Simon J. Bloch and Gibson Cook 5/15/2015
Computer Vision - Swarthmore College - Spring 2015

Computer Vision Final Project: 3D Scanner

Introduction:

For our final project in Computer Vision we wanted to tackle something that applied interesting elements of the course in a substantially challenging way. After some discussion with our professor, we decided to deepen our understanding of 3D geometry, and put together a working 3D scanner. Much of our work follows that of Greg Turk and Mark Levoy in their famous Stanford bunny paper[1]. For calibration of the projector-camera setup we roughly follow the method proposed by Gabriel Falcao, Natalia Hurtos, and Joan Massich[2]. The project ended up being more involved than we had hoped, but we successfully obtained the depth map we wanted.

An overview of our approach is as follows: we project a vertical beam of light onto an object, move the beam across the screen, use stereo reconstruction to obtain a depth map, rotate the turntable on which the object rests, and scan it again. Merging the scans from each rotation gives a full 3D point cloud that can be viewed in a suitable application. We did not effectively implement the rotation component of the scan, but we were able to generate an accurate point cloud from one view. Since we see this project as a strong proof of concept for future work, we have been fairly detailed in the documentation of the scanning technique and various programs. This report contains an outline of the projector-camera setup we used, how we calibrated it, a description of the scanning process, analysis of our results, and discussion of future work.

Setup:

We mounted a Casio Green Slim projector and a Logitech C905 webcam to a small table, fixing the webcam slightly above and beside the projector with a metal rod since offset is needed for stereo reconstruction. At a reasonable distance from the projector table we placed a metal turntable. The turntable rotates smoothly around a vertical axis. We carefully attached a paper wheel with degree markings on top of the turntable to measure rotation. Initially we had hoped to automate this process for convenience and to eliminate human measuring error, but we could not access a stepper motor that exerted sufficient torque.

All code for this project is written in Python using Numpy and the OpenCV library. We also use the preview application that comes standard on Macs to view and manipulate images.

---

[1] *Zippered Polygon Meshes from Range Images*, by Greg Turk and Mark Levoy

[2] *Plane-based calibration of a projector-camera system*, Gabriel Falcao, Natalia Hurtos, Joan Massich

Figure 1. Our Hardware Setup

Calibration:

The work described in the following section is from our program calibrate.py. This program finds the intrinsic parameters for both the camera and projector as well as the rotation and translation between them. First, we take several images of a poster board that has a printed chessboard taped to its upper right corner. We hold this poster board in front of the projector and project the very same chessboard pattern onto it (it was necessary to desaturate the projected image to avoid overexposure). We take twenty images, slightly changing the orientation of the poster board for each. The images are copied into two sets and edited using the preview application on the Mac. For one set we cover the printed chessboard; for the other set we cover the projected chessboard. A second script is run to look through each image and notify us of any images for which the chessboard corners cannot be found. After removing these images from our data set, we run our calibration program.

First we find the intrinsic parameter matrix of the camera, *K_cam*, by establishing three frames: an object frame, a projector frame, and a camera frame. The object frame lies on the poster board and has its origin at the top left square of the printed chessboard. The projector frame has its origin at the optical center of the projector, and the camera frame has its origin at the optical center of the camera. We use known coordinates and find the chessboard corners in our printed images, then obtain the intrinsic parameter matrix of the camera. We also get the rotation and translation vectors from each calibration image. With these we can find a homography that maps from the camera frame to the object frame.

The homography is now applied to the images showing only the projected chessboard. We find the projected corners in the object frame, along with the projected corners in the camera frame and the projected corners in the projector frame. We can now calibrate the projector itself. After we obtain *K_proj* and relevant distortion coefficients, we use the projected points in all three spaces to run a stereo calibration and obtain the final rotation and translation (R and t) from the normalized image plane to the projector plane. *K_cam*, *K_proj*, R, and t will later be used to determine depth information, so these are the things we pass into our main program.

Master program:

Inside our main program--called scan3D.py--we call calibrate.py to get calibration data. We perform our reconstruction using three major nested loops. The innermost loop looks at the pixel locations of points along a projected beam and recovers depth information. The loop outside of that calls the scan program and iterates over every projected line. It adds each new 3D point to the point cloud. The final loop outside of that calls the scan program successively, accounting for a 20 degree rotation each time. Thus, the final method should produce a single, high-resolution single point cloud of merged 3D point scans. The finished point cloud is saved as a .npz file and can be viewed using PointCloudApp.py.

When the scanner program is called, it moves a bright line in horizontal steps of 10 pixels and takes a photo at each specific X-value. We take the absolute difference from a background image, and threshold around greener pixel values to separate out the areas where the line hits the object. After binarizing the output image, we have a collection of pixel addresses, each corresponding to the specific X-value at which the line was projected.

