3D Reconstruction

R&D Project Report

Submitted by

Sriram Kashyap M S
Roll No: 08305028

Under the guidance of

Prof. Sharat Chandran

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai

2009

Abstract

Traditionally, the input to the graphics system is a scene consisting of geometric primitives, their material properties and the light sources. These scenes are usually generated using a 3D modeling package. Often, it is desirable to be able to directly import an existing real object into the virtual world. One method is to use expensive hardware such as a 3D scanner to extract 3D information. This project explores the alternate solution to this problem, namely 3D reconstruction using images captured by cameras.

The goal of this project has been to develop a system which can capture the 3D information about a given object, using image based methods. This involves capturing images from multiple viewpoints, calibrating the cameras used for capturing the scene, segmenting the background from the foreground object in the images captured by the camera, and providing these calibrated images as input to a visual hull construction system to extract 3D information.

Acknowledgement

I thank my guide Prof. Sharat Chandran for his invaluable support and guidance. I thank Biswarup Choudhary for his insightful suggestions and continued support. I also thank the VIGIL community for their support, and the SPANN Lab for providing the turntable and motor controller.

Table of Contents

1 Introduction

This project involves capturing images of an object from multiple view points and building a 3D model (the visual hull of the input images) out of it. This model will be further used to perform image based rendering of the object from novel viewpoints.

The project augments the initial phase of the existing Image Based Animation framework. The currently available system has been tested on synthetic inputs from rendered 3D scenes. The main goal of the project will be to build an image acquisition system which will simplify the process of capturing images of real-world objects and extracting 3D object information from these images.

The phases involved are as follows:

1. Verification of existing code base: Test the existing visual hull creation system using available datasets
2. Create a image capture setup: This setup will provide object images and corresponding calibration images
3. Image Segmentation: The visual hull construction algorithm requires each image pixel to be binary-classified as background/foreground
4. Camera Calibration: Given a set of images with a calibration pattern in them, compute the camera matrices corresponding to those views
5. System Integration: Modify the existing system so that it can seamlessly accept data coming in from the above steps

2 Verification

The first phase involved verifying that the existing system works for other datasets (and not just for those created by POV-Ray). To do this, I familiarized myself with the existing codebase and user interface. I imported datasets from a previous 3D reconstruction project and tested if the resulting 3D models were convincing. Certain assumptions of the original codebase were no longer valid and were rectified. The most major of these was the assumption that images are taken by moving the camera on the surface of a sphere of fixed radius around the object. This was modified to allow arbitrary distance between object and camera.

The knight dataset is a computer generated set of 8 images of a chess knight. The temple dataset is a set of 16 images from the TempleRing dataset of the University of Middlebury. Both datasets contain images at a resolution of 640x480.

The results from this reconstruction test are shown below. The reconstructed images are simply point models and do not contain texture information. The colors seen in the reconstructions here and through the rest of the report are due to the color coded visualisation of 3d data. The R,G,B components of the 3d points vary with their X,Y,Z coordinates, thereby providing a sensation of depth.

3 Capture setup

The capture setup is an acquisition system for taking images of an object from various viewpoints. It was decided that we would require a mechanized turn table to make the object rotate at regular intervals of time. This turntable has been procured from the SPANN lab of the Electrical Dept.

A camera is mounted on a tripod at a fixed height and distance from the table. The table and the background are covered with a uniform colored diffuse surface (sheets of paper). Ambient lighting is

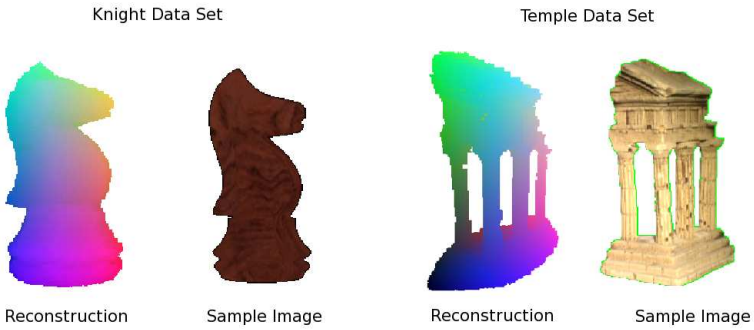

Figure 1: Image acquisition setup

achieved by using flood lights reflected off thermocol sheets from behind the camera. This ensures that most of the shadows are behind the object and do not show up in the captured images.

