CS436 Project Phase 3

December 2, 2025

Overview

Most of you have successfully reconstructed specific structures (e.g., a single wall, a corridor, or a desk) using the Incremental SfM pipeline. To complete the project correctly, we must demonstrate how the tour application navigates across these structures—for example, moving from "Wall A" to "Wall B," or from a corridor to a room.

However, due to a lack of visual overlap between these distinct areas during data capture, the reconstructions often exist as separate `.ply` files with independent coordinate systems. To create a cohesive Virtual Tour for Phase 3, these independent point clouds (and their associated camera poses) must be registered (aligned) into a single **Global Coordinate System**.

Important: Implementing Three.js for only a single wall or a small portion of the scene is not acceptable. You must demonstrate a proper virtual–tour experience, where the user can move *between* different walls/different portions of a room, or areas and view all aligned point clouds together in one continuous 3D environment.

Interactive Virtual Tour Implementation

To build the final viewer, organize your independent reconstructions (point clouds and camera poses) into separate Three.js **Groups**. This allows you to align distinct scene sections (e.g., merging "Wall A" and "Wall B") by transforming the Group object, ensuring navigation nodes remain geometrically consistent with the point cloud. Construct a **View Graph** where camera poses act as nodes connected by shared visibility. When a user interacts to navigate, determine the optimal target node and execute a smooth transition: use Linear Interpolation (**Lerp**) for position and Spherical Linear Interpolation (**Slerp**) for orientation. Crucially, when calculating interpolation targets across different Groups, query the node's **World Position** rather than local coordinates to account for the Group's transformation. Simultaneously, apply a CSS opacity cross-fade between the starting and ending source images to create a seamless sense of 3D motion.

Method 1

Mathematical Concept

We define a transformation matrix T composed of a rotation R and a translation t:

$$T = \begin{bmatrix} R_{00} & R_{01} & R_{02} & t_x \\ R_{10} & R_{11} & R_{12} & t_y \\ R_{20} & R_{21} & R_{22} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1}$$

Points in the second cloud (P_{old}) are updated via: $P_{new} = T \times P_{old}$.

Sample Python Script for Merging

```python
import open3d as o3d
import numpy as np
import copy

def merge_partial_clouds():
    # 1. Load your separate PLY files
    pcd_A = o3d.io.read_point_cloud("wall_A.ply")
    pcd_B = o3d.io.read_point_cloud("wall_B.ply")

    # Paint them for visualization
    pcd_A.paint_uniform_color([1, 0.706, 0])      # Orange
    pcd_B.paint_uniform_color([0, 0.651, 0.929])  # Blue

    # 2. Define Transformation for Wall B relative to Wall A
    # Example: Rotate 90 deg around Y-axis and move 3 meters on X
    theta = np.radians(90)

    # Rotation Matrix
    R = np.array([
        [np.cos(theta), 0, np.sin(theta)],
        [0,             1, 0],
        [-np.sin(theta), 0, np.cos(theta)]
    ])

    # Translation Vector
    t = np.array([3.0, 0.0, 0.0])

    # 4x4 Transformation Matrix
    T = np.eye(4)
    T[:3, :3] = R
    T[:3, 3] = t

    print("Transformation Matrix applied to Wall B:\n", T)

    # 3. Apply Transformation
    pcd_B.transform(T)

    # 4. Merge
    combined_pcd = pcd_A + pcd_B

    # 5. Visualize and Save
    o3d.visualization.draw_geometries([combined_pcd])
    o3d.io.write_point_cloud("merged_room.ply", combined_pcd)

    # IMPORTANT: Return T to use for camera transformation
    return T

if __name__ == "__main__":
    T = merge_partial_clouds()
    # Save T for camera transformation
    np.save("transformation_matrix.npy", T)
```

Listing 1: merge_clouds.py

Updating Camera Poses

For the Virtual Tour to work, the cameras associated with Wall B must move exactly as the points did. If you do not update the camera poses, the user will navigate through empty space or inside walls.

