**Abstract:**

Our project presents an implementation of the Structure from Motion (SfM) technique for 3D reconstruction from a set of 2D images. SfM recovers the 3D geometry of a scene by estimating the camera poses and triangulating 3D points from corresponding 2D feature points across multiple views. The core algorithm we followed consisted of four main steps: (1) calibrating our camera using the AprilTag fiducials to obtain the camera matrix with the intrinsic parameters, (2) detecting and matching feature points between image pairs using SIFT, (3) estimating the essential matrix and camera poses, and triangulating the matched 2D points into 3D points to build the initial point cloud, and (4) adding new views by matching to the previous point cloud and triangulating additional 3D points. After having successfully completed all these steps, we were able to visualize the resulting 3D point cloud along with the estimated camera locations and orientations. Additionally, we used the open-source COLMAP package to reconstruct the same scene, and compare its output against our own SfM implementation. Finally, we created an additional visualization which not only visualized the final point cloud and cameras, but instead used colored points based on their corresponding pixel values from the images. Overall, we were successfully able to perform a colorized 3D reconstruction from our 2D images that included the camera positions and orientations, and especially after denoising our point cloud, it looked incredibly clear.

**1. Introduction:**

In 1981, the seminal work of Longuet-Higgins established the formulation of what is now known as the Essential Matrix. This matrix, which was later generalized to be known as the Fundamental Matrix, provided the foundation to recover relative camera poses from image correspondences. In the following decades, this discovery sparked intense research, leading to algorithms for automatic feature detection, matching, camera calibration, and maximum likelihood estimation techniques. By leveraging all fundamental principles from multiple view geometry, the ability to recover the 3D structure and camera poses just from scenes from 2D images alone was discovered, and named Structure from Motion (SfM). Today, SfM is a widely useful capability in not just Computer Science, but also in Robotics, AR/VR, Forensics, Photogrammetry, and many more.

The goal of this project is to build upon the foundations laid by Longuet-Higgins and subsequent researchers, and use the power of SfM for 3D reconstruction from a set of captured 2D images. Our project implements the core SfM pipeline, using the pre-made dataset that included the April Boards which made it easy for camera calibration. Subsequently, the first step was to calibrate our camera using pre-written functions from earlier pre-sessions and labs. Next, we had to implement robust feature detection so that we could match keypoints across two different images. After having found all the matched keypoints from the first two images, we then had to calculate the Essential Matrix, create a RANSAC inlier mask for valid matches, calculate the camera matrices for both views, and then finally triangulate the matches to create a 3D object point cloud. Along the way we used 2D and 3D visualizations to ensure correctness and after undergoing this verification process, we proceeded to automate this process so that it could be repeated for more than two photos, allowing us to have multiple angles of the scene. With our PnP solver pipeline in place, we were able to create a 3D point cloud using as many images from our dataset as we desired.

As the main bulk of the work was done, we proceeded to implement an extension feature. We chose to include the true color of the scene in our visualization. To achieve this, we extended our approach to color the 3D points based on their corresponding pixel values from the images. Inherently, we also visualized these colored 3D points in an interactive 3D plot. Since our SfM implementation was now complete, we compared our results against the reconstruction obtained from the popular open-source COLMAP package.

Throughout this paper we will go into depth about how we were able to implement each of the steps described above as well as discussing the strengths and weaknesses of each implementation. By thoroughly documenting our approach, insights, and results, we hope this project can serve as a valuable resource for understanding the core concepts and practical considerations involved in SfM and multi-view 3D reconstruction.

**2. SFM Pipeline**

**2.1 Camera Calibration**

The first step in our SfM pipeline was to calibrate our camera in order to obtain the intrinsic parameters of the camera used to capture the input images. As we covered in lecture, these intrinsic parameters model the internal properties of the camera, such as focal length, principal point, and distortion coefficients, which are all necessary for an accurate 3D reconstruction.

