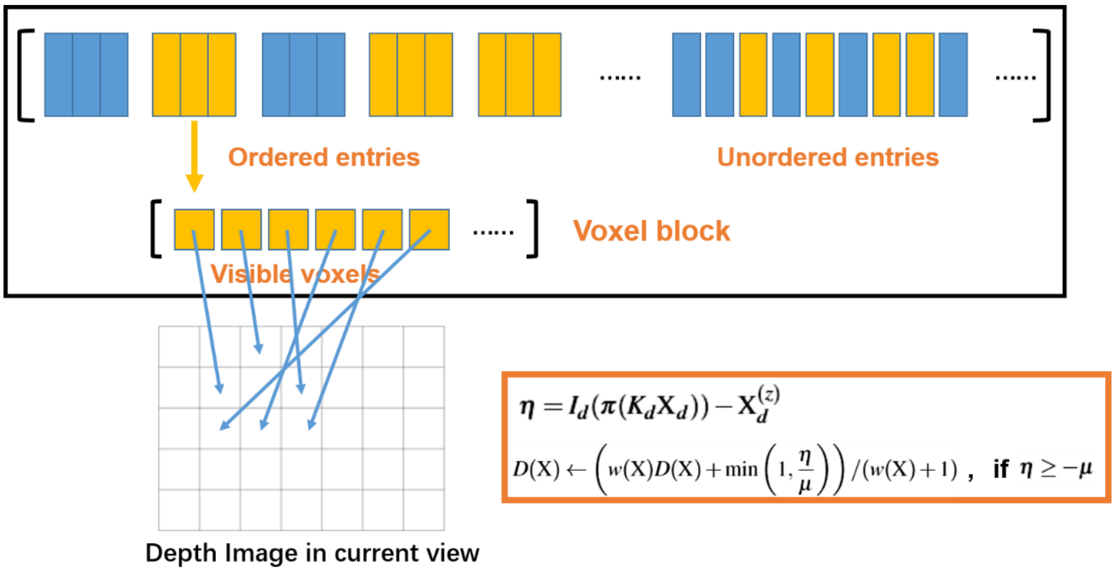


0	0	0	1	0	2	0	0	1	0	0	3	0	0	0	2	……

Fig.3-6. Visible table

4. Save visible table as a binary list to the memory block. We use a bit to represent each element in visible table. For elements whose values are greater than 0, we set corresponding elements in binary visible list as 1. Otherwise, we set 0. The binary visible list is stored in a pre-allocated linear memory block, which can be retrieved by frame index.

Integration



……
……
Ordered entries
Unordered entries
……
Voxel block
Visible voxels
$\eta = I_d(\pi(K_d X_d)) - X_d^{(z)}$
$D(X) \leftarrow \left(w(X)D(X) + \min\left(1, \frac{\eta}{\mu}\right) \right) / (w(X)+1)$, if $\eta \geq -\mu$
Depth Image in current view

Fig.3-7. Integration: project all visible voxels to the depth image and update voxel values.

As showed in Fig.3-7, given visible table in allocation stage, we search visible blocks by visible entries. Then we project all centers of voxels in all visible blocks to depth image in current view according to camera's intrinsic and extrinsic calibration parameters. After that, we update T-SDF value in each voxel. See formulas (3.1, 3.2).

$$\eta = I_d(\pi(K_d X_d)) - X_d^{(z)} \tag{3.1}$$

$$D(X) \leftarrow \left(w(X)D(X) + \min\left(1, \frac{\eta}{\mu}\right) \right) / (w(X)+1) \text{ , if } \eta \geq -\mu \tag{3.2}$$

In formula (3.1), K_d is camera's intrinsic parameters, X_d is center of a voxel, π computes inhomogeneous 2D coordinates from the homogeneous ones and the superscript (z) selects the Z-component of X_d. I_d is pixel values in depth image.

In formula (3.2), D(X) is the T-SDF value, w is a field counting the number of observations in the running average, which is simultaneously updated and capped to a fixed maximum value, u is a threshold for truncated signed distance function(TSDF).

More details can be found in [Voxel Hashing][InfiniTAM].

