Reality Capture and Immersion: A Workflow for 3D Scanning to VR in Unity

Mathias Dalla Palma

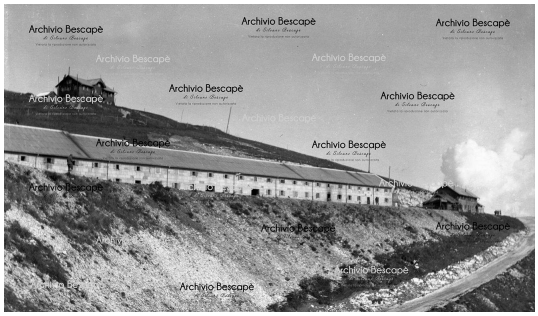
February 14, 2025

Abstract

This project explores a workflow for reconstructing real-world structures in a virtual environment using photogrammetry and camera pose estimation. The process begins with capturing a set of photographs of a building, which are processed in 3D Zephyr to generate a point cloud, mesh, and textures while also extracting the extrinsic parameters of each image. The reconstructed model is then imported into Unity alongside a selected photograph, which is placed and aligned using its extrinsic parameters to create the illusion of a real-world image existing within the virtual space. The final step involves implementing camera pose estimation by applying feature matching techniques to retrieve the extrinsic parameters of a newly introduced image. This project also explores different feature matching algorithms to evaluate their effectiveness in pose estimation. Finally, it will be shown how the model can be integrated into a simple augmented reality system, allowing users to interact with it in a straightforward yet effective way to explore its significance and potential applications.

1 Introduction

Cultural heritage preservation has greatly benefited from advancements in 3D scanning and virtual reality (VR) technologies. Historical sites and monuments, often at risk of deterioration, can be digitally archived and experienced in immersive virtual environments. This project focuses on the 3D digitization of the Forte Lisser barracks, a secondary structure located near the main fort in the Asiago Plateau, northern Italy.



A bit of history
Built during World War I, the barracks served as housing for soldiers and a stable for horses, providing essential logistical support to the operations of Forte Lisser.
These barracks were strategically positioned to accommodate troops stationed at the fort while ensuring a nearby supply of mounted units for reconnaissance and transport.
Although time and weather have taken their toll on the structure, it remains an important piece of military history, offering valuable insight into the daily lives of soldiers during the war.

The goal of this project is to generate a VR representation of Forte Lisser barracks using photogrammetry. The process begins with capturing a set of photographs of the building, which are then processed in 3D Zephyr to reconstruct a detailed 3D model. Alongside the model, Zephyr extracts the extrinsic parameters of each photograph, describing the camera's position and orientation relative to the scene.

The generated model and a selected photograph are imported into Unity, where the photo is placed and aligned with the 3D scene using its precomputed extrinsic parameters. This alignment creates the illusion that the photograph exists physically within the virtual space, allowing users to view the fort through both a real-world image and a 3D reconstruction.

A key objective of this project is to implement a semi-automated pipeline that minimizes direct coding interaction. The workflow is designed to be as user-friendly as possible, allowing the transition from real-world photographs to a fully immersive VR scene with minimal manual intervention.

The final stage of the project involves camera pose estimation, a technique used to determine the position and orientation of a new photograph relative to the 3D model. This is achieved by combining feature matching algorithms with Fiore's method for camera pose estimation. A comparison of different feature matching techniques is conducted to evaluate their performance and effectiveness in this context.

By focusing on cultural heritage digitization and automation, this project not only preserves an important historical site but also establishes a practical and efficient workflow for integrating real-world imagery into VR. The results demonstrate how photogrammetry and computer vision techniques can be leveraged to create immersive and historically significant virtual experiences with minimal technical barriers.

2 Image Capturing

The first step in the 3D reconstruction process is to capture a high-quality set of images to ensure accurate photogrammetric reconstruction. Several factors influence the effectiveness of this step, including lighting conditions, camera stability, and coverage of the structure.

