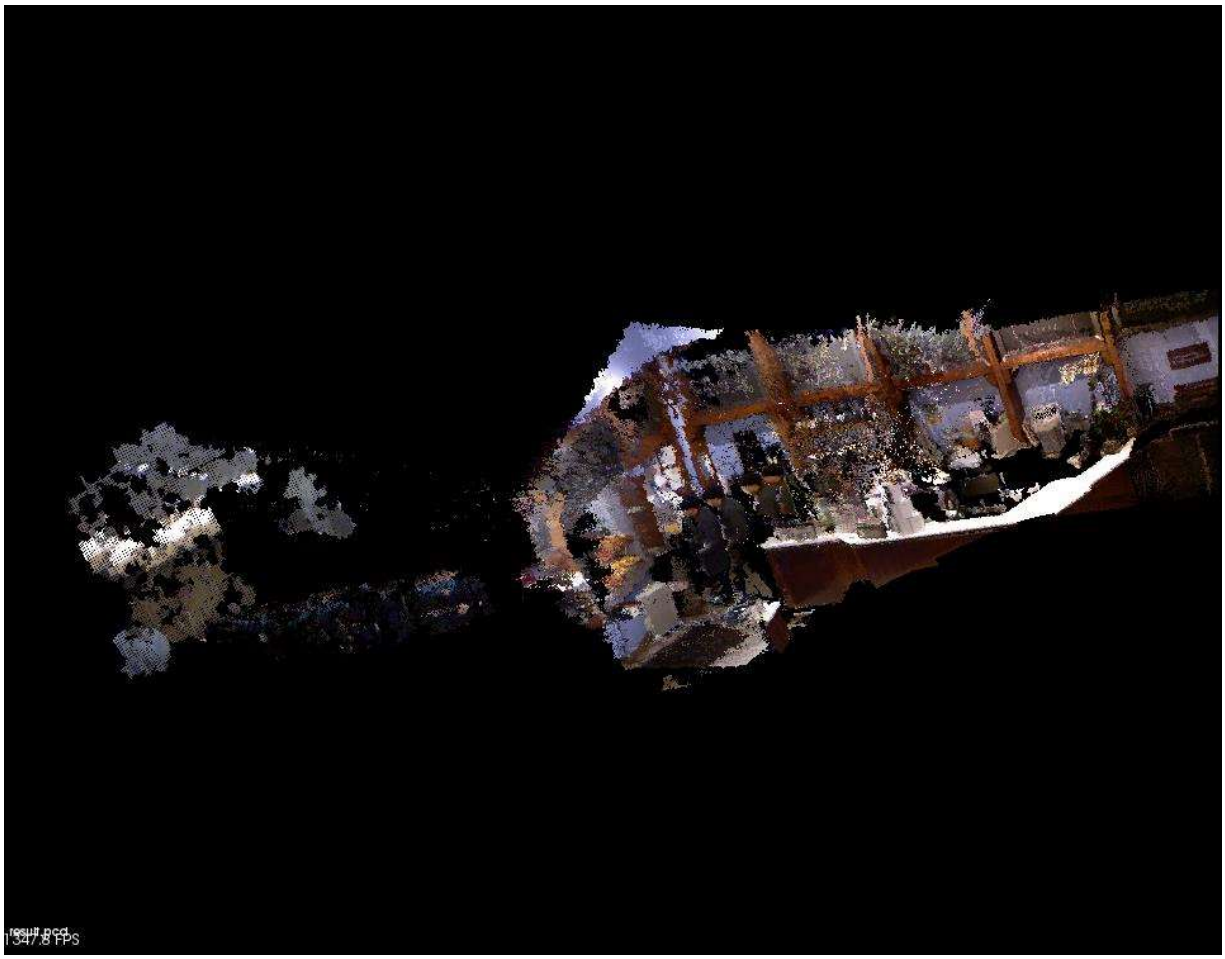
Yue Zhang

RGBD SLAM



•Problem Summary

In this project I propose a mapping system that generates 3D maps using the actual environment and objects as input, which can take advantage of the dense color and depth images provided by RGB-D cameras. By tracking feature points and their relation in space, and inferring the camera position, a 3D map can be generated.

For Augmented Reality Games, the device has to know its 3D position in the world. This project provides a method of calculating this through the spatial relationship between itself and multiple key points. This is an important aspect to implement computer animation, virtual reality and augmented reality.

• Description of work

Extract and match the features in one frame with another,

SLAM is composed of two parts: "Localization" and "Mapping". Now look at the positioning problem. To solve the motion of the robot, we must first solve such a problem: Given two images, how do you know the motion relationship of the image?
This problem can be solved with a feature-based or direct method. Although the direct method has developed a certain degree, the current mainstream method is still based on the feature point. In the latter method, you first need to know the "features" in the image and the one-to-one correspondence of these features.
This problem can be solved with the classic ICP algorithm. The core is singular value decomposition (SVD), Since ICP is not provided in OpenCV, we use PnP to solve in the implementation.
To extract features from an image, the first step is to calculate the "keypoints" and then calculate the "descriptors" for the pixels around those keys.

```
cv::Ptr<cv::FeatureDetector> _detector = ORB::create();
cv::Ptr<cv::DescriptorExtractor> _descriptor = ORB::create();
```

There are mainly 3 kinds of key point, SIFT, SURF, FAST, ORB, in this project I will use ORB key points. We keep the keypoints in a structure called KeyPoint. The KeyPoint structure has a member variable Point2f pt, which refers to the pixel coordinates of this key point. In addition, some key points are parameters such as radius and angle, which will be like a circle in the picture.