3.3.3 Defusion

There are also two stages in defusion process: updating visible table and repealing. In updating visible table stage, we build a visible table for current processed frame. In repealing stage, we will remove data influenced by old camera pose from a scene volume. Next, we will introduce these two stages in detail.

Updating visible table

1. Retrieving binary visible list in memory block by current processed frame's index.
2. Transform the binary visible list to the entries visible table as described before. Traverse the binary visible list, for bit set as 1, we set corresponding elements in visible table as 1. Otherwise, we set 0.

Repealing

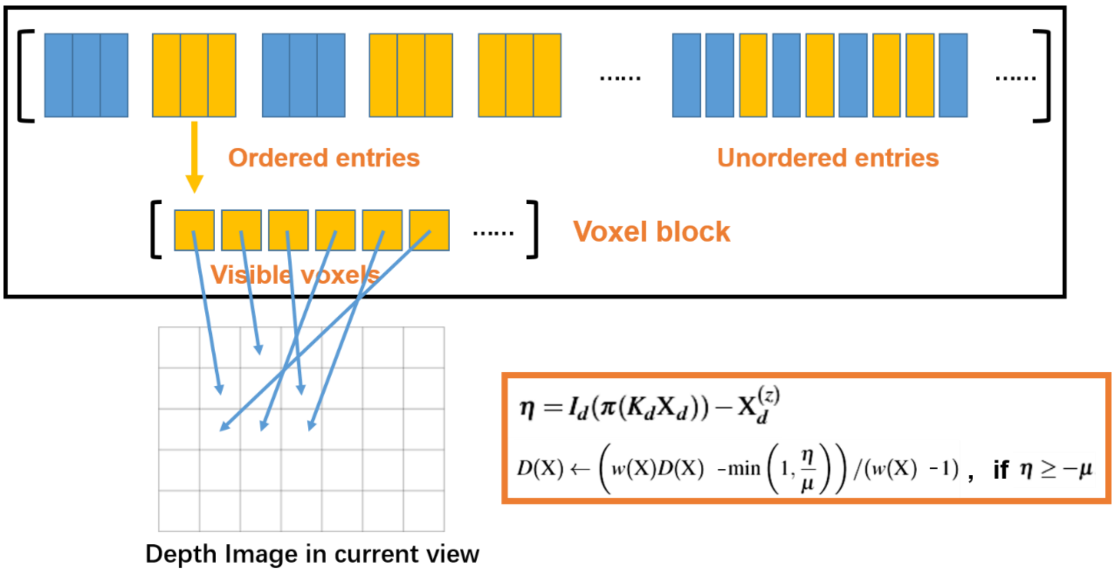

......
......
Ordered entries
Unordered entries
......
Voxel block
Visible voxels
$\eta = I_d(\pi(K_d X_d)) - X_d^{(z)}$
$D(X) \leftarrow \left(w(X)D(X) \ -\min\left(1, \frac{\eta}{\mu}\right)\right) / (w(X) \ -1)$, **if** $\eta \geq -\mu$
Depth Image in current view

Fig.3-8. Repealing: project all visible voxels to the depth image and update voxel values.

As showed in Fig.3-8, given visible table in updating visible table stage, we search visible blocks by visible entries. Then we project all centers of voxels in all visible blocks to depth image in current view according to camera's intrinsic and extrinsic calibration parameters. After that, we update T-SDF value in each voxel. See formulas

(3.3, 3.4).

$$\eta = I_d(\pi(K_d X_d)) - X_d^{(z)} \tag{3.3}$$

$$D(X) \leftarrow \left(w(X)D(X) \ - \min\left(1, \frac{\eta}{\mu}\right) \right) / (w(X) \ -1) \text{ , } \textbf{if } \eta \geq -\mu \tag{3.4}$$

Formula (3.3) is the same as Formula (3.1).

The parameters in formula (3.4) is the same as formula (3.3).

3.3.4 Rendering

We perform ray casting to render the image in current camera view. More specifically, we cast rays from the optical centre through each pixel and find their first intersection with the observed 3D surface in scene volume. In its simplest form, this requires evaluating the SDF along all points of the ray. However, as already stated in [Kinect Fusion], the values D(X) in the T-SDF give us a hint about the minimum distance to the nearest surface and allow us to take larger steps. Detail procedure is showed in Fig.3-10. More details can be found in paper[InfiniTAM].