Because we used the pre-made dataset, we were able to leverage the AprilTag fiducials for our camera calibration. For those who aren’t familiar, the AprilTags are black and white square markers, each with a unique pattern that can be detected and identified. The pre-made dataset included images of two AprilBoards with known marker positions — a coarse board and a fine board. Thus, the `detect\_aprilboard` function, taken from the pre-sessions and labs, will detect these markers in any given image using the `pupil-apriltags` library.

Next, the `calibrate\_camera` function, also obtained from the pre-sessions and labs, performs the calibration process. It iterates through the calibration images, detects the AprilTags in each image by calling the `detect\_aprilboard` function, and stores the corresponding 2D image points and known 3D board points. In its last step, it calls the OpenCV function, `calibrateCamera`, which uses these 2D-3D correspondences to estimate the camera's intrinsic matrix and distortion coefficients.

By successfully calibrating our camera and obtaining the camera's intrinsic matrix used for our SfM pipeline, we can ensure that projections between 3D world points and 2D image points will be accurate.

**2.2 Keypoint Detection and Matching**

After calibrating the camera, the next step is being able to establish a correspondence between 2D points across two images. We accomplished this through keypoint detection and matching using the Scale-Invariant Feature Transform (SIFT) algorithm.

Our first function, `detectKeypoints`, utilizes OpenCV's SIFT implementation to detect keypoints and compute their descriptors for a given image — it simply returns both of these values. Our next function, `matchKeypoints` , performs the matching between two sets of keypoints and descriptors. In order to find correspondences between two sets of keypoints, we used the brute-force matcher with the L2 norm and cross-checking to obtain symmetric matches. This function returns a list containing the top matches based on descriptor distance, as well as the corresponding 2D coordinates for the matched points in both the first and second image. Finally, by tuning the `GOOD\_MATCH\_PERCENTAGE` parameter, we could control the trade-off between the number of matches and their quality. After some cross-validation, we found that using the best 20% of matches sufficiently denoised the matches while still providing enough correspondences for later steps.

It is important to note that while the process that we have just described is for the first two images, it will later be generalized so that it can be used to add subsequent images in order to grow our point cloud and have a more accurate 3D reconstruction.

**2.3 Base View**

After having calculated our camera matrix and the top match key points across two images, we could now initialize the iterative reconstruction process by first obtaining the point cloud using the first two views. We will refer to this as the base view.

Using the 2D coordinates of corresponding points across the base view, we estimated the Essential Matrix `E` and recovered the relative camera poses between the two views using OpenCV's `findEssentialMat` and `decomposeEssentialMat` functions.

Conveniently, the `findEssentialMat` function, when using the RANSAC method, returns a set of “inliers,” which is a subset of the matches that correspond to the decided upon Essential Matrix. In other words, we could now effectively tell which matches actually corresponded to some distinct mapping between the two images. By converting this into a mask, `base\_inliers`, we denoised the reconstruction, filtering out outlier matches between any two images.

The next step was computing the two camera matrices for the two views. We located the first camera at world coordinate origin (identity matrix), something that won't apply to any other camera; each subsequent camera position can be calculated by rotating and translating the previous camera after calculating some pose estimation. For the second camera, this can be done very simply by using `decomposeEssentialMat`, which returns two rotation matrices and a translation vector. Per [cite source here], there are four possible orientations for the camera given the essential matrix decomposition, so we visually inspected the options to determine the correct transformation.

With these first two camera matrices and the coordinates of the matches, we were able to obtain the initial 3D point cloud of the base view using `cv2.triangulatePoints`. This initial reconstruction from the base view formed the foundation for our subsequent steps, where we incrementally added new views to expand the 3D point cloud.

**2.4 Adding Additional Views via PnP Solver**

In the last step of our pipeline, which comes after obtaining the initial 3D point cloud from the base view, we fully developed our iterative approach that incrementally added new views and points in order to expand upon the 3D reconstruction. For the remainder of Section 2.4, we will refer to the new view we’d like to add as the “current view” and refer to the last view added before the current view as the “previous view”.

