ChatGPT

3D Room Reconstruction from Photos – Step-by-Step Implementation Plan

Overview

Inspectors can **reconstruct a room in 3D** from just a handful of photos by using photogrammetry – a process that infers 3D structure from overlapping 2D images. The goal is to produce a **3D mesh model** of the room along with approximate dimensions. This plan outlines an end-to-end pipeline using **free, open-source tools** that run locally in Python. We focus on Structure-from-Motion (SfM) and Multi-View Stereo (MVS) pipelines (e.g. COLMAP, Meshroom) and mention optional AI depth/segmentation tools. We also note commercial alternatives (Instant NeRF, Matterport, Kaarta) for future integration. The output will include a textured 3D mesh (OBJ/PLY format), estimated room dimensions, and optionally labeled surfaces or objects.

Key Steps: We will cover environment setup, image capture guidelines, running SfM and MVS to get a point cloud and mesh, scaling the model for measurements, optional object segmentation, and visualization of results. Each step lists relevant tools, commands, and tips.

Tools and Frameworks to Install

Before starting, set up the following **open-source libraries** (preferably in a Python 3.x environment):

- **COLMAP** – A state-of-the-art photogrammetry toolkit (C++ with Python bindings). It performs SfM and dense reconstruction. Install via pip (`pip install pycolmap`) for Python use or use the COLMAP command-line. COLMAP is robust and accurate, but can be resource-intensive (GPU highly recommended for feature extraction).
- **Meshroom (AliceVision)** – An all-in-one photogrammetry GUI/pipeline. Download the prebuilt package for Windows/Linux from AliceVision's GitHub and use the included `meshroom_batch` for command-line use. Meshroom automates the full pipeline and produces a textured mesh. It requires an NVIDIA GPU for the dense reconstruction step and is relatively user-friendly.
- **OpenMVG + OpenMVS (optional)** – Alternative SfM and MVS libraries. These require building from source and using their CLI tools, but can achieve results similar to COLMAP. Use if you need more customization or integration at library level.
- **MiDaS / DPT (Depth Prediction Models)** – (Optional) Monocular depth estimation networks (Intel's MiDaS, Dense Prediction Transformer, etc.) that predict a depth map from a single image. These can be installed via pip (`pip install git+https://github.com/isl-org/MiDaS.git`) and run on each image to assist in reconstruction or measuring. *Note:* Depth from a single image is *relative* (no absolute scale) [1], but can help fill in geometry or provide per-image depth hints.
- **Detectron2 or Segmentation Models** – (Optional) For object/material labeling, install a segmentation model (e.g. Detectron2 for instance segmentation, or MIT SceneParse for semantic segmentation). These can label 2D images (e.g. walls, floor, furniture) which you can project onto the 3D model in a later step.

- **Open3D** – A Python library for 3D data processing and visualization. Install via `pip install open3d`. We will use Open3D to visualize the point cloud/mesh and to perform simple measurements in Python.
- *(Optional)* **Instant-NGP (Instant NeRF)** – NVIDIA's neural rendering toolkit (C++/CUDA, with some Python API). This is an advanced tool that can **train a NeRF (Neural Radiance Field)** from images to produce a 3D scene extremely fast. If you have a high-end NVIDIA GPU, you can compile Instant-NGP from GitHub. This won't produce a mesh by default (it yields a radiance field), but you can extract point clouds or use techniques like Gaussian Splatting for a point-based model. We mention it for future use; the initial demo will use the more traditional photogrammetry approach.

  Installation Tips: For COLMAP, using the `pycolmap` pip package is easiest. Ensure you have the proper C++ redistributables or compile environment if needed. Meshroom requires an NVIDIA GPU; no installation is needed – just download and unpack. If `meshroom_batch` fails to run, set the `PATH` or `LD_LIBRARY_PATH` as described in Meshroom's docs (the binary is self-contained but needs the AliceVision binaries in its folder). MiDaS and segmentation models will require PyTorch and possibly a CUDA-capable GPU to run efficiently. Open3D works on CPU or GPU and is straightforward via pip. Always test that each tool can be imported or executed before proceeding.