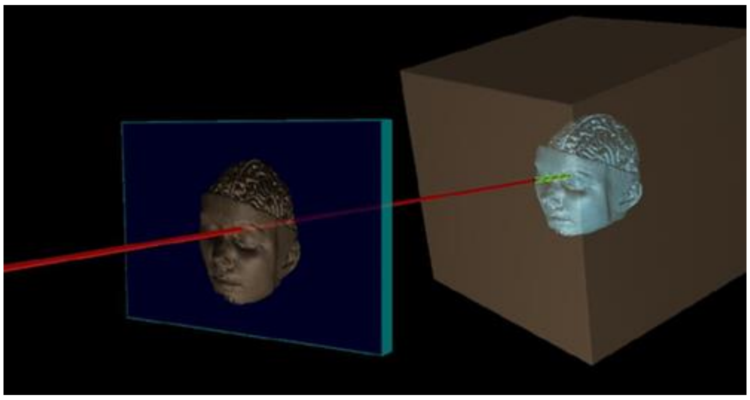


Fig.3-9. Ray casting.

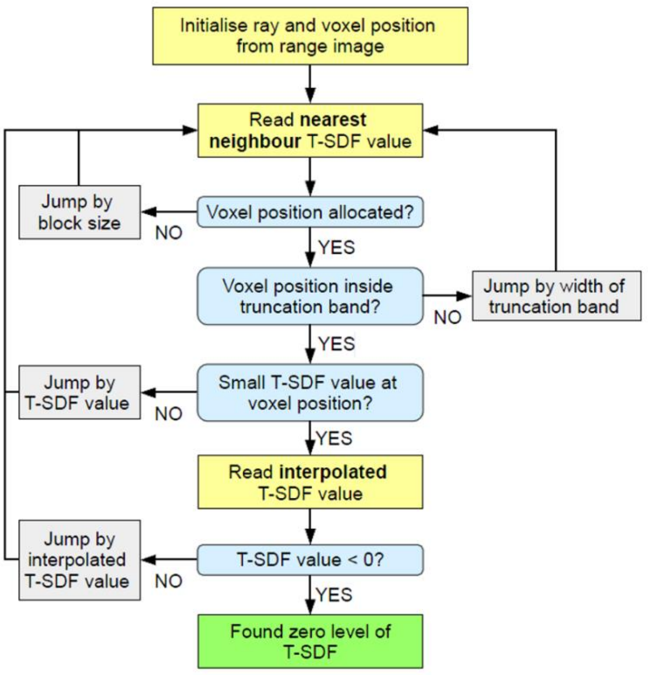

Initialise ray and voxel position from range image
Read nearest neighbour T-SDF value
Jump by block size
Voxel position allocated?
NO
YES
Voxel position inside truncation band?
Jump by width of truncation band
NO
YES
Jump by T-SDF value
Small T-SDF value at voxel position?
NO
YES
Read interpolated T-SDF value
Jump by interpolated T-SDF value
T-SDF value < 0?
NO
YES
Found zero level of T-SDF

Fig.3-10. Flow chart of the ray casting procedure.

3.3.5 Reconstruction

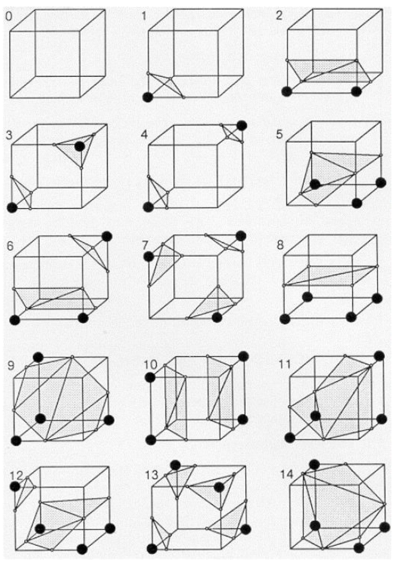


Fig.3-11. Marching cubes.