2.1 Lighting and Environmental Considerations

For optimal results, photos should be taken under conditions that minimize strong shadows and overexposure. Ideally, evenly diffused natural light is preferred, such as on overcast days or during golden hours (early morning or late afternoon), to avoid harsh contrasts. However, given the seasonal conditions at the time of this project, the presence of snow on the ground was unavoidable. While snow can sometimes introduce issues such as reflections or loss of texture detail, in this case, it is not expected to significantly affect the reconstruction results, as the focus remains on the building's structure.

2.2 Camera and Shooting Strategy

The main dataset consists of 80 photographs, captured using a Nikon Coolpix L830 camera at a resolution of 3264×1836 pixels. To ensure a consistent and accurate reconstruction, special attention was given to maintaining the same central point in each image, hoped to help Zephyr align the photos more effectively. The photos were taken while following a semicircular path around the front side of the building, allowing for optimal coverage of its facade.

Figure 1: One of the captured images

This portion of the structure was chosen specifically because it is better preserved, making it more suitable for generating a detailed and accurate 3D model.

2.3 Drone Video Capture

In addition to ground-level photographs, an aerial video was captured using a consumer-grade drone to provide an overhead perspective of the building. The video serves as a complementary dataset, offering a broader understanding of the structure's layout. However, as will be analyzed later in the documentation, the reconstruction quality from the drone footage is not as satisfactory as the photo-based dataset. This is likely due to the lower camera quality, motion blur, and lack of precise image control compared to manually taken photographs. Despite this, the drone footage remains a useful reference for understanding the overall structure.

Figure 2: Enter Caption

3 Model Reconstruction in 3D Zephyr

Once the image dataset was captured, the next step was to process the photographs in 3D Zephyr to generate a 3D model. Zephyr provides a streamlined wizard-based workflow, which was used to facilitate the reconstruction process.

3.1 Reconstruction Process

Using the Zephyr wizard, the images were processed through a straightforward and user-friendly procedure that automatically handled key steps such as feature detection, camera alignment, dense reconstruction, and texture generation, resulting in the final 3D model.

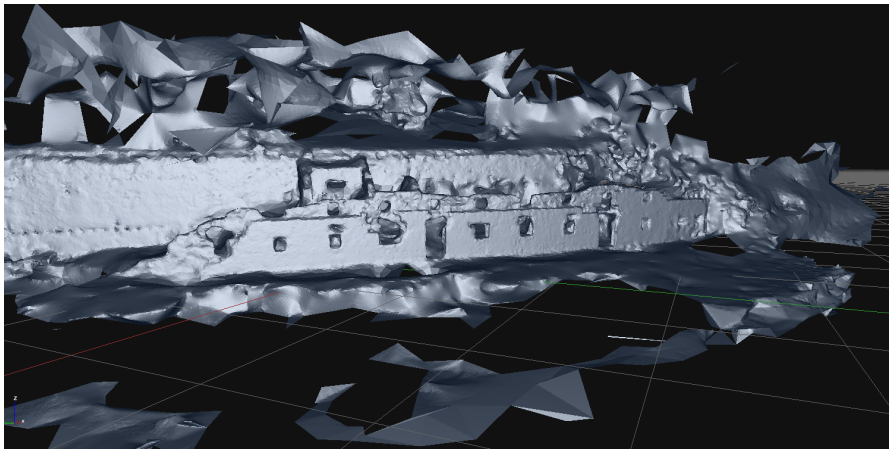
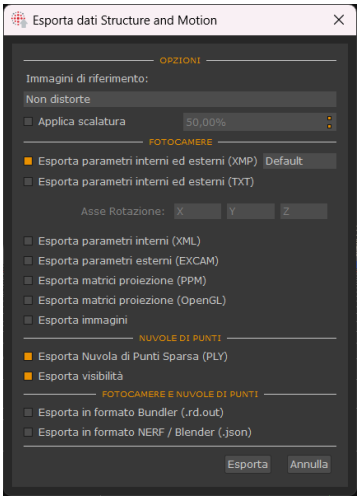


Figure 3: Result of the reconstruction

After the reconstruction, the following elements were exported:



- 3D model with textures (obj):A textured mesh representation of the reconstructed scene.
- Dense cloud point (ply): A high-resolution collection of 3D points representing the scanned scene.
- Extrinsic parameters for each image (xmp): Camera pose data, including position and orientation, stored in an XMP file.
- Visibility file (txt): A file containing 2D-to-3D correspondences, specifying which 3D points are visible in each image.