Step 1: Capture 2D Photos of the Room

The reconstruction quality depends heavily on photo quality and coverage. **Take 4–5 photographs** of the room from different angles, following these guidelines:

- **Ensure overlap and diversity:** Every part of the room should appear in at least 3 photos with high overlap. Avoid taking all images from one spot; move around to get different viewpoints (corners, doorway, etc.). Do not just pan in place – change position if possible. Overlap is crucial so that the SfM algorithm can find common features between images.
- **Avoid blur and low texture:** Use good lighting or a tripod/steady hand – images must be sharp (motion blur will hurt feature detection). If walls are blank or texture-less, add identifiable markers (printable patterns or post-its) temporarily on the walls to help algorithms find keypoints. You can remove these markers later from the model or walls.
- **Consistent camera settings:** If possible, use the same camera and settings for all images. Keep exposure consistent to avoid big lighting changes between photos. Enable image metadata (EXIF) – particularly focal length – as photogrammetry tools can read focal length from EXIF to initialize camera intrinsics.
- **Cover all surfaces:** Take photos such that you see each wall and the floor/ceiling from multiple angles. For example, capture one from each corner of the room looking inward, and maybe one from the center. Ensure there is parallax (perspective change) between images – e.g., a chair seen against the wall in one photo and from a shifted angle in another. This parallax is what allows 3D triangulation of points.
- **Avoid moving objects:** The scene should be static. If people or pets are in the room, have them stay still or (preferably) not in the shots. Moving objects will confuse the reconstruction. Also avoid shiny mirrors or windows if you can (cover them), because reflections do not reconstruct properly.

Example: For a 10×12 ft office, you might take one photo in each corner at about chest height, plus one or two from the center. Each photo should see at least two of the same walls/furniture as another photo. This will give sufficient overlap. Make sure features like wall corners, door frames, furniture edges appear in multiple shots.

Step 2: Set Up Project and Organize Images

Organize your image files in a folder (e.g., `project/images`). It's best to use filenames without spaces and ensure all images are JPEG or PNG. Note the image directory path for input to tools. For COLMAP, initialize a new project: you can either use the GUI or directly use the CLI. If using CLI/Python, no need for a GUI project file – the output can go to a workspace folder.

If using **Meshroom**, you can skip explicit project setup – simply prepare the images folder. For **COLMAP (via pycolmap or CLI)**, do the following setup:

- If using `pycolmap`: you can define a workspace path (e.g., `project/`) and ensure an empty database (SQLite) is in place. PyCOLMAP functions will handle creation if not.
- If using COLMAP CLI: create an empty directory `project/` with a subfolder `images/` containing the photos. COLMAP will create a `database.db` in the workspace to store features and matches.

Make sure the working directory or paths are correctly noted. Also, check camera intrinsics: if photos have EXIF, COLMAP will auto-detect focal length. If not, you may need to manually specify camera parameters (focal length, sensor width) – COLMAP can use a camera model database or you can input approximate values.

Step 3: Run Structure-from-Motion (Sparse Reconstruction)

Structure-from-Motion finds feature points in images, matches them across images, and solves for camera positions and a sparse 3D point cloud. You have two main options:

Option A: COLMAP (command-line or Python) – COLMAP's SfM module will produce a sparse point cloud and camera parameters. You can run it step-by-step or use the automatic reconstructor.

- *COLMAP Automatic Reconstruction:* Easiest method – run a single command that does feature extraction, matching, and mapping. For example:

  ```
  colmap automatic_reconstructor \
      --workspace_path /path/to/project \
      --image_path /path/to/project/images \
      --dense 0
  ```

  This will output a sparse model in `project/sparse/` (and skip dense since `--dense 0`). Check the log for how many images were registered (all should register if overlap was good).

- *COLMAP Manual SfM:* If you prefer manual control or using pycolmap:

- **Feature Extraction:** Detect features in each image (e.g. SIFT features). In CLI: `colmap feature_extractor --database_path project/database.db --image_path project/images`. This populates the database with keypoints and descriptors.
- **Feature Matching:** Find matching feature points between image pairs. CLI: `colmap exhaustive_matcher --database_path project/database.db`. (For many images, you might use a sequential matcher or vocab tree to be more efficient, but with 5 images exhaustive is fine).

