CS 378 - Final Project Report - Group 6

3DScanner

Sean Cascketta, Matthew Diamond, Miguel del Valle, Nishant Gupta

Overview

The original goal of our project was to use a Nintendo 3DS to extract 3D models of scenes from videos taken with the two cameras on a Nintendo 3DS. Our initial objectives were to:

- Understand the AVI file format produced by a Nintendo 3DS
- Reconstruct a 3D scene using a single stereo image pair produced by a 3DS
- Track the 3DS camera position
- Properly rectify multiple 3D scenes together

Within the first week, we understood the format for a 3D video from a Nintendo 3DS and were able to write a script to extract two video streams from a 3D video file. However, the quality of the video streams were lacking in detail and noisy which made rectifying the images difficult and produced disparity maps of poor quality. Due to the setbacks encountered while trying to use the video from the 3DS, we decided to use a homemade stereo camera to capture video instead.

Given this, our final objectives were to:

- Properly calibrate our homemade stereo camera
- Create a high-quality point cloud using rectified images from a proper calibration
- Composite multiple point clouds together to create a single scene

We were able to complete the first two objectives to a satisfying degree within our time constraints. Using a printed chessboard pattern, we were able to calibrate our homemade stereo camera and properly rectify images from a video. With properly rectified images, we could produce high-quality disparity maps (with far less noise) and create reasonably accurate point clouds, far better than the results from the second project on stereo.

We were unable to approach the problem of compositing multiple point clouds together as camera calibration turned out to be more time consuming than expected. This was due to poor OpenCV documentation on camera calibration and confusing errors.

Background

Our project utilizes the open-source OpenCV and Numpy library and documentation extensively. Numpy is used for general-purpose array computation. The most notable functionality we utilized from OpenCV is calibrating our stereo camera and computing stereo correspondence from a rectified stereo pair of images.

Our camera calibration module uses OpenCV to find the intrinsic and extrinsic parameters of the left and right camera. The algorithm used by OpenCV is based on the methods outlined in the paper *A Flexible New Technique for Camera Calibration* (Zhengyou Zhang, 2000). In short, this algorithm computes the initial intrinsic camera parameters, estimates the initial camera pose based on correspondences from 3D and 2D points, and then uses the Levenberg-Marquardt algorithm to minimize the error from observed feature points and projected object points.

The parameters for each camera are later used to calibrate the camera as a stereo pair. Again, we use OpenCV (the StereoSGBM class) to compute stereo correspondence for a rectified stereo pair of images used to produce a disparity map.

A fundamental problem in computer vision is computing the 3D shape of an unknown, arbitrarily shaped scene from multiple photographs taken at known but arbitrarily distributed viewpoints.

In order for us to compose multiple point clouds together to create a scene, we explored the idea of Space Carving, as is mentioned in the paper *A Theory of Shape by Space Carving* (Kutulakus, Seitz).

In short, the basic idea of Space Carving is:

- Divide the “looked-at” space conceptually into evenly divided cubes of space, called voxels (AKA “volume elements”). We assume that some of them are space, and some are solid -- but we don’t know yet which ones.

- Initially hypothesize that every voxel is opaque.
- Carve away the voxels whose colors don't match.

Unfortunately, due to time constraints, we were not able to implement Space Carving, as the other objectives took longer than expected.

Methods

Hardware

Originally, our primary piece of hardware was a stock Nintendo 3DS, which have two cameras on the back which can be used for taking stereo images and video.

Our homemade camera consists of two inexpensive, consumer-grade Logitech cameras taped (somewhat) securely to a piece of wood. Occasionally, we had to adjust the orientation and re-calibrate the cameras.

Our code runs fine on several consumer laptops, but the demo in our presentation ran on an Lenovo laptop with an Intel i5 processor with 8 GB of RAM. Any computer with at least 4 GB of RAM and modern processor should be capable of running our code.

Software

At a high level, the majority of our code is structured into two Python modules, **calibrator** and **stereo_rig_model**, described in detail below. The **stereo_rig** module is similar to stereo_rig_model but is used to tune the stereo block matching parameters. The **record_calibration** module is used to record videos of our chessboard calibration rig. Finding the corners of the chessboard every time we wanted to re-calibrate the cameras was tedious, so we have a video in our *test_data/videos/calibrator* directory to re-calibrate the cameras. All of the Python modules make extensive use of OpenCV and Numpy.