3.1.1 Model Reconstruction from Drone Footage

To further evaluate the effectiveness of different data sources, a drone video of the building was used instead of individual photos. The same Zephyr wizard workflow and maximum settings were applied, but instead of importing photos manually, the "Import images from video" option was selected. This approach introduced new parameters:

- Sampling Time: Determines how many frames per second are extracted as images.
- Filter Similarity: Discards frames that are too similar to avoid redundant data.

Despite multiple attempts with different parameter combinations, none of the reconstructions achieved a model quality comparable to the main dataset. The final 3D model from the drone footage displayed significant inaccuracies, including wobbly surfaces, even in areas that should be flat and poor structure definition, resulting in a less realistic reconstruction.

Figure 4: Real captured image and the corresponding virtual view

As a result, this method was ultimately discarded, but the output is included for comparison to illustrate the limitations of using low-quality drone footage for photogrammetry.

3.2 Model Improvement

Although the reconstruction process was largely successful, some artifacts and missing areas were present in the generated model.

Removing the Sky Artifacts

One issue encountered was that the blue sky in the background was mistakenly interpreted as part of the structure. To resolve this: Zephyr's selection tool was used to isolate and remove the sky portions

of the model. In particular a tool for selecting faces by color was used resulting very effective because no other blue elements were present in the dataset, making the selection highly precise.

Fixing Holes in the Mesh

Another issue was the presence of holes in areas where there should have been a solid surface. Zephyr does not provide a built-in tool for automatic hole filling, so Blender was used to correct the mesh.

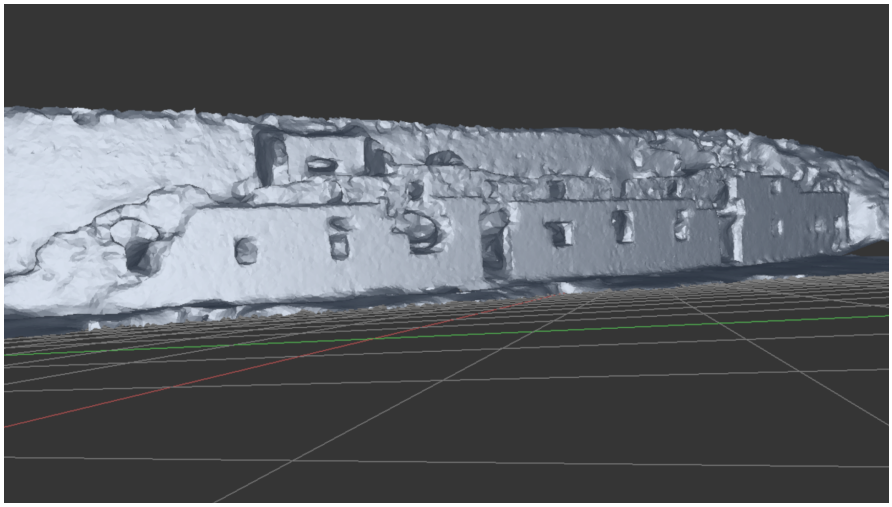


Figure 5: Final model after the improvements

4 Unity Scene

The next step involves setting up a Unity scene to render the 3D model and align one of the photos used in the reconstruction. The objective is to position a Unity camera with the same intrinsic and extrinsic parameters as the real camera that captured the original dataset image. This ensures that the photo appears seamlessly integrated into the virtual environment.