Calculate the descriptor on the keypoint. The descriptor is a matrix structure of cv::Mat, each of which represents a feature vector corresponding to the Keypoint. The more similar the descriptors of the two keypoints, the more similar these two key points are. It is through this similarity that we detect motion between images.
The second step is feature matching. Calculate the descriptor on the keypoint. The descriptor is a matrix structure of cv::Mat, each of which represents a feature vector corresponding to the Keypoint. The more similar the descriptors of the two keypoints, the more similar these two key points are. It is through this similarity that we detect motion between images.

Next, we match the above descriptors. In OpenCV, you need to choose a matching algorithm, such as bruteforce.

The matching descriptors seems to be too much, matching many dissimilar things. (Because the two images only rotate horizontally, the horizontal match line is correct, and the others are mismatched). Therefore, we need to filter these matches, for example, remove the distance too much. The criteria for screening are: remove matches that are four times greater than the minimum distance.

The 3rd step is solving PnP. The core of solving PnP is to call the SolvePnPRansac function in OpenCV.

```
zy@zy:~/Projects/RGBD_SLAM/code/bin$ ./detectFeatures
Key points of two images: 500, 500
Find total 500 matches.
min Distance = 3, max Distance = 88
321
inliers: 199
R=[-0.02451700673298071;
 0.03158071949315056;
 0.01527050514702728]
t=[0.02571859231617519;
 0.008729831876806041;
 0.007775584805084141]
```

After the solution is completed, rvec and tvec contain information about displacement and rotation.

Joint point cloud.

The splicing of point clouds is essentially a process of transforming point clouds. This transformation is often described by a transform matrix:

$$T = \begin{bmatrix} R_{3\times3} & t_{3\times1} \\ O_{1\times3} & 1 \end{bmatrix} \in R^{4\times4}$$

The upper left part of the matrix is a 3x3 rotation matrix, which is an orthogonal matrix. The upper right part is a 3 × 1 displacement vector. The lower left is a 3×1 scaling vector, which is usually taken as 0 in SLAM, because things in the environment are unlikely to suddenly become larger and smaller. The bottom right corner is 1. This array can be used to transform points or other things in a homogeneous manner.

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ 1 \end{bmatrix} = T \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ 1 \end{bmatrix}$$

Since the transformation matrix combines rotation and scaling, it is a more economical and practical expression. It is widely used in robotics and many three-dimensional space related sciences. The point cloud transform function is provided in PCL. As long as the transformation matrix is given, the entire point cloud can be moved:

```
pcl::transformPointCloud( input, output, T );
```

In OpenCV, a vector is used to express the rotation. The direction of the vector is the axis of rotation, and the size is the arc of rotation. We first convert the rotation vector into a matrix using Rodrigues transform and then assemble into a transformation matrix. code show as below:

```
cv::Mat R;
cv::Rodrigues( result.rvec, R );
Eigen::Matrix3d r;
cv::cv2eigen(R, r);// convert cv format to eigen format

// combine Rotate matrix with Translate vector to get Transform matrix
Eigen::Isometry3d T = Eigen::Isometry3d::Identity();

Eigen::AngleAxisd angle(r);
cout<<"translation"<<endl;
//Eigen::Translation<double,3> trans(result.tvec.at<double>(0,0), result
T = angle;
T(0,3) = result.tvec.at<double>(0,0);
T(1,3) = result.tvec.at<double>(1,0);
T(2,3) = result.tvec.at<double>(2,0);

// convert to cloude
```

Cope with a frame stream

The data we used are taken from the nyuv2 data set: http://cs.nyu.edu/~silberman/datasets/nyu_depth_v2.html. This is an internationally recognized, well-known data set.
My way to cope with multiple frames, to say it simply, is to match the new data with the previous frame, estimate its motion, and then add up the motion.

```
// read a Frane with particular index
FRAME readFrame( int index, ParameterReader& pd );
```

readFrame is a function to read frame data. Telling it that I want to read which frame , it will find out the data and return a FRAME structure.

After getting the match, we determine if the match was successful and discard the failed data. Why do you do this? Because of the previous algorithm, a result can be made for any two images. For an unrelated image, it is obviously wrong. So remove the situation where the match fails. We have adopted 3 methods to verify it the match is failed:

1. Remove the frame with too few good matches. The minimum good match is defined as:

```
min_good_match=10
```

2. Remove the lessin frame in solvePnPRASNAC, the same reason is defined as:

```
min_inliers=5
```

3. Remove the case where the resulting transformation matrix is too large. Because the motion is assumed to be coherent, there is no such big distance between the two frames:

```
max_norm=0.3
```

We calculated a value that measures the size of the motion:

$$\|\Delta t\|+\min(2\pi-\|r\|,\|r\|)$$

It can be seen as the sum of the norms of displacement and rotation. When this number is greater than the threshold max_norm, we think the match is wrong.