A complication that occurs due to the light source behind the camera is that the camera and tripod now cast diffused shadows on the table surface. To prevent this from causing any problems, a set of basis images are taken without the object on the table, so we can assume that the camera shadow is now part of the background.

The desirable features of a camera are:

1. Manual Exposure: Auto exposure causes unexpected intensity variations between views
2. Unlimited burst mode: The ability to take images at regular intervals of time, continuously
3. Resolution: A resolution of greater than 2 mega pixels is desirable, but in practice, even 640x480 is seen to provide acceptable results

The camera used is a Logitech QuickCam Pro 9000 webcam. The disadvantage of using a webcam is that image quality is much lower (but it is still acceptable for our purposes). The major advantage is that the camera can be fired by wire (over USB). This means that a script can be written to control the camera, thereby obtaining greater precision in timing.

To fire the camera at regular intervals of time and save the captured images, OpenCV's camera toolkit has been used. OpenCV 1.1 supports setting of camera capture resolution, due to which it was possible to capture images at low resolutions (640x480) as well as high resolution (960x720).

The turntable is controlled by a stepper motor controller and rotates based on an internal clock which can be manually configured using a dial on the controller. A marker has been placed on the turntable axis to specify a standard initial configuration before the start of any capture process.

A note on camera exposure and white balance: Although the webcam used for the experiments supports manual exposure and white balance, the tool used for image capture may not support these features. While Logitech's own tool supports them, a third party tool like OpenCV's image acquisition system does not support these custom settings. In such cases, drastic variations in color can be noticed as the contents of the scene change. This was overcome to some extent by introducing a bright white (or colored) strip in a corner of the image, as part of the static background. Adding such a feature to the background ensures that even if no object is placed in the acquisition setup and only the background is visible, it will not desaturate the image.

Figure 2: Image acquisition setup

4 Image Segmentation

Segmentation involves separating the background image from the foreground object. The existing reconstruction module expects 32 bit png input, with an alpha channel per pixel, which denotes whether the pixel is background (alpha=0) or not. To do this, I capture a background image which do not contain the object. This background image is compared pixel by pixel with the images containing the object. Various criteria that can be used to label a pixel as background have been explored. The simplest form of thresholding is RGB thresholding where RGB components of source and background images are compared and a threshold is applied. This works well except for the shadow regions near the base of the object. Note that although there are no hard shadows cast by the light source, shadows appear due to lesser ambient light at contact points (where the object meets the table). To fix this, an additional threshold function based on HSV color space was explored. This idea has been discarded because jpeg compression tends to make the Hue and Saturation very noisy. Instead, it has been found that thresholding on Cb-Cr components of the image can remove some of the shadow regions.

Note that in the images below, this method is not able to fully remove shadow artifacts near the base of the object. This is not expected to cause problems in the final reconstruction due to the nature of the volume intersection operation performed during Visual Hull construction from Silhouettes.If the shadow artifacts are sufficiently noisy, in the final reconstruction, their effect will be greatly reduced. Testing with real data has shown that in most cases, a few small patches of spurious 3d data crop up in the final reconstruction if the segmentation incorrectly marks a few regions of background as foreground. This is more acceptable than the case where foreground pixels are marked as background, and the entire object volume gets chopped off at that point. Therefore, the basic idea followed for segmentation has been to optimistically label pixels as foreground.

