Exploring Efficient Image-Based 3D Reconstruction and Rendering

Soohark (Scott) Chung
Department of Computer Science
Stanford University
Stanford, CA; U.S.A
soohark@stanford.edu

Abstract—This project proposes a pipeline for producing 3D content from 2D images. It goes through each step in the pipeline and evaluates whether or not such a pipeline is feasible.

Keywords—mesh decimation; structure from motion

I. INTRODUCTION

Despite continued advancements in 3D web and mobile technologies such as WebGL and faster mobile graphics processors, there is a lack of 3D content on the web. I decided that the main cause of this was that there wasn’t an easy way to produce 3D content. Traditional methods require an artist to model a 3D mesh and to paint textures manually. Instead, I propose a pipeline for producing 3D content from a set of 2D images. If such a pipeline could be built, we may not be too far off from using 3D content for popular usage cases such as online shopping or social media. I go through each step in this pipeline and test and evaluate the algorithms I chose. The end goal is to determine whether or not this sort of pipeline is feasible for widespread adoption using current technologies.

II. PROPOSED PIPELINE

Figure __ outlines my proposed pipeline. I’m going to go over the pipeline and explain some of my decisions here. First I take a set of 2D images of an object from different angles. Next, I use PMVS [2], a structure from motion algorithm, to derive a point cloud from these images. I decided to use structure from motion, because it would make pipeline accessible without special hardware such as high resolution 3D scanners. In order to get the camera parameters that PMVS needs as input, I use VisualSFM, which uses SIFT descriptors and pairwise image matching to derive the camera positions. Next, I use Poisson surface reconstruction [7] to create a 3D mesh. I decided to use PMVS and Poisson surface reconstruction, because both of these algorithms are very popular accepted state-of-the art algorithms with open source implementations. Next, I project the images back on to the 3D mesh using he camera parameters derived with VisualSFM in order to assign vertex colors to the 3D mesh. Next, I use mesh decimation to reduce the size of the mesh. This step is necessary to try to achieve minimum file sizes for web purposes. Finally, I wanted to implement some sort of view-dependent rendering algorithm. I didn’t get to this step during the course of this project, but it helped motivate my decision to use vertex colors instead of texture mapping to render colors. Vertex colors work better than texture mapping with most view-dependent rendering algorithms. Also, by using vertex colors, I avoid having to publish high-resolution texture files along with the mesh.
In this section, I’m going to go over some relevant past works.

A. PMVS

PMVS, patch-based multi-view stereopsis [2], is a popular method for dense structure from motion. The algorithm works by first matching corner and blob features in different images. Then the algorithm expands initial matched patches by adding neighboring patches to them. Then the algorithm filters the expanded patches to filter out erroneous matches and to enforce geometric consistency. By doing this expansion and filtering iteratively, you can retrieve a point cloud that covers the entire surface of the 3D object.

B. Poisson Reconstruction

Poisson reconstruction [7] is a popular surface reconstruction method that models reconstruction as a spatial Poisson problem. It handles small amounts of noise well; however, it tends to return blobby looking results when the input doesn’t have enough details.

C. Mesh Decimation

I looked at several options for appearance based mesh decimation techniques. Some of the early works in the area can be seen in [3] and [4]. The methods are based on new error metrics called imperceptibility distance. This error metric basically measures the maximum contrast in the texture for the worst-case spatial frequency (based on the size of the collapsed edge) that can be affected by collapsing an edge. The paper presents examples of dense meshes that were simplified up to 90% with little noticeable changes in the rendered results. However, these papers use texture mapping rather than vertex colors. The second paper also chooses to simplify the mesh optimizing from a predetermined viewing angle. This means they preserve the geometry of the mesh by considering whether or not an edge is visible and whether or not it is part of the silhouette. For my intended purposes, the imperceptibility distance that they defined is useful, but I’ll have to modify their algorithm to preserve the overall geometry. In the end, I chose an algorithm that extends the quadrics error approach [5], [8].

D. View-dependent Rendering

I also looked at past works in view-dependent rendering while formulating my pipeline. An early paper in image-based rendering [6] basically boils down to looking up each polygon color in an image closest to the current viewing angle where the polygon you want is visible. The paper covers some practical implications of how to make the algorithm more efficient and a strategy for filling holes with the most likely color based on the polygon’s neighbors. The eigen texture method [1] is an efficient method for image-based rendering for interactive settings. The method basically involves doing PCA on each triangular patch viewed from different angles and representing the texture as a linear combination of PCA components. This way the weights on the components can be interpolated between precalculated angles.