The Mathematics

If your SfM pipeline stores cameras as a rotation matrix R_{cam} and translation vector t_{cam}, recall that the **Camera Center** (C) in world coordinates is:

$$C = -R_{cam}^T \cdot t_{cam}$$

Important

Critical Understanding: The vector t_{cam} stored in your camera matrix is **NOT** the camera's position in world space. It is a transformed version. You must:

1. Convert t_{cam} to the actual camera center C
2. Transform both R_{cam} and C
3. Convert the new center back to t_{new_cam}

To transform a camera pose with transformation matrix T (the same matrix used for the point cloud):

1. **Extract Old Camera Center:**

$$C_{old} = -R_{old\_cam}^T \cdot t_{old\_cam}$$

2. **Transform Camera Orientation:**

$$R_{new\_cam} = R_{transform} \cdot R_{old\_cam}$$

3. **Transform Camera Center:**

$$C_{new} = R_{transform} \cdot C_{old} + t_{transform}$$

4. **Recompute t_{cam}:**

$$t_{new\_cam} = -R_{new\_cam} \cdot C_{new}$$

Python Implementation

```python
import numpy as np
import json

def transform_camera_pose(R_old, t_old, T_transform):
    """
    Transform a camera pose by transformation matrix T.

    Args:
        R_old: 3x3 rotation matrix
        t_old: 3x1 translation vector (NOT camera center!)
        T_transform: 4x4 transformation matrix

    Returns:
        R_new: 3x3 transformed rotation matrix
        t_new: 3x1 transformed translation vector
    """
    # Extract transformation components
    R_transform = T_transform[:3, :3]
    t_transform = T_transform[:3, 3]

```

```python
    # Step 1: Get old camera center in world coordinates
    C_old = -R_old.T @ t_old

    # Step 2: Transform rotation
    R_new = R_transform @ R_old

    # Step 3: Transform center
    C_new = R_transform @ C_old + t_transform

    # Step 4: Recompute t (this step is critical!)
    t_new = -R_new @ C_new

    return R_new, t_new

def transform_all_cameras(camera_file, T_transform, output_file):
    """
    Transform all cameras in a JSON file.

    Args:
        camera_file: Path to JSON with camera poses
        T_transform: 4x4 transformation matrix
        output_file: Path to save transformed cameras
    """
    # Load cameras
    with open(camera_file, 'r') as f:
        cameras = json.load(f)

    # Transform each camera
    for cam in cameras['cameras']:
        R_old = np.array(cam['rotation'])  # 3x3 matrix
        t_old = np.array(cam['translation'])  # 3x1 vector

        R_new, t_new = transform_camera_pose(R_old, t_old, T_transform)

        # Update camera data
        cam['rotation'] = R_new.tolist()
        cam['translation'] = t_new.tolist()

        # Optional: also store camera center for convenience
        C_new = -R_new.T @ t_new
        cam['center'] = C_new.tolist()

    # Save transformed cameras
    with open(output_file, 'w') as f:
        json.dump(cameras, f, indent=2)

    print(f"Transformed {len(cameras['cameras'])} cameras")
    print(f"Saved to: {output_file}")

# Example usage
if __name__ == "__main__":
    # Load the same T matrix used for point clouds
    T = np.load("transformation_matrix.npy")

    # Transform Wall B cameras
    transform_all_cameras(
        "cameras_wall_B.json",
        T,
        "cameras_wall_B_transformed.json"
    )
```

Listing 2: transform_cameras.py

Three.js Implementation

Alternatively, you can handle transformations on the web side. If you export your cameras for Wall A and Wall B into separate arrays in your JSON file, you can group them in Three.js.

```javascript
// JavaScript / Three.js Logic

// 1. Create a group for Wall B content
const wallB_Group = new THREE.Group();

// 2. Add points and cameras to this group
wallB_Group.add(wallB_PointCloud);
// Add camera helpers or navigation nodes to this group...

// 3. Apply the transformation matrix directly
// This ensures exact correspondence with Python transformation
const T = new THREE.Matrix4();
T.set(
    R[0], R[1], R[2], t[0],  // First row of T
    R[3], R[4], R[5], t[1],  // Second row of T
    R[6], R[7], R[8], t[2],  // Third row of T
    0,    0,    0,    1      // Homogeneous row
);
wallB_Group.applyMatrix4(T);

// OR, if you prefer position/rotation (less precise):
// wallB_Group.position.set(3, 0, 0);
// wallB_Group.rotation.y = Math.PI / 2;

scene.add(wallB_Group);
```

Listing 3: three.js camera transformation

Important

Note on Three.js Transformations:
Using `applyMatrix4()` ensures your transformation matches exactly what you did in Python.
If you use separate `.position` and `.rotation` properties, Three.js applies transformations in a specific order (Scale → Rotate → Translate) which may not match your mathematical transformation matrix depending on the rotation center.
For Phase 3, either approach is acceptable as long as your scene geometry appears correct.