- **Sparse Mapping:** Reconstruct the scene by solving for camera poses and 3D points. CLI: `colmap mapper --database_path project/database.db --image_path project/images --output_path project/sparse` (this creates a `0` model in `sparse/0`). If you already have known camera intrinsics/poses (not typical in our case), you'd use `point_triangulator` instead. The mapper outputs `cameras.txt`, `images.txt`, `points3D.txt` (or .bin equivalents) describing the calibration and sparse points.

After this, you should have a **sparse point cloud** of the room structure (just a few 3D points where features were found) and camera pose estimates. You can inspect the sparse cloud by opening `project/sparse/0` in the COLMAP GUI or any tool that reads COLMAP outputs. If using pycolmap, you can load the Reconstruction object and examine points.

Option B: Meshroom (AliceVision) – Meshroom performs SfM and MVS in one pipeline automatically. To run SfM with Meshroom:

- Use the Meshroom GUI *or* the `meshroom_batch` CLI. For a quick headless run, execute:

  ```
  meshroom_batch --input /path/to/project/images --output /path/to/
  project/output
  ```

  This will run the full default pipeline (feature extraction, matching, SfM, depth maps, meshing, texturing). Meshroom's pipeline will automatically do all steps unless configured otherwise. You can monitor the console output or the status in the GUI if you opened it. After SfM, Meshroom will produce a sparse point cloud internally (and you can see a preview in the GUI "3D Viewer" with cameras and sparse points once SfM nodes finish). Meshroom also saves the SfM result as an **Alembic (.abc)** file containing cameras and sparse cloud. This is typically in the `MeshroomCache/StructureFromMotion` folder.

Regardless of tool, **verify the SfM result**: Ideally all input images are registered (if some images didn't align, you may need to retake or ensure more overlap). You should see key room features (corners, edges) as sparse points. Don't worry that it's sparse – the dense step comes next.

Tip: Save the camera poses and intrinsics from this step – they'll be needed for scaling and for dense reconstruction. COLMAP's output `images.txt` has camera extrinsics per image, and `cameras.txt` has focal length etc. Meshroom's output can be converted (or you can export a Cameras.sfm or .json). These will be used if we integrate with other tools or for metric measurements.

Step 4: Run Multi-View Stereo (Dense Reconstruction)

With camera poses known, we now compute a **dense point cloud or depth maps** for the scene (MVS). This step uses the images and camera parameters to find dense correspondences for every pixel, producing a **detailed point cloud** of the room surfaces.

Option A: COLMAP Dense – Using COLMAP CLI, continue from the sparse model:

1. **Image Undistortion (optional):** COLMAP can undistort images if camera models have distortion. Run `colmap image_undistorter --image_path project/images --input_path project/sparse/0 --output_path project/dense`. This will create `dense/images` (undistorted) and prepare for dense stereo. (If your camera intrinsics are already pinhole with no distortion, this step can be skipped, but it's harmless to run.)

2. **Depth Map Estimation:** Run COLMAP's PatchMatch stereo:

   ```
   colmap patch_match_stereo --workspace_path project/dense
   ```

   This computes a depth map for each image (in `dense/stereo/depth_maps/`) and normal maps. It uses GPU if available (set `--PatchMatchStereo.geom_consistency true` for better results on indoor scenes with possible repetitive structure).
3. **Point Cloud Fusion:** Fuse the depth maps into a unified point cloud:

   ```
   colmap stereo_fusion --workspace_path project/dense --output_path
   project/dense/fused.ply
   ```

   This generates `fused.ply`, a dense point cloud of the scene. You can open `fused.ply` in a viewer (e.g., Meshlab or Open3D) to see the dense reconstruction. It should look like a "shell" of your room – many points on walls, floor, furniture, etc.
4. **(Optional) Clean-up:** The raw dense cloud may have outliers (floating points). COLMAP doesn't automatically remove all outliers, but you can filter them by adjusting `stereo_fusion` parameters (like `--max_reproj_error`) or later by a statistical filter in Open3D.