After finishing scanning, the last step is to extract geometry surface of the scanned scene. We use a triangular mesh to represent the reconstructed surface. To extract the isosurface, we employ the marching cubes algorithm on the scene volume. In the scene volume, each center of voxel is regarded as a corner of a cube, and T-SDF value represent the distance between the corner and the isosurface. Eight adjacent voxels compose a cube. Since there are eight vertices in each cube and two states, inside and outside, there are only 2^8 = 256 ways that a surface can intersect the cube, which can be reduce to 14 patterns. See Fig.3-11. By enumerating these 256 cases, we create a table to look up surface-edge intersections, given the labeling of a cubes' vertices. The table contains the edges intersected for each case. More details can be found in paper[Marching cubes].

4 Conclusions

SLAMRecon is a real-time 3D dense mapping system based on RGB-D camera which can avoid the problem about drift and loop closure when scanning compared with previous systems. The system is developed based on ORB-SLAM2[ORB-SLAM] and InfiniTAM[InfiniTAM], we thank authors of these two systems to open source their code. No system is perfect. This is the first version of our code and it may have some problems. We may offer more robust system in future.

We welcome developers to do further research based on our SLAMRecon system and improve our system.

5 User Guide

In this section, we will introduce how to use the SLAMRecon system in details.

5.1 Running the system

In this section, we will introduce how to use the SLAMRecon system in details.

1. Run the system, you can get the below interface. See Fig.5-1.

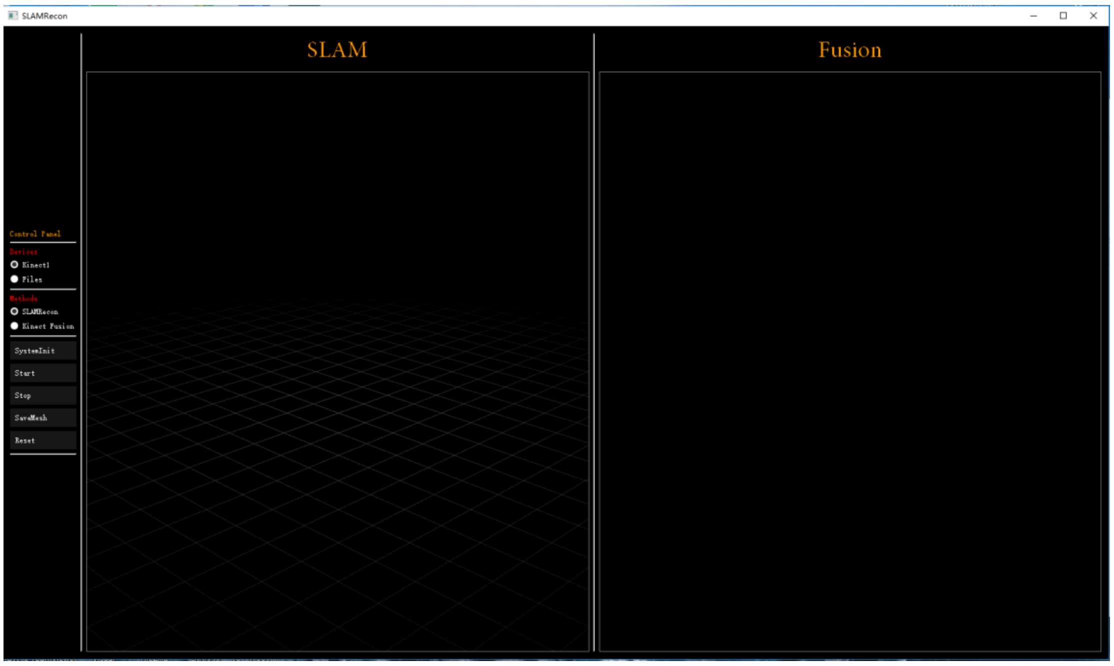
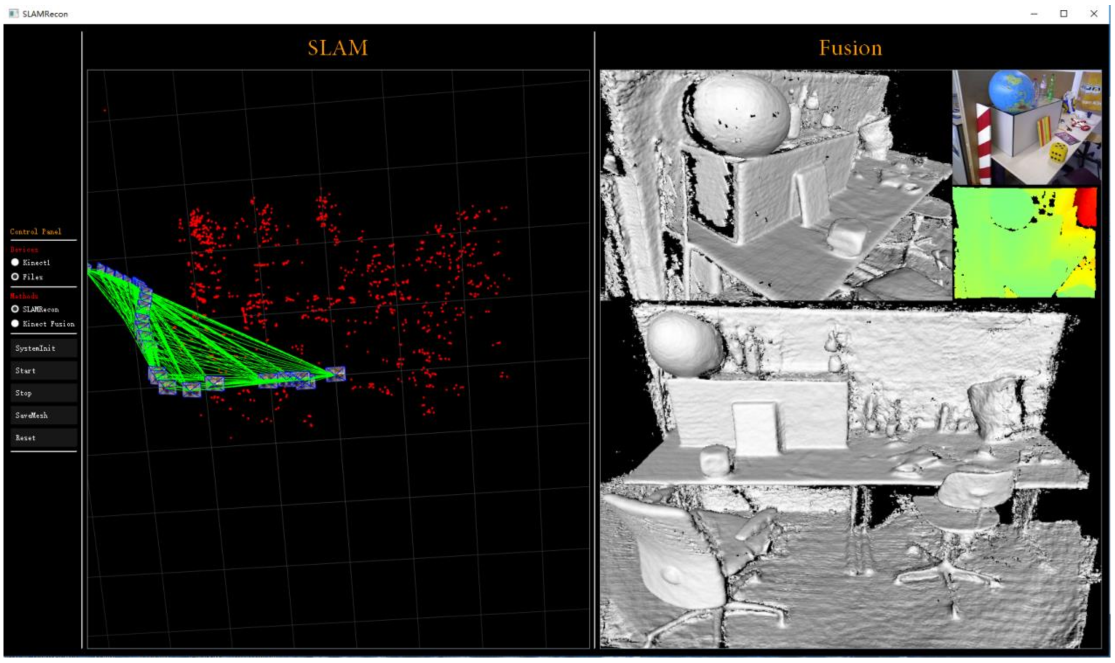