The real data tests also showed that varying exposure levels and lighting conditions can drastically change how the scene looks in different views. This means that a single threshold value is not sufficient for segmentation. Therefore, we need to vary the threshold adaptively based on the input image. To do this, the expected ratio of number of foreground pixels to background pixels is sent as input to the application. This remains relatively constant throughout the process and can be obtained by testing with

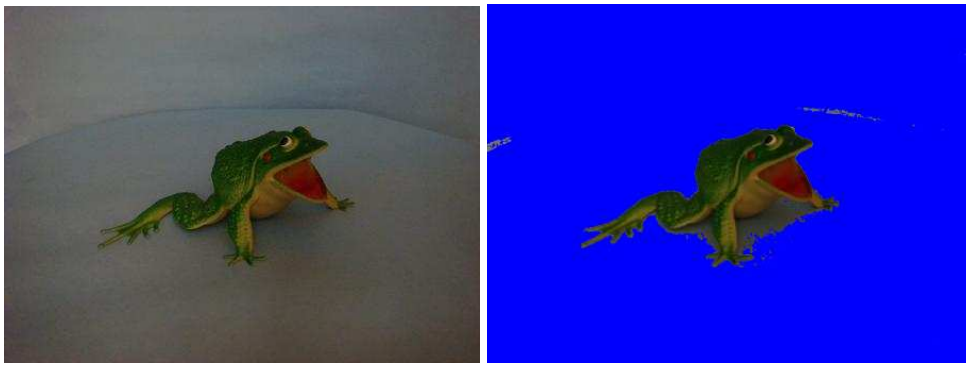

Figure 3: Original frog image and RGB thresholded image

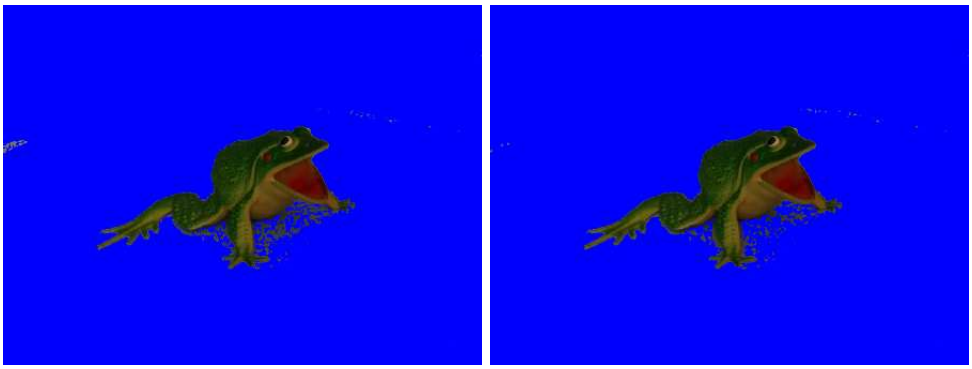

Figure 4: HSV Thresholding and Cb-Cr thresholding

a sample input image. The algorithm starts off with an initial threshold and iteratively segments the image, while increasing the threshold in each iteration, until it obtains the expected ratio of foreground to background pixels.

5 Camera calibration

5.1 Point correspondences

The first step in camera calibration is to find a set of point correspondences (between image and object spaces). This requires us to capture perspective distorted pictures of a known image and process it to find features on it that correspond to the actual pattern. This provides us with point correspondences. It is necessary to find a pattern that works well in practice. The typical calibration pattern is an NxN chessboard pattern. This will not work for our case because there are situations where the entire pattern will be inverted. Distinguishing between the pattern and its 180 degree rotated version is hard while using the chessboard pattern. Therefore, I made a custom pattern with 8x8 black squares, and 2 red squares (corner markers) for alignment (for example, to find our bearings when the entire pattern is rotated by 90 degrees). The idea was to get the centers of these 64 black squares and use them for calibration.

Figure 5: Non adaptive and adaptive thresholding for segmentation

To locate the black squares, I used a technique where the axis (the line joining the centers of the red boxes) would be swept perpendicular to itself, across the board. Each time the axis is shifted by a small amount, we check how many new boxes have been crossed by the line. If 8 new boxes have been crossed, we mark that as a column.