Option B: Meshroom Dense – If you ran the full Meshroom pipeline, it has already done dense reconstruction and meshing for you. After the pipeline completes, you'll find a dense point cloud and mesh in the output. Specifically, Meshroom's default pipeline will produce a **textured mesh** (OBJ) directly. Internally, it computes depth maps (similar to COLMAP's PatchMatch) and then does mesh reconstruction + texturing. The outputs can be found in `MeshroomCache`: look for the `Meshing` node output (e.g., `scene.obj`) and `Texturing` output (textured OBJ/MTL). Meshroom also saves a dense point cloud as part of the process (in the DepthMap or Meshing stage caches). To get the dense cloud, you can either: - Generate a cloud from the depth maps (Meshroom's "DepthMapFusion" node produces a dense cloud, often saved as `.ply` in the MeshroomCache). - Or export the mesh's vertices as a point cloud.

Given Meshroom yields a mesh directly, you might skip to Step 5. However, understanding the dense cloud is useful for measurements and debugging.

Quality Check: At this stage, you should have a dense representation of the room. If using COLMAP, open `fused.ply` (e.g., in Open3D: `o3d.io.read_point_cloud("fused.ply")` and visualize). The point cloud should show walls, floor, ceiling as clusters of points. Some areas might be sparse if they lacked texture (e.g. a plain white wall might have holes – you'll address this later by adding artificial texture or using depth priors). If large parts of the room are missing in the dense cloud, you may need more images or check that those images were properly registered.

Step 5: Mesh Generation and Texturing

Next, convert the dense point cloud into a **3D mesh** (triangular surface). A mesh is easier to interpret for measurements (e.g., planar surfaces) and visualization, and can be textured with the original images.

- **COLMAP Poisson Mesher:** COLMAP provides a Poisson surface reconstruction utility. Run:

```
colmap poisson_mesher --input_path project/dense/fused.ply --
output_path project/dense/meshed.ply
```

This uses the Poisson Reconstruction algorithm to produce `meshed.ply` – a mesh (with vertices and faces). It's usually a **watertight** mesh that might fill holes (e.g., it may create a closed surface, sometimes sealing off areas with no data). You can also try `delaunay_mesher` (COLMAP can do Delaunay meshing on depth maps for smaller scenes). The Poisson mesh might be high-poly; you can simplify it later.

- **OpenMVS or MeshLab (alternative meshing):** You could feed the point cloud into OpenMVS's `ReconstructMesh` tool, which often preserves detail. MeshLab also has a Screened Poisson Surface Reconstruction filter you can use on the point cloud (via GUI or PyMeshLab).

- **Meshroom Meshing:** If using Meshroom, this step is already done if the pipeline finished. The output `texturedMesh.obj` (or similarly named) is your mesh. It should come with a texture image (UV mapped). Meshroom's texturing is typically good if lighting was consistent.

- **Texturing in COLMAP:** COLMAP does **not** produce textured meshes by itself in the current version (it produces geometry only). To texture the COLMAP mesh, you can use an external tool. One approach: use OpenMVS's `TextureMesh` on the mesh and input images to bake textures. Another approach: import the mesh and images into Blender or Meshlab and use a texture baking plugin. For a simple demo, texturing can be skipped; we can rely on point colors or do a uniform color for visualization. (If needed, you can also apply the original image as textures by projecting onto the mesh, but that's advanced.)

After meshing, **inspect the mesh**. It should have surfaces for walls, etc. Watch out for any large holes. If ceilings or floors are missing (common if not photographed well), you can fill them in Meshlab (closing holes) or just note the limitation.

If the mesh is extremely dense (millions of triangles) and slow to handle, consider **simplifying** it. For instance, Meshroom's UI has a mesh decimation step; or use Meshlab's Quadric Edge Collapse decimation to reduce poly count while preserving shape. Keep a few hundred thousand triangles for a good balance.

