ME 735: CGPM
Multi-View Reconstruction
Project Final Report

Darshan Makwana, Harsh Kavediya, Vighnesh Nayak, Lakshay Garg

November 2024

1 Objective

Rendering 3d objects using a set of 2-D images taken at multiple orientations
We aim to reconstruct a 3d object in it's voxel representation by continuously optimizing it against a prior of given images taken at multiple orientations

2 Algorithm

2.1 Graph Cut Refinement

The `VolumetricGraph` class is designed to segment a 3D voxel volume through graph-based optimization, allowing refined object representations in voxel grids. The algorithm involves multiple key steps:

- **Initialization and Hull Surface Creation**:
  - The constructor initializes parameters such as volume, voxel intensities, centers, and a regularization parameter, `_lambda`.
  - The `init_hull_surf` method iterates through the volume to identify voxels as part of the object's surface or interior. Boundary voxels, which neighbor invalid (background) voxels, are labeled as hull surfaces, while interior voxels are retained for further processing.
- **Graph Construction**:
  - Using the `maxflow` library, `init_graph` creates a graph where each voxel is represented as a node. Neighboring and terminal (source and sink) links are added to the graph to enable segmentation.
- **Terminal Links**:
  - Terminal links are created in the `add_t_links` method. Boundary voxels, defined as those with at least one neighboring background voxel, are connected to the sink with infinite weights, encouraging these voxels to be assigned to the background.
  - Interior voxels are connected to the source with weights derived from the regularization parameter and voxel size, `_lambda * voxel_size`. This finite weight constrains interior voxels to belong to the foreground.
- **Neighbor Links**:
  - The `add_n_links` method links each voxel to its neighboring voxels within the volume. Weights between these neighbors are computed based on intensity differences, using a Gaussian similarity function: $W = \exp\left(\frac{-D_{ij}}{2\sigma^2}\right)$, where D_{ij} is the squared intensity difference and σ controls the sensitivity to intensity changes.
  - This approach ensures smooth transitions and penalizes large differences between neighboring voxels, promoting coherent segmentation.
- **Graph Cut Computation**:

  - The `run` method computes the graph cut by maximizing flow in the graph, segmenting the voxel volume into background and object regions based on the optimized configuration of the source and sink.
  - After segmentation, `get_refined_volume` returns the resulting refined volume, identifying the final segmentation as a binary mask.

- **Output and Storage**:
  - The final segmented volume is stored in JSON format. Each voxel in the refined hull is saved with its coordinates and a constant color, `#EE4B2B`.
  - This JSON data provides a detailed visual representation of the segmented model, suitable for rendering or analysis.

2.2 Ray Casting

The algorithm performs volumetric rendering by ray-casting from a camera position through a voxel grid using multiple images and camera poses. For each image, rays are projected from 2D pixel coordinates into 3D space. Key points include:

- **Voxel Intersection**: For each depth along a ray, the corresponding 3D point is mapped to voxel grid coordinates.
- **Color Accumulation**: Colors are accumulated in each voxel encountered along the ray if it lies on the object's surface.
- **Normalization**: RGB values are averaged per voxel based on the count of hits.
- **Output**: Voxel data, including color, is saved in JSON format for visualization.

3 Formulation

3.1 Image Generation

The `Scene` class manages 3D scene rendering and camera setup to generate sample images. Key aspects of the algorithm are as follows:

- **Scene Initialization**:
  - Initializes with image dimensions, loads a 3D mesh using `pyrender`, applies a transformation to the pose, and sets ambient lighting.
  - Creates a perspective camera attached to a `pyrender` scene node, allowing dynamic view sampling.
- **View Sampling**:
  - `sample` creates random or user-specified camera poses by combining rotation matrices for the X, Y, and Z axes and positions the camera for the scene view.
  - Captures and enhances images, and optionally, generates depth maps indicating object positioning in the scene.
- **Image Generation and Intrinsics**:
  - `generate_samples` captures a series of images, each with associated camera poses and binary masks.
  - `intrinsics_matrix` calculates the camera's intrinsic parameters from the projection matrix, essential for accurate depth calculations and perspective.