We recover depth data from an object by using our stereo calibration to interpret the position of the beam, which appears shifted by the 3D contours of the object, as depth information. The pixel coordinate where the line hits the object can be expressed as a ray in the camera frame. The projected green line can be expressed as a plane. We define that plane as a function of the optical center of the projector and of the pixel address of the line we project. We find the plane's normal vector and recover depth information, by which we scale our object points in camera space.

Results:

Our results for this project included many successes, but we did not reach our final goal. A small, but confounding calibration bug took up too much time for us to actually start rotating and merging multiple point clouds. The issue involved a radial distortion error that made the overall depth data unreliable and added abnormal curvature to the point cloud. After rigorously debugging and reviewing theory, we overcame the bug, which was simply due to mixing-up the focal lengths, *fx* and *fy*.

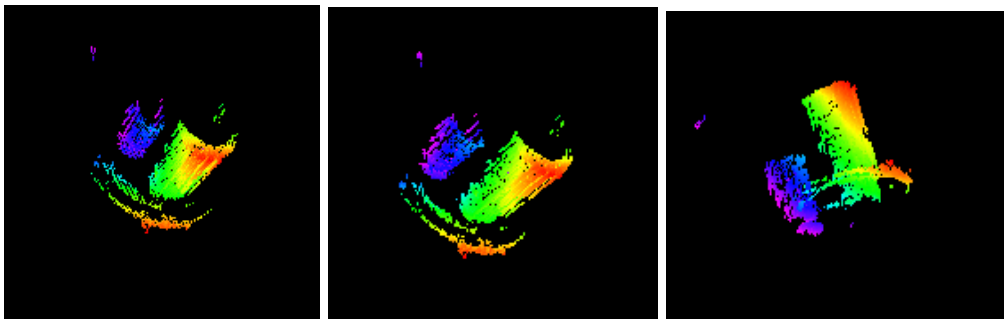

Figure 2. Successful Point Cloud

The calibration setup proved to be the most difficult part of the project; we initially had no intuition for what the intrinsics and extrinsics should look like, and it was difficult to tell if errors in our point clouds were due to calibration or issues with our depth recovery methods. We developed a few workarounds to streamline the debugging process. First, we wrote two scripts--getpix.py and testpix.py--that obtain calibration images and test their quality. We used testbox.py to ensure our object was inframe and that reflective surfaces were masked out.

To evaluate the soundness of our calibration methods, we first printed and examined the point-sets in every different frame and format to make sure the input data was sound. We looked at the outputs of our various calibration functions to make sure that the intrinsic matrices, rotation and translation vectors, and rms error values were accurate and within reason. We also had to rigorously test our scan program to make sure we were only outputting points where the line hit the object. We experimented with several different filtration methods, visualizing our binarized outputs until we settled on the one found in the "clean" function of scanner.py.

There were several other challenges in this project. Equipment, for instance, took several days to assemble. We also struggled with working in sunlight. The spaces we used let in too much natural lighting for the green beam to be distinguished if we worked during the day. Rather than work around this, we would like to use our scanner irrespective of lighting conditions. We would also like to fix the relative locations of the projector and camera so we only have to take calibration images once. We think these two problems could be solved in one fell swoop by scanning with the Microsoft Kinect. It uses an infrared projector-camera setup that works in any lighting, and its components are fixed in relation to one another.

Future Work:

Time ended up getting the best of us. Though we ultimately obtained reasonable, undistorted point clouds, we did not have the time to start merging multiple rotations--let alone fit a mesh to the point cloud.

There is currently too little precision in our methods; we don't scan across every pixel and often see outliers and artifacts in our data. To reach our long-term goal of generating a fully printable 3D model, we would have to focus on first obtaining a dense, precisely merged point cloud representing the object's entire surface. Only then could a stitching algorithm be run to create the polygonal mesh.

We could benefit from further vectorizing our many nested loops and improving the computational efficiency of our program, possibly converting it to C++. Other future improvements include adding an adequate working motor and integrating full rotational scans at multiple poses, ie. turning an object on its sides, top, bottom, etc., to help see inside objects and around obstacles.

Implementing our methods with Kinect would present an interesting opportunity to expand open source 3D scanning. Software like ours exists for Kinect, but given the difficulty of this type of work, there may be opportunities to innovate.

Conclusion and Acknowledgements:

We see this project as having been a worthwhile learning experience. We faced certain practical realities, and we leave with a greater understanding of the time management strategies necessary to

accomplish hardware-extensive and ambitious tasks. We've never been better equipped to apply 3D geometry in computer vision.

We offer immense thanks to our professor, Matt Zucker, without whom we would have been totally lost. He implemented the calibration method presented by Falcao, Hurtos, and Massich; provided resources for the 3D geometry and point cloud generation; and was a constant resource for debugging. Further thanks to Ed Jaoudi, James Johnson, Erik Cheever, Swarthmore Engineering, Media Services, and Cosmo Alto.