Step 6: Scale and Measure the Room Dimensions

One challenge is that pure photogrammetry yields a model up to an **unknown scale** – it has no real-world units unless we provide a reference. We need to scale the model to get **approximate width/height/depth** of the room:

- **Using a Known Distance:** The most reliable way: have a known measurement in the scene. For example, if you know the height of a door or the length of a specific wall, use that. Say the door height is 2.0 m in reality, and in your reconstructed model the distance between the door's top and bottom points is 1.5 (in arbitrary units). The scale factor would be ~1.33 (2.0/1.5). You'd multiply all coordinates by 1.33 to scale the model to meters. In Open3D, you can pick two points and compute the distance, then compute a scale.
- **Using Camera Metadata:** If the focal length and sensor size were known, tools sometimes recover scale, but typically **SfM has scale ambiguity** even with fixed intrinsics. So do not rely on

the tool to output in, say, meters. It might assume an arbitrary unit where the average camera distance is 1, etc.

- **Assume a dimension:** If no object of known size is present, you can estimate. For instance, standard ceiling height in a US home might be ~8 feet (2.44 m). If the model's ceiling-to-floor difference is, say, 3.5 units in model coordinates, you infer 3.5 units ≈ 2.44 m, so scale accordingly. This is a rough method.
- **Use AR for validation (optional):** As a one-off check, you could use a phone's AR measuring app or LiDAR (if available on an iPhone Pro/Android) to measure one or two distances in the room, then use those to scale the photogrammetry model.

To perform scaling in Python with Open3D, for example:

```python
import numpy as np
import open3d as o3d
pcd = o3d.io.read_point_cloud("fused.ply")
# Suppose we manually identified two points on a known-length wall:
point_ids = [100, 200]  # example indices
pts = np.asarray(pcd.points)
dist = np.linalg.norm(pts[point_ids[0]] - pts[point_ids[1]])
print("Distance in model units:", dist)
scale = KNOWN_REAL_DISTANCE / dist
pcd.scale(scale, center=(0,0,0))
o3d.io.write_point_cloud("fused_scaled.ply", pcd)
```

You can do similar for a mesh (scale the vertex coordinates). After scaling, **width, height, depth** can be found by computing the axis-aligned bounding box of the room points. E.g., `bbox = pcd.get_axis_aligned_bounding_box(); print(bbox.get_extent())` which gives (Δx, Δy, Δz) in whatever unit you scaled to.

For our demo, if no exact reference is available, we will provide dimensions in a relative sense (e.g., "The room measures ~4.5 m by 3 m by 2.5 m (L×W×H)" based on typical assumptions).

Important: State clearly that measurements are approximate. Without ground-truth scale, the numbers might be off by a uniform scale factor. However, relative dimensions (ratios) and shape should be correct. If high precision is needed, consider placing a known-size marker (like a checkerboard or even a letter/A4 paper whose dimensions are known) in one photo – the model can then be scaled using that.

Step 7: (Optional) Object and Material Segmentation

If required, we can label or **segment the reconstructed model** to identify objects (furniture) or materials (walls vs floor). This involves two sub-steps: 2D image segmentation and mapping labels to 3D. This is an advanced step and may be skipped in a simple demo, but here's an approach:

- **Run segmentation on images:** Use a pre-trained model like **Mask R-CNN** (via Detectron2) or **Segment Anything (SAM)** to get masks of objects in each photo. For example, Mask R-CNN (trained on COCO) can detect chairs, tables, person, etc., and give you masks. Similarly, a semantic segmentation model (e.g., MIT Scene Parsing) can classify pixels as wall, floor, ceiling. Run this for each input photo.

- **Project labels into 3D:** Using the camera poses from SfM, you can assign labels to points or mesh faces. For each 3D point, find which images saw that point (COLMAP's `points3D` has image references). Check the corresponding pixel in those images' segmentation outputs. You can do a simple majority vote or nearest neighbor in image space to label the 3D point. For a mesh, you can do a texture transfer: render the mesh into each image and color faces by the image's labels.
- **Color-code or isolate classes:** Once points/faces have labels, you might, for example, color all "wall" points one color, "floor" another, "chair" another, etc., or output separate meshes per category.