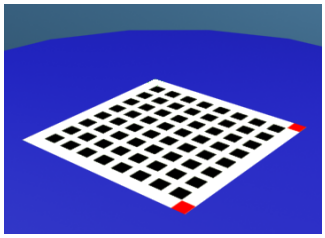

Figure 6: Calibration pattern

The problem with this technique, as was seen in later experiments, is that it does not take into consideration the fact that the entire board has been distorted by a perspective projection. This means that, after a while, the sweeping axis would gather eight boxes, that do not actually belong to the same column. So we may have the first 7 boxes from the first column, and the first box of the next column, while totally missing the last box of the current column. The problem caused by perspective distortion is that sweeping the axis along a vector perpendicular to it in image space is wrong. The vector that is perpendicular to it in object space, has been prespective distorted and in image space, it is no longer perpendicular to the axis.

To fix this problem, two more corner markers were added (in green color). This ensures that we can locate and uniquely label all four corners of the board. It also means that we can now take into account the perspective distortion. The sweeping axis is now no longer simply translated. Instead, consider two primary axes, one formed by the red boxes, and another formed by the green boxes. The sweeping line is obtained by linearly interpolating between these two axes.

The boxes in the image are identified by thresholding the image using the color of the box we are looking for and identifying boxlike contours of the expected size. To check if a contour is boxlike, we look at the ratio of its enclosed area to the area of its bounding box. More robust tests involve simplifying the contour and checking how many edges the contour has. Another useful test is to check how close these contours are to the edge of the screen, thereby eliminating outliers.

To identify the various features (red, green, black boxes), I generalised the algorithm so that it would work with any color and any number of boxes. Thresholding for the boxes is based on the shape of the histogram. Boxes of a particular color cause peaks in the histogram. This means that thresholding has to

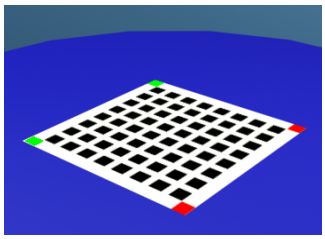

Figure 7: Modified calibration pattern

be done at the valley just after the peak, so as to include that particular color of box in the thresholded image. By evaluating the gradient of the histogram at each point, the algorithm directly jumps from one valley to another thereby performing faster and more robust thresholding. This process can be improved by smoothing out the histogram function before performing thresholding operations. Note that while dealing with black and white images, this technique is accurate, but in the case of grayscale images, it is an approximation.

Note that the feature that is recorded from these patterns is the center of each black box. This is different from most other calibration approaches where corners of the boxes are sought. Recording for the center of each box is more stable and hence was chosen over box corners.

5.2 Calibration

The previous step provides us with a set of point correspondences. To solve this system of equations and get the projection matrix, the first method I used was the least squares technique, which would fit a projection matrix to the observed data in a least squares sense. It involves calculating the pseudo inverse of a matrix X as $[(X'X)^{-1}X']$. I found that this solution does not work for the current problem. The least squares solver works when our input points lie on two or more different planes. Here, all the inputs are on a single plane (z=constant). This caused the system to blow up and give either zeros or a very large numbers in the matrix each time. I found that I could get reasonable re-projection accuracy if i randomly perturbed the z values by small amounts.

The next method I tried was the pseudo inverse using SVD. It fixed the numerical instability issues that came with constant z values in the input data. It turns out that the SVD method is more numerically stable than the regular pseudoinverse techique. To make things better, openCV has already implemented a SVD based solver for a system of linear equations (cvSolve). This not only ensured correct results, but vastly simplified the code.

It turns out that these methods cannot be used to calibrate camera matrices in the current scenario (planar points). The key idea here is that not all 3x4 matrices are calibration matrices. The least squares method tries to fit a 3x4 matrix to a set of observed values. It does not take into consideration the additional constraints that an actual camera imposes on the system. In this case, since all the observed points lie on a single plane, z=0, the components of the projection matrix corresponding to the 'z' coordinate were all near zero. Therefore, the Z coordinate was always being ignored, and all points in the scene would be projected as if their Z component was 0.

The third method used to calibrate cameras is the OpenCV camera calibration function (cvCalibrateCamera). It turns out that this method works well in practice, and can calibrate an entire array of cameras at once. This means that if I have 100 images taken using the same camera from different angles, cvCalibrateCamera can extract the intrinsic and extrinsic parameters of the camera all at once. It takes about a minute now to calibrate all the cameras, once the 3D and corresponding 2D points for each scene are fed into it.