IV. PMVS/POISSON RECONSTRUCTION

Even though PMVS is a very popular state-of-the-art algorithm for dense point cloud reconstruction, it still has its faults. It requires a fairly large number of photos to get a good

---

**Figure 1:** Outline of proposed pipeline
point cloud. Also, it needs corners and textures to latch onto. Therefore, it fails to capture points on smooth surfaces without texture, and has empty patches in its point clouds. Another problem is that if there are detected edges in the image where the geometry is actually smooth (like at the edges of the printed letters on a label), the normal direction associated with the points at those edges isn’t stable. These issues cause problems when using Poisson surface reconstruction. The unstable normals and patches without any points which are often adjacent cause Poisson reconstruction to return bumpy, deformed surfaces. Also, the result of Poisson reconstruction tends to have a blobby look to them due to the nature of the algorithm.

Figure 2: Mesh reconstruction workflow; a. camera positions, s; b. dense structure from motion; c. Poisson surface reconstruction; d. colored vertices
V. MESH DECIMATION

Mesh decimation is necessary to reduce the number of vertices in the mesh. For example, in the undecimated mesh for the salt container there are close to a million vertices. Now, if I had decided to use texture mapping, this step would be much easier. I could use any mesh decimation algorithm I wanted that preserves geometric features. Then, I could re-parametrize the mesh and re-project the textures back onto the mesh to get new texture coordinates. To begin my exploration, I implemented a baseline mesh decimation algorithm based on the quadrics error metric [5]. At each vertex of the triangular vertex, you can calculate the normal direction as an average of the normals of the adjacent faces. Then, at each face, you can fit a quadric surface to the vertices with associated normals. The distance between the center of the quadric surface to the center of the flat triangular face can be used as an error metric for the geometry. I borrow starter code for a basic mesh decimation assignment for CS348a, and build my implementation on top of it. My implementation builds a priority queue of vertices based on the quadrics error associated with each vertex allowing it to quickly find and collapse edges that would introduce the smallest quadrics error.

Since I wanted to use vertex colors, I must use a mesh decimation algorithm that preserves areas where there are contrasts in color as well as geometric features. I looked at several appearance-based mesh decimation algorithms. After much deliberation, I decided to implement an extended version of the quadrics error metric [8]. I chose this method, because it worked well with vertex colors and it uses geometric features that aren’t optimized for a particular viewing angle. The new quadrics error metric works by mapping each vertex to a higher-dimensional space where each vertex now has the position data, $x, y, z$, and color data, $r, g, b$. Then, you can do the same quadric calculations by fitting a 6-dimensional hypersurface to each face. Now, since I eventually wanted to do view-dependent rendering, I explored the idea of adding rgb color data for each image that I used to construct the mesh. However, this would have meant that I needed to do the quadrics calculations in $3+3n$ dimensional space. This quickly blows up if you have more than 15 or so images, which is necessary for PMVS. Therefore, I needed to take an average of all the textures before I passed it through the algorithm. You can control how much impact the color information has by scaling the color values with a constant scaling factor. It’d be interesting in the future to see if different ways of combining the textures work better than simple averaging. You could also do something more complex like taking the maximum gradient across all the different textures when calculating the quadrics.

VI. MESH DECIMATION RESULTS

The version of the mesh decimation algorithm that takes color into account obviously works better. In FIGURE you can see that edges across which there are large color contrasts is preserved much better with the improved algorithm. In this example, the decimated meshes have 1% of the number of vertices as the full mesh (936693 vs. 9366 vertices). Consequently, the file size is also reduced by a similar order of magnitude. The full mesh file is 105mb while the decimated mesh files are 691kb in size.

I also calculated some error metrics to evaluate how close the decimated mesh is to the original mesh rendered out from the same camera angle. The repeatability was calculated by running feature detectors on the renderings of the original mesh and the decimated mesh.

![Image of mesh decimation results]

Figure 3: Repeatability of SIFT detector

Figure 4: Repeatability of Harris detector
You can see that the improved algorithm works better for various metrics.

VII. CONCLUSION AND FUTURE WORK

So, going back to my original motivation for this project, I have to say, I don’t think the pipeline I proposed is suitable for wide adoption yet. The main bottleneck in the pipeline is structure from motion. We’re probably far away from solving this problem. However, the only reason I wanted to use structure from motion was to make the pipeline accessible through widely available hardware. With the release of the new Microsoft Kinect and Google’s Tango project, it might not be too idealistic to think that everyone will have 3D scanners in the future. In that case, we could use a hybrid approach and use 3D sensor data to help improve the structure from motion step.

Also, I did not get to view-dependent rendering during the course of this project as I had originally planned, and I would like to expand on this project in the future. I had a lot of ideas for view-dependent rendering. I would like to implement some variation of the Eigen Texture [1] technique. I would also like to explore ways of dealing with high frequency textures. I would like to try to devise an approach that preprocesses the images using a low-pass filter. Then, at the end of the pipeline, I’d like to try to represent the high frequency content that’s been filtered out using small repeatable tiles of textures.

VIII. APPENDIX

The PMVS2 package, VisualSFM, and Meshlab was used for this project. Mesh decimation was implemented in Visual Studio using C++, openMesh, and Eigen. Please look at the READ_ME.txt file for information on where to download all the tools necessary and run the pipeline.

REFERENCES