This step can get complex, and results depend on segmentation accuracy. If the room has simple geometry, a heuristic approach can be used: e.g., fit a plane to the largest horizontal cluster of points = floor; the vertical large planes = walls; etc. There are research papers on indoor segmentation using 3D point clouds, but those are beyond scope of this demo. Simpler: just identifying furniture via 2D could be useful (e.g., highlight all chairs detected).

Materials: If you want to identify materials (drywall vs tile vs wood floor), that's even more complex – it would require either a trained classifier on patches or metadata. Likely out of scope for now, unless you have a specific model.

In summary, segmentation is optional. It might be practical to at least detect large planar surfaces (walls/floor) from the geometry and label them, since inspectors care about those. One can do plane detection on the mesh or cloud (e.g., RANSAC plane fitting for each wall).

Step 8: Visualization and Export of Results

Finally, we want to visualize the 3D model and possibly export it for others (e.g., create a PDF report or an interactive file).

- **Using Open3D (Python):** You can create a simple visualization:

  ```python
  import open3d as o3d
  mesh = o3d.io.read_triangle_mesh("meshed.ply")
  mesh.compute_vertex_normals()
  o3d.visualization.draw_geometries([mesh])
  ```

  This will open an interactive window where you can inspect the room model. If you only have a point cloud (no mesh), you can visualize that similarly with `o3d.geometry.PointCloud`. Open3D also allows measuring distances: not with a built-in GUI tool, but you could pick points programmatically. For a quick demo, printing the bounding box extents (as mentioned in Step 6) gives room dimensions.

- **MeshLab or CloudCompare:** These GUI tools are great for viewing and cleaning results. You might load `meshed.ply` in MeshLab to check it. CloudCompare can handle point clouds well (including units and measurements, slicing, etc.). For an inspector use-case, CloudCompare could be used to do simple measurements on the point cloud (distance between two points, floor area by fitting a plane, etc.). In our local demo, we'll stick to Open3D or MeshLab for simplicity.

- **Export formats:** Common formats include:

- **OBJ** (with MTL and textures) – good for meshes + texture, widely used (you can import into Blender, Sketchfab, etc.). Meshroom already gives an OBJ. If you have a PLY mesh without texture, you can convert to OBJ (though it won't magically have texture).
- **PLY** – good for point clouds or meshes (can include vertex colors from images if using techniques like vertex coloring). COLMAP's fused PLY has per-vertex color (averaged from images). If you open fused.ply in MeshLab, you'll see a colored point cloud thanks to this.
- **GLB/GLTF** – an efficient web-friendly 3D format. You can convert an OBJ+texture to GLB using tools like Blender or the `obj2gltf` converter. This could be useful if the client wants to view the model in a web browser or embed in a report.

- **PDF report:** For a simple report, you might take screenshots of the model (e.g., a top-down view showing dimensions, a perspective view of the textured mesh) and include the measured dimensions. There are also ways to embed 3D models in PDFs (as U3D), but that's advanced and not widely supported.

- **Example Output Check:** As a sanity check, you could compare the reconstruction with known room dimensions. If the model is scaled correctly, try measuring one side using the model: e.g., pick two opposite corners of the floor in the point cloud and compute the distance – does it match the actual room width? This is a validation step if ground truth is known. If off, adjust scale or note the discrepancy.

Lastly, consider that inspectors might want the **floor plan** or **cross-sections**. While not explicitly asked, note that once you have a scaled 3D model, you can derive a floor plan by projecting the points to a plane. There are tools to automatically extract walls and create CAD models (Matterport and others do this). Open-source alternatives include `FloorNet` or simply slicing the point cloud at a certain height and exporting points as 2D. Given our focus, we'll mention that a floor plan can be extracted but leave details for later if needed.