Furthermore, to verify if the matrices that are obtained from cvCalibrateCamera truly what is expected, I wrote a simple voxel renderer which uses a given 3x4 matrix to render the scene. The disadvantage of using a 3x4 matrix is that we dont get the relative z values in image space. ie: there is no easy way of implementing a z-buffer. But the application is sufficient to visualize what a given camera matrix is actually doing.

As a final step, I also had to modify a part of the original code that took camera coordinates, and allow it to take these 3x4 camera matrices that have been computed. One requirement of the system that is not satisfied by just projection matrices is that we need relative positions of the cameras in world space. The Image based animation system renders scenes by performing a near neighbour search to find cameras approximately corresponding to a given novel view, and obtaining texturing information for the novel view, from these nearby views. To allow this query, we have to pass 3 additional numbers corresponding to the position of each camera in space. This can be approximated by flipping its view vector and scaling it to the required distance at which new views will be generated.

6 Testing

6.1 Toolchain

I have written a set of shell scripts that will automate most of the steps involved in the system. They take as input a set of calibration images, an equal number of object images (all named in alphabetical sequence), a color config file that specifies the colors of the boxes in the calibration pattern, a background image, and a file specifying the dimensions of the visual hull to be constructed. A single script invocation will perform the calibration and segmentation process, select those views that have been properly calibrated and generate a dataset file for the visual hull construction algorithm.

6.2 Synthetic data

Techniques to capture images, segment them, capture calibration patterns and calibrate the camera setup have been described in the previous sections. To test these tools, I chose to first develop a synthetic data set where all parameters can be controlled, and no noise is present in the readings. I built a simple 3D model by displacing points on a sphere to make it look like a spiked shell. It is a fairly complex model with sharp features and occlusion. I created a synthetic (animated) turntabe on which I placed a calibration pattern and took a sequence of 27 images around 360 degrees. Then I captured images of the 3D model. These files form the input to the tool chain. A set of shell scripts have been developed, that will simplify the process of 3D reconstruction. The steps in the process of reconstruction are:

1. Extract an ordered set of 64 points from each calibration pattern
2. Pass these points to the camera calibration routine to get a set of 3x4 camera matrices
3. Run a segmentation pass on each image of the object to generate png images with the background set to transparent
4. Create a dataset file with the matrices, the extents of the reconstructed object in 3d space, and the paths to the segmented files
5. Run the visual hull generation routine on the dataset file to get the final reconstruction

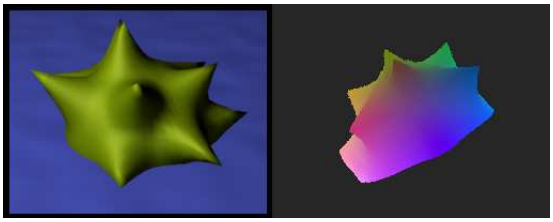

Figure 8: Sample input image and resultant visual hull for spiked shell dataset

6.3 Real data

The next stage is to check if the system works for actual data captured by the webcam, with the object placed on an actual turntable and lit by the indirect light source as described earlier.

One of the key problems is that the turntable has to be aligned to the exact same position while starting calibration image capture, and object capture. Otherwise the 2 sets of images will not correspond to each other and we will have wrong calibration data. To fix this, I used the simplest solution possible. A marker is attached to the axle of the turntable, and another corresponding marker is attached to the fixed base. Before capturing images, these markers are manually aligned.

Another problem faced here is that when the entire background consists of a single color(blue) sheet, the camera tends to desaturate the image. Therefore, the background colors as seen when no object is placed and when an object is placed, are very different. It turns out that the camera performs automatic white balance and saturation control. So, when it sees only one color in the scene, it desaturates everything so that the scene turns gray. This was a hurdle because I could no longer get useful background images. To fix it, I placed a small white strip of paper just within the view frustum of the camera, so that it would not perform such aggressive white balance. It works better now even without any object on the table. Another possible solution to this is to set the camera to manual exposure(which may or may not be possible depending on the camera). More simply, we can use the clone tool from any image processing software to erase the object from one of the object images, thereby obtaining a good approximation of the background image. This works very well in practice.