Complete Workflow Example

Here's a complete workflow from point cloud merging to camera transformation:

```python
import open3d as o3d
import numpy as np
import json

# Step 1: Merge point clouds and save transformation
def merge_and_save():
    pcd_A = o3d.io.read_point_cloud("wall_A.ply")
    pcd_B = o3d.io.read_point_cloud("wall_B.ply")

    # Define transformation (adjust for your scene!)
    theta = np.radians(90)
    R = np.array([
        [np.cos(theta), 0, np.sin(theta)],
        [0, 1, 0],
```

```
        [-np.sin(theta), 0, np.cos(theta)]
    ])
    t = np.array([3.0, 0.0, 0.0])

    T = np.eye(4)
    T[:3, :3] = R
    T[:3, 3] = t

    # Apply and save
    pcd_B.transform(T)
    combined = pcd_A + pcd_B
    o3d.io.write_point_cloud("merged_room.ply", combined)
    np.save("transformation.npy", T)

    return T

# Step 2: Transform cameras
def transform_camera_pose(R_old, t_old, T):
    R_transform = T[:3, :3]
    t_transform = T[:3, 3]
    C_old = -R_old.T @ t_old
    R_new = R_transform @ R_old
    C_new = R_transform @ C_old + t_transform
    t_new = -R_new @ C_new
    return R_new, t_new

def transform_cameras(T):
    with open("cameras_wall_B.json", 'r') as f:
        cameras = json.load(f)

    for cam in cameras['cameras']:
        R_old = np.array(cam['rotation'])
        t_old = np.array(cam['translation'])
        R_new, t_new = transform_camera_pose(R_old, t_old, T)
        cam['rotation'] = R_new.tolist()
        cam['translation'] = t_new.tolist()

    with open("cameras_wall_B_transformed.json", 'w') as f:
        json.dump(cameras, f, indent=2)

# Step 3: Combine all cameras for Three.js
def combine_cameras():
    with open("cameras_wall_A.json", 'r') as f:
        cameras_A = json.load(f)
    with open("cameras_wall_B_transformed.json", 'r') as f:
        cameras_B = json.load(f)

    all_cameras = {
        "cameras": cameras_A['cameras'] + cameras_B['cameras']
    }

    with open("all_cameras.json", 'w') as f:
        json.dump(all_cameras, f, indent=2)

# Run complete workflow
if __name__ == "__main__":
    print("Step 1: Merging point clouds...")
    T = merge_and_save()

    print("\nStep 2: Transforming cameras...")
    transform_cameras(T)

    print("\nStep 3: Combining all cameras...")
```

```
78    combine_cameras()
79
80    print("\ n    Complete! Use merged_room.ply and all_cameras.json")
```

Listing 4: complete_workflow.py

Method 2

Less Recommonded.

If your point clouds do not share features (e.g., opposite walls), you can manually align them using MeshLab.

1. Open MeshLab and import both `wall_A.ply` and `wall_B.ply`.
2. Select the Manipulator Tool (the geometric gizmo icon).
3. **Transform:** Select `wall_B`.
   - Press **T** to Translate (move) the wall to the correct relative position.
   - Press **R** to Rotate the wall to the correct orientation.
4. **Merge:** Once visually aligned, right-click on the layer menu and select **Flatten Visible Layers**.
5. **Export:** Export the merged mesh as `merged_scene.ply`.

Note: This method only merges the points. You must manually estimate the translation and rotation you applied in MeshLab and apply it to your camera positions in Three.js or your JSON data file. This may or may not work in end product.

Final Notes

For the final submission, it is acceptable if the transitions between separate point clouds are slightly disjointed, provided the relative geometry of the room/walls makes sense. The key requirement is that your virtual tour demonstrates:

- A coherent 3D space that represents your captured environment
- Proper camera navigation between viewpoints
- Correct spatial relationships between different sections

Good luck with Phase 3!