4.1 Setting-up the scene

To begin, the following assets were imported into Unity as game assets:

- The 3D model exported from Zephyr.
- A photo from the dataset.
- The XMP metadata file generated by Zephyr, which contains the camera's extrinsic and intrinsic parameters.

The first issues encountered was that Zephyr uses a right-handed coordinate system, while Unity operates in a left-handed system. As a result, the imported model appeared misaligned. To correct this a rotation transformation had to be applied to properly orient the model within Unity's world space. The second issues was that the XMP file encodes the camera's rotation as a 3×3 rotation matrix, while Unity requires Euler angles (X, Y, Z). To ensure the camera's orientation matched its real-world counterpart, the rotation matrix had to be converted into Euler angles before being applied to the Unity camera.

To streamline the process of placing the model, aligning the photo, and configuring the Unity camera, a C# script was developed. This script ensures that the 3D model, reconstructed photo, and camera are correctly positioned using the extrinsic and intrinsic parameters stored in the XMP file. The script takes as input the three files previously imported as game asset. To make the process accessible and user-friendly, the script allows users to drag and drop these assets from Unity's Asset Menu into the Inspector panel of a designated GameObject.

4.1.1 C# Script

The script follows a modular approach for better maintainability and clarity. The three input are declared as public fields so they can be assigned via drag-and-drop in the Unity Inspector. The execution happens inside the Update() function, where a conditional check ensures that the script runs only once when triggered by setting run = true. To keep the code organized and easy to modify, the main steps of the workflow are divided into separate functions, each handling a specific part of the process.

```
//imports ...

[ExecuteInEditMode]
public class SetUpSceneScript : MonoBehaviour
{
    public GameObject model;
    public Sprite image;
    public TextAsset xmp;
    public bool run;

    public static Vector3   private void PlaceModel(){ ... }

    private void PlaceandSetupCamera(CameraParameter cameraParameter){ ... }
    private CameraParameter GetCameraParameterFromXmp(){ ... }
    public static Vector3 GetEulerAnglesFromRotationMatrix(float3x3 m){ ... }

    private void PlaceImage(){ ... }

    void Update()
    {
        if(run){
            //create a new model from the mesh
            PlaceModel();

            //create a new camera
            CameraParameter data = GetCameraParameterFromXmp();
            PlaceadnSetupCamera(data);

            //create canvas with the image
            PlaceImage();

            //prevent looping
            run=false:
        }
    }
}
```

The workflow begins with the correct placement of the 3D model in the Unity scene.The function instantiates a GameObject of the model in the scene, applies the correct rotation to align it with Unity's coordinate system, and adds a Mesh Collider to ensure proper physical interactions.

```
private void PlaceModel(){
    GameObject modelObject = Instantiate(model);
    modelObject.name = "Model";
    modelObject.transform.rotation = Quaternion.Euler(-90, 180, 0);
    modelObject.AddComponent<MeshCollider>();
    modelObject.GetComponent<MeshCollider>().sharedMesh=
        modelObject.GetComponentInChildren<MeshFilter>().sharedMesh;
}
```

Next, the GetCameraParameterFromXmp() function parses the XMP file, extracting the extrinsic

parameters (camera position and orientation) and intrinsic parameters (focal length, sensor size, etc.), encapsulating them into a simple class for easier access and use in the setup. Before placing the camera, it is necessary to convert the rotation matrix (3×3) into Euler angles (X, Y, Z), as Unity does not accept rotation matrices directly.

```
public static Vector3 GetEulerAnglesFromRotationMatrix(float3x3 m)
{
    // Adjust the rotation matrix to the Unity ref
    float m00 = m[2][2];
    float m01 = m[0][2];
    float m02 = m[1][2];
    float m10 = m[2][0];
    float m11 = m[0][0];
    float m12 = m[1][0];
    float m20 = m[2][1];
    float m21 = m[0][1];
    float m22 = m[1][1];

    float sy = Mathf.Sqrt(m21* m21 + m22 * m22);

    bool singular = sy < 1e-6; // If sy is near-zero, use a different convention

    float x, y, z;
    if (!singular)
    {
        x = Mathf.Atan2(-m20, sy);
        y = Mathf.Atan2(m10, m00);
        z = Mathf.Atan2(m21, m22);
    }
    else
    {
        x = Mathf.Atan2(-m20, sy);
        y = 0;
        z = Mathf.Atan2(-m12, m11);
    }

    x = Mathf.Rad2Deg * x;
    y = Mathf.Rad2Deg * y;
    z = Mathf.Rad2Deg * z;

    return new Vector3(x, y, z);
}
```

