Assignment 2: Voxel-Based 3D Reconstruction Report

Satwiko Wirawan Indrawanto - 6201539
Basar Oguz - 6084990

A) Introduction

This document reports the implementation and methods of a voxel-based 3D reconstructor. This application operates on footage of the same scene from 4 different, calibrated camera viewpoints to reconstruct the foreground of the scene as a surface mesh in a virtual voxel-based, half-cube 3D scene.

B) Camera Calibration

We have begun with calculating the intrinsics of the cameras using the *intrinsics.avi* file that was provided for each of the 4 cameras. For this, the CalibrateCamera program we made for the first Assignment was used. The only minor modification was changing the input from webcam to *intrinsics.avi*. The calibration was made with 30 snapshots from different segments of the video file. The intrinsics calibration yielded us focal lengths (fx,fy), optical centers (cx,cy), as well as 5 distortion coefficients which were all stored in the *intrinsics.xml* of the respective *data/cam* folder with the format provided.

To find the world coordinates of the cameras, or calculating the extrinsics, the *checkerboard.avi* was ran through the given Camera::detExtrinsics(...) function with carefully marking checkerboard corners in the same order, starting from the same corner for every camera. The marked corners of the 6-by-8 checkerboard, in total 48 corner coordinates were saved in *boardcorners.xml*. These two calibration steps allows us to locate the cameras with respect to the scene and also each other. The combination of the intrinsics and extrinsics (rotation and translation vectors) were saved in the *config.xml* to project voxels from 3D space to the 2D screen of each camera.

C) Background Subtraction

In order to reduce the noise that a single background image would produce on the final black and white mask, we have used pixel averaging to create a more suitable background image. A final *background.png* was created by averaging the same pixel position's RGB values over all the frames in the *background.avi*. Each frame is stacked into layers using Photoshop and processed into one image using the mean function. This way, the undesirable effects of minor changes in lighting or other uncontrollable conditions have been smoothed out.

Using the averaged *background.png*, we have converted the image to HSV color model representation. We have experimented with different threshold parameters for Hue, Saturation and Value to extract a binary pixel mask that represents the foreground. We have found that for this particular scene, the following thresholds work best, therefore we initialized our trackbars to: [Hue: 10, Saturation: 20, Value: 50].

We have found that the HSV color model, even with the best threshold parameter inevitably results in some rogue white pixels on the mask. Also, some pixels in the foreground that are false negatives were causing planar cuts in the final reconstruction. We have improved the quality of the foreground mask with an “erode-dilate-erode” routine, applied to the masks after thresholding. Erosion and dilation are both convolutional operations, so the kernel sizes can vary. We have experimented with different sized kernels by adding trackbars to the UI that change the kernel window sizes. The most compelling results were had by using a 2x2 erosion kernel and a 4x4 dilation kernel, therefore the program initializes with these values.

D) 3D Reconstruction and Surface Mesh using PolyVox

The voxel initialization process was done as provided in the example code. However, OpenMP was enabled to increase voxel construction significantly (4-5x). We also implemented PolyVox to draw the surface mesh instead to just just voxels. This can be toggled by using the “D” key.

E) Surface Mesh (PolyVox)

PolyVox’s library had to be compiled and built to be used, so we compiled it using CMake for a 64 bit version. To use PolyVox to draw the mesh, first we need to create a volume to store the 3D grid of voxels. This is stored inside PolyVox::SimpleVolume(..) by looping through all the visible voxels. We used the PolyVox::MarchingCubesSurfaceExtractor function to convert volume data to triangle mesh. This function will make use of a smooth density field and consider a voxel to be solid if it is above a certain threshold.

To draw the mesh on GLUT,, instead of using points to draw the voxels, we need need to use triangles to construct the mesh. This triangle primitive can be initiated using glBegin(GL_TRIANGLES). The mesh color is then calculated based on the vertices normal.

A demo video can be seen on https://www.youtube.com/watch?v=kJSH8JsiGos.

References

http://www.volumesoffun.com/polyvox/documentation/0.2.1/manual/install.html

http://www.volumesoffun.com/polyvox/documentation/0.2.0/manual/tutorial1.html