Fig.5-1. User interface.

2. Initialization. For RGB-D images stored on disk, choose Files option and for RGB-D camera(Kinect1, Xtion), choose Kinect1 option. And next you need choose our SLAMRecon or original Kinect Fusion method to do slam and fusion. But whatever method you choose, before run the slam and fusion, you need to do the initialization(click the SystemInit button).

3. If you choose our SLAMRecon method, the system will ask you to choose twice. First, you need choose the FILES_PARAM3.yaml for Files option or KINECT_ONE_PARAM.yaml for Kinect1 option. It is important to note that the ORBvoc.txt need to put the same folder as param files. The system will automatically load this file. And then you need to choose the RGB-D image data folder. At last the system will load camera calibration matrix, opencv distortion parameter and orb vocabulary from this files and data folder. For the original Kinect Fusion method, you only need choose once.

4. After initialization, you can click Start button to start the slam and fusion. You also can stop the process(click Stop button) and restart(click Start button after stop). If you click Reset button, the system will reset, and you should start from the 2 step(new initialization). At last, you can save the fusion result(click SaveMesh button). See Fig.5-2. You can see the running process of the system.



Devices: Kinect1 [✓ Files]
Methods: [✓ SLAMRecon] Kinect Fusion

Fig.5-2. Running process.

6 References

[ORB-SLAM] Mur-Artal R, Montiel J M M, Tardos J D. ORB-SLAM: a versatile and accurate monocular SLAM system[J]. IEEE Transactions on Robotics, 2015, 31(5): 1147-1163.

[InfiniTAM] Kahler, Olaf, et al. "Very high frame rate volumetric integration of depth images on mobile devices." IEEE Transactions on Visualization and Computer Graphics 21.11 (2015): 1241-1250.

[Voxel Hashing] Nießner M, Zollhöfer M, Izadi S, et al. Real-time 3D reconstruction at scale using voxel hashing[J]. ACM Transactions on Graphics (TOG), 2013, 32(6): 169.

[Kinect Fusion] Newcombe R A, Izadi S, Hilliges O, et al. KinectFusion: Real-time dense surface mapping and tracking[C]//Mixed and augmented reality (ISMAR), 2011 10th IEEE international symposium on. IEEE, 2011: 127-136.

[Marching cubes] William E. Lorensen, Harvey E. Cline:
Marching cubes: A high resolution 3D surface construction algorithm. SIGGRAPH 1987: 163-169