The **test_calibrator** and **test_stereo_rig_model** modules have basic tests of our code's functionality, which can be run with the **run_tests.sh** shell script.

We also wrote a (now vestigial) shell script **process_3ds_video.sh** which extracts the left and right video streams from a 3D video file from a Nintendo 3DS.

The **calibrator** module computes the undistortion and rectification transformations to rectify images from our stereo camera.

The calibration process starts with finding the intrinsic and extrinsic camera parameters for the left and right camera. Before we can calibrate the stereo camera, we need to estimate the intrinsic parameters of each individual camera to a high degree of accuracy.

We accomplished this by taking a recorded video of a chessboard (our calibration pattern), finding the corners of the chessboard, and then using the identified corners (our object points) with the points projected on the image to get a camera matrix and distortion coefficients.

With the camera matrix and distortion coefficients for each individual camera, we can find a full calibration of the two cameras as a stereo pair and save the calibration data as two files. At this point, we can load the calibrations at any time later on and use them to rectify images from our stereo camera.

The **stereo_rig_model** module computes the average disparity between two stereo cameras, then creates a point cloud. Using OpenCV's StereoSGBM class, we compute the stereo correspondence for a rectified stereo pair of images in order to produce a disparity map.

With the disparity map, we compute the 3D coordinates for each pixel, reproject the image points to 3D space using OpenCV's reprojectImageTo3D class, compute the colors, and export the point cloud into a PLY file in order to view it in Meshlab.

Results

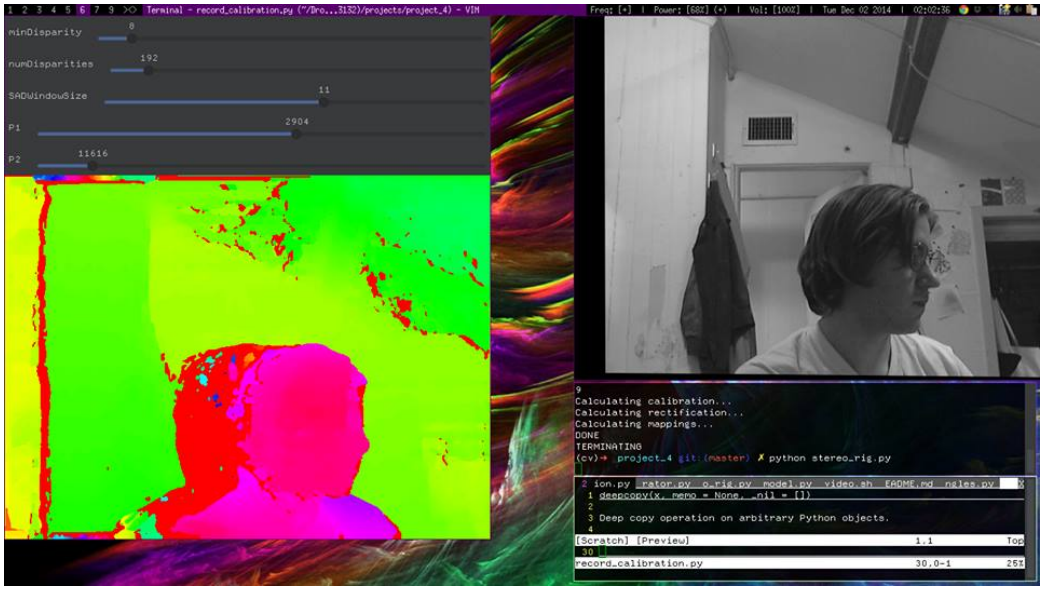
We were successful in calibrating our cameras to properly rectify images, our latest reference calibration data is located in the *test_data* directory:

- map_left_1.npy
- map_left_2.npy
- map_right_1.npy
- map_right_2.npy

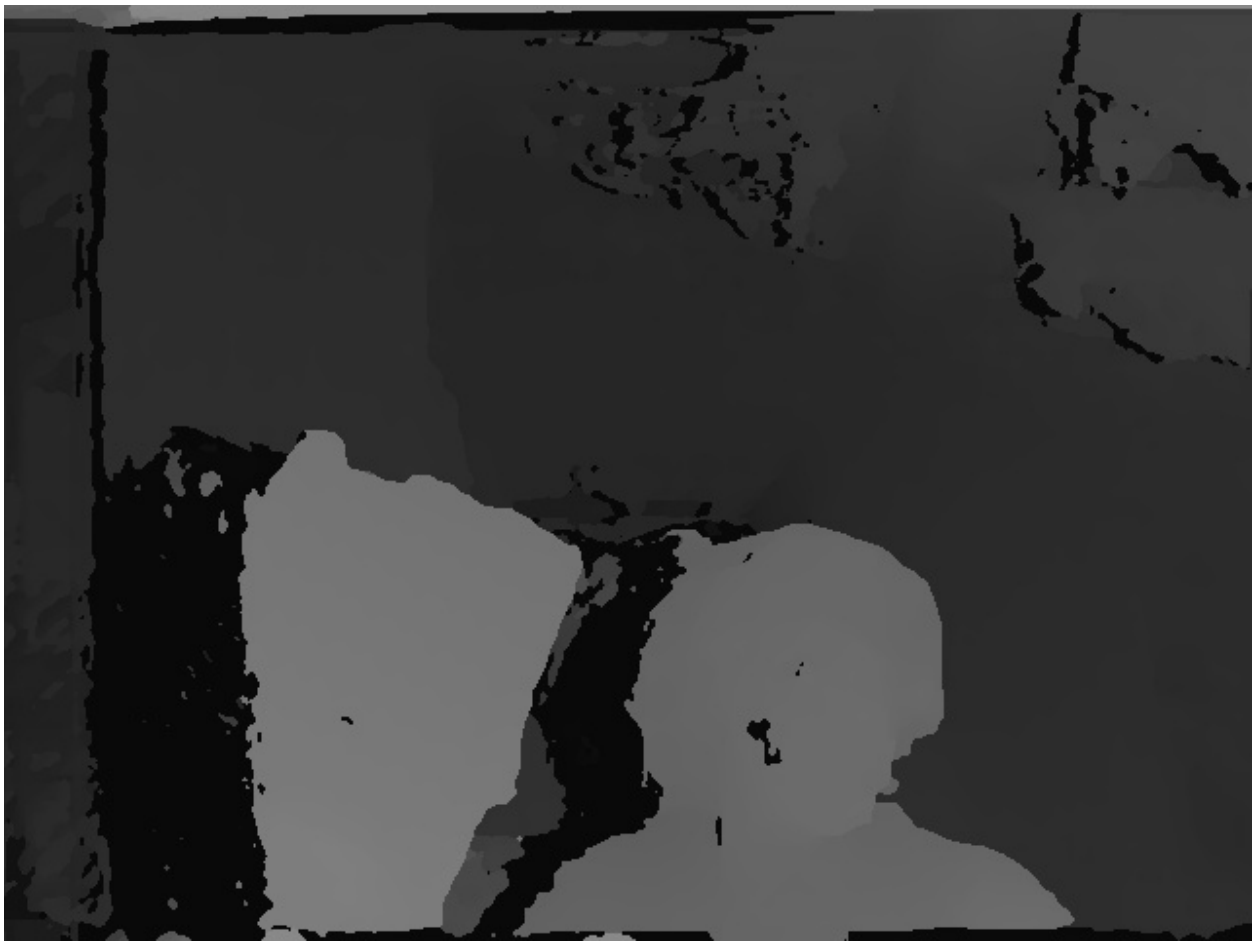
These files are loaded with Numpy and used to rectify the images in **stereo_rig_model**. The intrinsic camera parameters can be found in *test_data/left_k.txt* and *test_data/right_k.txt*.

The visual final results of our code consist of disparity maps and point clouds. The point clouds can be found in the models directory, and are viewable within Meshlab.

Disparity map on the left, original scene on the right, stereo matching tuner GUI in the top left.



Another disparity map.



Point cloud in Meshlab.