This code follows the standard procedure for extracting Euler angles from a rotation matrix while also accounting for the left-right coordinate system transformation.

Mathematically

The original rotation matrix R is structured as follows:

$$R = \begin{bmatrix} r_{00} & r_{01} & r_{02} \\ r_{10} & r_{11} & r_{12} \\ r_{20} & r_{21} & r_{22} \end{bmatrix}$$

To adjust the matrix to Unity's reference system, we perform a permutation of its elements:

$$R' = \begin{bmatrix} r_{22} & r_{02} & r_{12} \\ r_{20} & r_{00} & r_{10} \\ r_{21} & r_{01} & r_{11} \end{bmatrix}$$

Then the Euler angles $(\theta_x, \theta_y, \theta_z)$ of R are computed as follows:

$$\theta_X = \arctan 2\left(-r_{21}, \sqrt{r_{01}^2 + r_{11}^2}\right) \tag{1}$$

$$\theta_Y = \arctan 2(r_{20}, r_{22}) \tag{2}$$

$$\theta_Z = \arctan 2(r_{01}, r_{11}) \tag{3}$$

If a singularity case is detected, gimbal lock occurs, and the computation simplifies to:

$$\theta_X = \arctan 2\left(-r_{21}, \sqrt{r_{01}^2 + r_{11}^2}\right) \tag{4}$$

$$\theta_Y = 0 \tag{5}$$

$$\theta_Z = \arctan 2(-r_{10}, r_{00}) \tag{6}$$

Once the Euler angles are computed, the script instantiates a Unity camera and configures its position, rotation, and intrinsic parameters to match the real-world camera.

```
private void PlaceandSetupCamera(CameraParameter cameraParameter){
    GameObject cameraObject = new GameObject("Camera");
    cameraObject.AddComponent<Camera>();
    Camera camera = cameraObject.GetComponent<Camera>();

    camera.name = "Camera";
    camera.transform.position = cameraParameter.Position;
    camera.transform.rotation = cameraParameter.Rotation;

    camera.usePhysicalProperties = true;
    camera.focalLength = cameraParameter.FocalLenght;
    camera.sensorSize = cameraParameter.SensorSize;
    camera.lensShift = cameraParameter.LensShift;
}
```

Finally the original photo is rendered on a UI Canvas, which is made a child GameObject of the camera. This setup ensures that the image always stays in front of the camera's viewpoint, giving the impression that the photo is physically present in the VR environment.

```
private void PlaceImage(){
    //create a new canvas
    GameObject canvasObject = new GameObject("Canvas");
    canvasObject.AddComponent<Canvas>();
    canvasObject.AddComponent<CanvasScaler>();

    canvasObject.GetComponent<CanvasScaler>().uiScaleMode =
        CanvasScaler.ScaleMode.ScaleWithScreenSize;

    Canvas canvas = canvasObject.GetComponent<Canvas>();

    canvas.name="Canvas2";
    canvas.transform.SetParent(cameraObject.transform,false);

    canvas.worldCamera = camera;
    canvas.renderMode = RenderMode.ScreenSpaceCamera;


    //create a new image
    GameObject imageObject = new GameObject("Image");
    imageObject.AddComponent<Image>();
    Image rawImage = imageObject.GetComponent<Image>();
```

```
        rawImage.name = "Image2";
        rawImage.transform.SetParent(canvasObject.transform,false);
        rawImage.sprite = image;
        rawImage.preserveAspect = true;

        RectTransform rectTransform = rawImage.GetComponent<RectTransform>();
        rectTransform.anchorMin = new Vector2(0,0);
        rectTransform.anchorMax = new Vector2(1,1);
        rectTransform.offsetMin = new Vector2(rectTransform.offsetMin.x, 0);
        rectTransform.offsetMax = new Vector2(rectTransform.offsetMax.x, 0);
        rectTransform.sizeDelta = new Vector2(0,0);
    }
```