Once the data is obtained, the following process was followed:

1. Extract an ordered set of 64 points from each calibration pattern: This is one of the hardest problems to fix. When a camera captures images, the colors as captured by the camera are very different from the color that we perceive. I found that the thresholding functions were failing on several calibration images. I introduced backtracking so that the threshold can be lowered if we overshoot and find too many boxes. I also added an option of maximum distance between two boxes, to prevent 2 parts of the same box from being labeled as 2 different boxes. This fixed a lot of errors, but there are still problems with a few images. The system was automatically able to extract calibration parameters for 86 of 89 images in this particular test. The remaining patterns, for which correspondence information cannot be extracted, are discarded along with their corresponding object images.

2. Pass these points to the camera calibration routine to get a set of 3x4 camera matrices: This process works as long as the previous step does not produce erroneous results. Since there is no simple check that we can perform to find out if the result of the previous step is valid, the output matrices sometimes turn out to be erroneous. The simplest fix here is to build a tool that will allow the user to visually verify each camera matrix before it is used to perform reconstruction. A simple voxel renderer takes a file containing a set of calibration matrices as input and projects a 3D box using the matrices. This projected box is expected to rotate as the turn table rotates. If

in some cases the box seems to be out of place or distorted, or entirely missing, the user can delete the calibration data corresponding to this matrix, and specify to the system that the corresponding object image should be ignored.

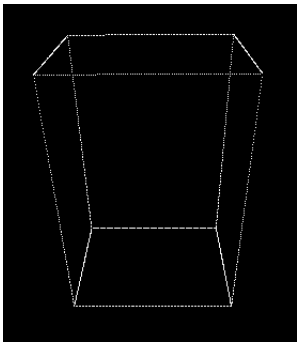

Figure 9: Camera matrix visualisation

3. Run a segmentation pass on each image of the object to generate png images with the background set to transparent: This step is hard because the camera keeps on adjusting the exposure and white balance, drastically desaturating the scene in some images. This means that the background image we are using to segment out the object is no longer truly valid. The key idea with segmentation is that it should conservatively label pixels as background. This means that it is all right if a few images have a lot of spurious data, but no image should be missing any data. Since the process of finding a visual hull involves volume intersections, any random spurious data that is found in some images will be discarded automatically.

Figure 10: Reconstructed visual hulls of frog dataset

7 Future Work

1. Better calibration using optimization techniques like bundle adjustment
2. Textured rendering of visual hull: Although the Image Based Animation codebase has been modified to support textured rendering, this feature is currently not working and needs some debugging to fix it.
3. Relighting: The relighting system requires images which have been captured under multiple lighting conditions, from multiple view points. Currently, it is not possible to supply reliable lighting data to the system.

8 Conclusion

The reconstruction process is mainly limited by the hardware and drivers. Better exposure and white balance control will lead to better segmentation, which in turn will lead to more accurate reconstruction.

Another source of error could be slight errors in turntable positioning which eventually add up to produce a significant negative impact on the result. A technique like bundle adjustment would help to provide better results in this case.

Visual hull based algorithms require cameras to have good coverage of the hemisphere surrounding the object. Otherwise, there will be observable differences between the actual object shape and its visual hull. For example, by restricting ourselves to a single camera elevation, it can be seen that a short protrusion behind the frog is present in the resulting hull, which is not part of the actual model. More coverage is the obvious solution, but it can be hard to calibrate the cameras for situations where camera elevation is too low (and the pattern will almost not be visible).

Further, no view samples are available for the object from the lower hemisphere. This can cause problems in the animation system, because even though the visual hull assumes that the bottom surface is flat, since we have no camera samples for the bottom surface, the rendering algorithm will be forced to output some default colour for such surfaces.

9 References

1. OpenCV camera control and calibration: opencv.willowgarage.com/
2. Image based Animation: http://www.cse.iitb.ac.in/ biswarup/projects/Motion/