The processing of adding additional views was similar to that of the base view construction, but differed in some key ways. Namely, we can no longer use the Essential Matrix to determine the rotation and translation between two camera views. This method required two 2D images between which we’d like to find a correspondence, while we now have a 3D point cloud and a 2D image. If we used the Essential Matrix, we would end up with a number of correspondences between 2D images, but these correspondences would be independent (ie. not in the same world coordinate system). To estimate the subsequent camera pose in the *same* world coordinate system, we must use a Perspective-n-Point (PnP) solver.

We began by calculating key point matches between the current and previous view, using the same method described in Section 2.2. We then used the `findEssentialMat` function between the previous and current keypoints and descriptors, but this time, we used only the returned inliers and threw out the Essential Matrix E itself. This allowed us to determine which of the matches were actually matches between the two views.

We then found the subset of new keypoints that have corresponding matches in the previous image; that is, the points that appear in matches between the current and previous view as well as between the previous and previous previous view. These will therefore be the 2D points that appear in our current matches that have corresponding 3D coordinates! Once we find the correspondences using a brute-force search, we can pass the set of 2D points and their 3D correspondences in the world coordinate system to `cv2.solvePnPRansac` to calculate the new camera pose.

Thus, for each new view, we first detected its key points and matched them against the key points from the previous view using `matchKeypoints`. Then, we performed a RANSAC-based filtering and the correspondence mask in order to obtain the subset of inlier matches between the new view and the existing 3D point cloud. And subsequently, using a brute force search, we found the 2D-3D correspondences between the new view's inlier 2D points and the corresponding 3D point from the previous reconstruction. To complete one iteration, with all the correspondences and points, we simply performed a triangulation to add the new points to our point cloud. Because we recursively update our variables, we can theoretically repeat this process for any number of images, but we will later discuss why this is not necessarily the case in practice.

**3. Extension: Colored 3D Reconstruction**

While our implemented SfM pipeline successfully reconstructed a 3D scene from a set of 2D images, we also wanted to incorporate color information from the input images to create a more visually appealing and informative reconstruction. We achieved this by mapping the color values of the corresponding pixels onto the triangulated 3D points, resulting in a colored point cloud visualization.

The key step in this extension was to associate the appropriate color information from the input images with each 3D point in the final reconstruction. We implemented this in the `find\_colors` function which takes an input image and a set of 2D points as arguments. For each 2D point, we define a local square region around that point, with the size controlled by the `region\_size` parameter, extract this local image region, and compute the average color value across all pixels in the region. The averaged RGB color value for each 2D point was then appended to a list.

In order for our extension to fully work in our final visualization, we integrated the color extraction process at two stages: Base View and Adding Additional Values.

After triangulating the initial 3D point cloud from just the first two views, we extracted the corresponding colors from the first image and appended them as an additional column.Then, for each subsequent view that was added to the reconstruction, we extracted the colors from the current image based on the 2D points, and appended them to the running list we had. Finally, when visualizing the complete 3D reconstruction, we combined the array of all the 3D points with the list of colors to create our final colored point cloud.

Our final `visualizeObjectPoints` function visualizes the 3D points with their associated colors in an interactive fashion which allows for visually appealing representation of the reconstructed scene. By incorporating color from the input images, we felt that our extension enhanced the visual quality and interpretability of the 3D reconstruction, especially when trying to see which points mapped to which objects from the original images.

**4. COLMAP Comparison**

**5. Takeaways/Advice/Lessons Learned**

**References**

Longuet-Higgins, H. C. “A Computer Algorithm for Reconstructing a Scene from Two

Projections.” *Nature*, vol. 293, no. 5828, Sept. 1981, pp. 133–35,

<https://doi.org/10.1038/293133a0>.

*OpenCV: Epipolar Geometry*. https://docs.opencv.org/3.4/da/de9/tutorial\_py\_epipolar\_

geometry.html. Accessed 6 May 2024.

**Next Steps:**

* Add strengths and weaknesses throughout the sections (follow the comments)
* Add Images
  + Should we have a section with just visualizations or add them throughout
    - If we add them throughout then we will need to have visualizations that improve as we go through the report (for example a two camera visualization followed by 9 camera one, or color vs no color)
* Are function names good throughout the explanations?