3.2 Visual Hull Generation

Key Idea:
The projection of a 3D point (X, Y, Z) in world coordinates onto the 2D image plane is given by:

$$x_{proj} = \frac{f_x \cdot X}{Z} + c_x, \quad y_{proj} = \frac{f_y \cdot Y}{Z} + c_y$$

where (f_x, f_y) are the focal lengths, and (c_x, c_y) are the camera center coordinates.
Steps:

- We first define the number of voxels along each dimension. The more the number of voxels, the better will be the reconstruction but higher will be the computational time.
- We then get the camera parameters (focal lengths and the camera center coordinates) and define the camera position and the object bounds
- The voxel centres are computed and for each image the position of voxel centres is determined with respect to the camera orientation by a transformation
- The projection of voxel centers in the image plane is determined by the above key idea
- The voxels which lie in the image mask and the image bounding box are termed as valid voxels
- The average voxel intensities are also calculated so as to be used in refining the Visual Hull
- The object mask is computed by considering the it to be made of voxels that are present in all the images
- The result is written into a JSON format, consisting of voxels of the object mask and their corresponding color

3.3 Refining the Visual Hull

Key Idea: The weight W between voxels based on intensity similarity in RGB space is defined as:

$$W = \exp\left(-\frac{D_{ij}}{2\sigma^2}\right)$$

where D_{ij} is the Euclidean distance between the RGB intensities of voxel i and voxel j, and σ is a scaling factor.
The energy function for graph cut segmentation is:

$$E(S) = \sum_{p \in S} D(p) + \lambda \sum_{p \in S, q \in N(p)} V(p, q)$$

where $D(p)$ represents the data cost for voxel p, and $V(p, q)$ is the pairwise smoothness cost between neighboring voxels p and q in the set S. Here, λ is a regularization parameter balancing data fidelity and smoothness.
Steps:

- Initializing the Hull surface
  - The valid voxels form the interior of the hull surface
  - The voxels in the interior that are surrounded by invalid voxels (background) form the boundary of the hull surface
- Initializing the graph
  - The voxels are considered to be the nodes with unique node ids
  - Terminal links with infinite weight are added between the boundary voxels and the sink
  - Terminal links with finite weight are added between the boundary voxels and the source
  - Neighboring links are added between all the nodes and their neighboring node in the graph with weight corresponding to the heuristic formula in the key idea

- Computing the Graph Cut
  - The graph cut is formulated as an optimization algorithm given by the key idea
  - We get the refined visual hull using the predefined graph cut solver in the maxFlow module
- The result is written into a JSON format, consisting of voxels of the refined visual hull and their corresponding color

4 Miscellaneous Experiments Performed

- Checking the effect of increasing the number of voxels along every axis (resolution) on the reconstruction quality
- Checking the effect of increasing the number of images on the reconstruction quality
- Pose estimation from real world videos

  1. Pose estimation using Phone sensors
     - We used data from the accelerometer and gyroscope of the phone to detect pose
     - The data used provided a very bad reconstruction results
     - We believe that main reason for this was poor accuracy of the phone sensors
  2. Pose estimation using ArUco
     - We set up our object on a ArUco board printed on a sheet of paper.
     - We calibrated our camera using OpenCV, based on the ArUco checkerboard size, by capturing images in various orientations.
     - A video was recorded of the object placed on the ArUco board.
     - We then used the solvePnP function in OpenCV to determine the pose of the image plane with respect to the camera.
     - However, the data provided very poor reconstruction results.
     - This was likely because the ArUco markers are highly sensitive to even slight changes in orientation, making pose estimation challenging under varying angles.