Step 9: (Optional) Explore Advanced/Commercial Solutions

(This is not part of the immediate demo but for evaluation purposes.) As the client is interested in feasibility, you might also consider these alternatives:

- **NVIDIA Instant NeRF (Instant-NGP):** This approach uses neural networks to reconstruct the scene faster than classical methods. With ~5 images, Instant NeRF can **interpolate the 3D scene** in seconds, producing a view-dependent representation. It's great for visualization (you can render new angles smoothly) and can handle low-texture regions by learning implicit surfaces. However, it does *not* directly give you real scale or a mesh – it gives a radiance field. One can sample points from it or use recent techniques (like Gaussian Splatting) to convert to a point cloud. If you have the hardware, you could try Instant-NGP: you would need camera poses (which you have from COLMAP) and then train the NeRF model. The output can be visualized with the Instant-NGP viewer. This might be more complex to integrate for now, but it's a promising avenue (NVIDIA's research even got Instant NeRF to be named one of the best inventions of 2022). Keep in mind NeRF outputs are not as straightforward to measure – you'd still need to extract a mesh or points and scale them.

- **Matterport:** Matterport offers a cloud-based solution and hardware (cameras with depth) that produces high-quality interior models and floor plans. They do have an API/SDK for uploading images and retrieving a 3D tour or data. For example, their services can generate a textured mesh and also a floor plan with dimensions. Using Matterport would likely involve taking a *lot* of photos (or a panoramic camera) and sending to their cloud – which might be outside the “local demo” requirement. However, if the client eventually wants a robust, supported solution, Matterport is a gold standard in real estate scanning. One could imagine integrating by using our images as input to Matterport’s platform (though they optimize for their own hardware and app). For now, we note it as an option if they require enterprise-level deliverables.

- **Kaarta:** Kaarta is known for mobile mapping devices (often LiDAR + vision) for real-time scanning. It’s more of a hardware solution – e.g., Kaarta Stencil 2 is a device that you walk through the space to generate an instantaneous 3D map. If the inspectors eventually want **real-time** on-site reconstruction, a SLAM-based approach like Kaarta’s could be considered. There isn’t a readily available Python SDK for Kaarta’s proprietary system (it’s an all-in-one solution), but data from such a device could be imported into our pipeline. In summary, Kaarta’s benefit is speed – no need to take separate photos and process offline, it gives you a point cloud on the fly. The trade-off is cost and complexity of hardware.

- **Others:** There are other commercial APIs and software like **Pix4D**, **Agisoft Metashape**, and **Polycam** (mobile app) that do photogrammetry. Pix4D is more drone-mapping focused but can do indoors; Metashape is a popular photogrammetry software with Python scripting (license required). Polycam (for iPhone/iPad) uses ARKit+photogrammetry to quickly scan a room (leveraging the device LiDAR on newer models) – it might even be a quick way for inspectors to scan via phone. For our local demo, we stick to free tools, but these are in the ecosystem.

Step 10: Pipeline Summary and Example Run

Let’s summarize the pipeline with a hypothetical example to ensure everything is clear:

1. **Images in**: 5 photos of a sample living room (taken per guidelines – overlapping views from various corners). Each photo ~12MP, with EXIF.
2. **Run COLMAP SfM**: Using pycolmap or CLI, features are extracted and matched. We get ~5,000 sparse points (feature points like corners of frames, edges of furniture) and camera poses. All 5 images registered successfully.
3. **Run COLMAP MVS**: Depth maps are computed. Fused point cloud has ~1 million points, clearly outlining walls, floor, sofa, table in the room.
4. **Mesh**: Poisson mesher creates a mesh with ~500k faces. The mesh covers walls and furniture surfaces. Some holes are present on the plain white ceiling (due to few features), but it’s mostly complete.
5. **Scale**: We know the ceiling height was 2.5 m. Measuring the vertical span in the model between floor and ceiling points gives 2.47 units – close, but we apply a scale of 1.012 to be exact. Now dimensions read: length ~4.0 m, width ~3.2 m, height ~2.5 m (so a roughly 13 ft by 10.5 ft room with 8.2 ft ceiling – plausible).
6. **Segmentation (optional)**: We run Mask R-CNN on one image; it detects a “couch” and “table”. We manually label those points in the cloud for demonstration (coloring couch points red).
7. **Visualization**: Using Open3D, we render the textured mesh (if we textured it) or a colored point cloud. We see a 3D view of the room – one can orbit around and see furniture placement. The bounding box shows ~4.0×3.2×2.5 m as noted. We screenshot this view. We also output the mesh as `room_model.obj`.
8. **Export**: Provide the OBJ and a PDF report. The report can include: an overhead view screenshot with approximate dimensions annotated (width/length arrows), and perspective views of the 3D

model. If segmentation was done, we could list “Detected objects: Couch (approx 2 m long), Table, etc.” and perhaps highlight their locations.