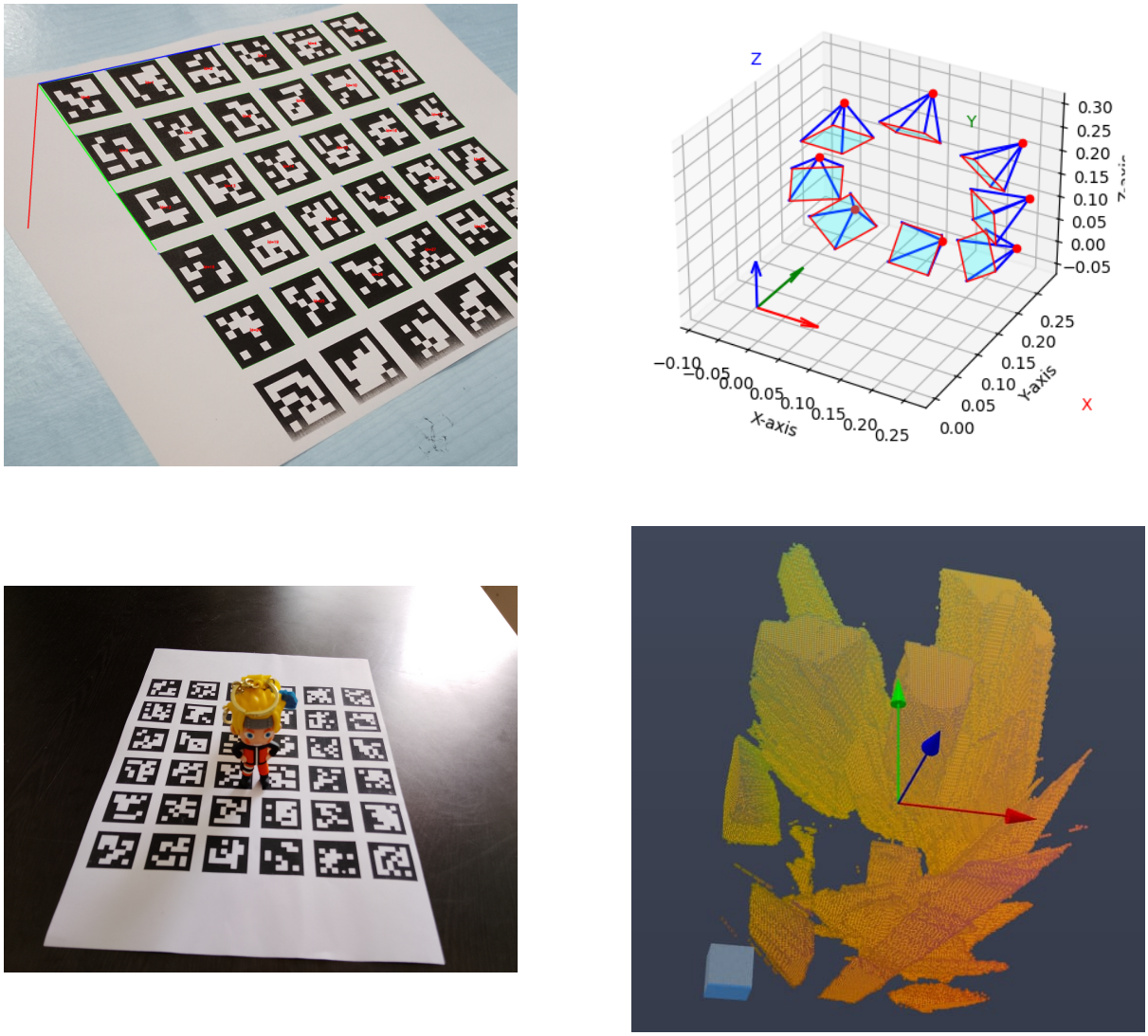

Figure 1: Object and its reconstruction using ArUCo for pose estimation

5 Key Results

- By visual inspection, the algorithm provides a fairly good reconstruction
- Increasing the number of images produced a better quality reconstruction
- Increasing the voxel resolution led to better reconstruction by effectively capturing more complicated features
- Graph cut refined the object and helped capture concavities better

Grid Size	10 images	25 images
50x50x50		
100x100x100		
200x200x200		

Table 1: Reconstruction of Objects with different number of images with different resolutions without graph cut refinement

Grid Size	10 images	25 images
50x50x50		
100x100x100		
200x200x200		

Table 2: Reconstruction of Objects with different number of images with different resolutions after graph cut refinement

Without Graph Cut | With Graph Cut

Table 3: Comparision of reconstruction with and without graph cut

Figure 2: Final Reconstruction After Ray Casting

References

[1] K. Kolev, M. Klodt, T. Brox, and D. Cremers, "Continuous Global Optimization in Multiview 3D Reconstruction," *International Journal of Computer Vision*, vol. 84, no. 1, pp. 80–96, Aug. 2009.

[2] F. Steinbrücker, C. Kerl, and D. Cremers, "Large-Scale Multi-resolution Surface Reconstruction from RGB-D Sequences," in *2013 IEEE International Conference on Computer Vision*, 2013, pp. 3264-3271.

[3] G. Vogiatzis, P.H.S. Torr, and R. Cipolla, "Multi-view stereo via volumetric graph-cuts," in 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05), 2005, vol. 2, pp. 391–398.