The developer following this plan can use the above as a template. Each step can be adjusted based on actual data and tools at hand (for instance, if Meshroom is used instead of COLMAP, skip the separate SfM/MVS commands and just run Meshroom, then scale the result).

Strengths and Limitations of the Approach

- **Open-Source Pipeline Strengths:** No licensing costs; full control over the data (runs locally). COLMAP and Meshroom are proven in accuracy – they can reconstruct with few cm accuracy given good images. The output mesh and point cloud are standard and can be integrated into other systems. This approach does *not* require special hardware (just a normal camera).
- **Challenges:** Photogrammetry might struggle with **low-texture surfaces** (blank walls, uniform carpet) – which are common in home interiors. In our pipeline, adding markers or using depth hints (MiDaS) can alleviate this. Also, with only 4-5 images, the reconstruction might miss some areas or have less detail. Usually, more images (10–20) yield better coverage. So, if feasible, capture more photos. However, we limited to 4–5 to test feasibility; the quality will be “simple but decent” as requested. Another limitation is scale ambiguity – we addressed this via manual scaling. One must be careful to do that step if measurements are needed. Lastly, processing time: with a good GPU, this pipeline might take a few minutes (feature extraction and dense steps are the longest). On CPU it can be significantly slower. For a demo, this is fine, but for daily inspector use, one might optimize or use GPU servers.
- **Depth AI (MiDaS) Strengths:** Can fill in geometry for feature-poor areas and work with a **single image** [2] . In an interactive app, one could imagine using MiDaS to quickly get a rough depth map of each photo and combine them. However, MiDaS gives *relative* depth – aligning multiple such depth maps to each other is essentially doing SfM again (with less precision). Also, its scale is arbitrary [1] . So depth AI is a good augmentation tool but not a complete solution here. We highlight it for cases where photogrammetry fails on a particular surface; one could project MiDaS depth for that surface as a patch.
- **Commercial Tools:** Instant NeRF’s strength is speed and photorealistic renderings – great for visualization, but it’s currently harder to get exact measurements out of a NeRF. Matterport and similar solutions are very user-friendly and produce excellent results (including measurements and object tagging automatically via AI), but involve cloud processing and cost. They might be considered if scaling up deployment – e.g., an inspector could upload photos to Matterport Cloud and get a model + auto-generated report of room dimensions and features. For now, our open-source route keeps everything local and customizable.

Conclusion and Next Steps

By following this plan, a developer can **implement a local Python demo** that takes a handful of room photos and outputs a 3D model with measurements. Start with the photogrammetry pipeline (SfM + MVS) using COLMAP or Meshroom to get the core 3D mesh [3] . Then apply scaling for real-world dimensions. Optionally enhance the model with segmentation or integration of depth networks if needed. The result can be visualized in Python or exported to common formats for stakeholders to review.

Example deliverables could be: an `.OBJ` model of the room, a set of images (screenshots) showing the 3D reconstruction from different angles with key dimensions annotated, and perhaps a simple text summary (“Room approx 4.0 m × 3.2 m × 2.5 m; objects identified: sofa, table.”).

With this foundation in place, further improvements can be explored, such as increasing image count for better fidelity, using **Gaussian Splatting** or NeRF for faster capture in the future (these can complement the pipeline – e.g., using sparse SfM from COLMAP as input to Gaussian Splatting as noted in a recent SIGGRAPH 2023 method), or building a UI for inspectors to load images and automatically get a report. The open-source nature of the tools ensures flexibility to iterate on this prototype without hefty investment.

---

[1] [2] Scade.pro
https://www.scade.pro/processors/cjwbw-midas

[3] Tutorial: Meshroom for Beginners — Meshroom v2023.1.0 documentation
https://meshroom-manual.readthedocs.io/en/latest/tutorials/sketchfab/sketchfab.html