After executing the script, the results are as shown in the next picture. The elements are correctly instantiated with the proper parameters, and in the scene, the model, the camera, and the aligned image are visible.

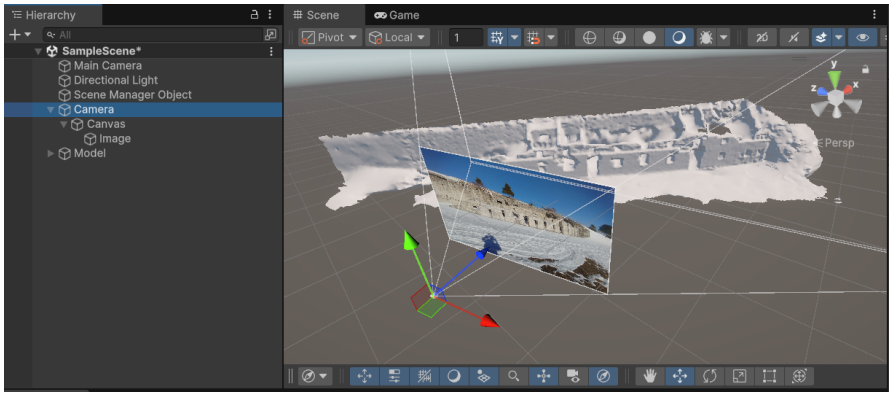


Figure 6: Enter Caption

4.1.2 Alternative Alignment Method: Fiore's Mehtod

An alternative approach to computing the extrinsic parameters of the camera is the Fiore's Method. Unlike the previous method, which directly retrieves the extrinsic parameters from Zephyr's XMP files, this approach uses 3D point cloud data and 2D-3D correspondences to compute the camera's rotation and translation from scratch.

Mathematical Foundation

The method relies on Perspective-n-Point (PnP) estimation, solving for the camera's extrinsic matrix $[R|\vec{t}]$, where:

- R is the rotation matrix (3×3)
- $\vec{t}$ is the translation vector (3×1)

Given a set of known 3D points $\vec{P_i} = (X_i, Y_i, Z_i)$ and their corresponding 2D projections $\vec{p_i} = (x_i, y_i)$, the goal is to find the best transformation that satisfies the projection equation:

$$s\vec{p_i} = K[R|\vec{t}]\vec{P_i}$$

where:

- K is the camera intrinsic matrix, containing focal length and principal point. It is retrieved from the metadata of the image.
- s is a scaling factor.

To compute R and $\vec{t}$, the method uses the visibility file from Zephyr, which provides precomputed 2D-3D correspondences. A MATLAB script was used to process this data and compute the camera extrinsic parameters. The script was modified to output an XMP file, ensuring compatibility with the Zephyr format and seamless integration into the Unity pipeline.

The results obtained using this method were convincing and coherent with Zephyr's ground truth calculations. The small mathematical differences observed did not significantly affect the perceived alignment in the Unity scene, confirming the method's practical effectiveness.

4.2 Final Scene and Interaction Test

The final Unity scene successfully integrates the 3D model, aligned photo, and camera setup. The result is visually convincing, as the photo appears to be physically present in the virtual environment.

Figure 7: Unity camera prospective

To further evaluate the illusion of interaction, a simple script was used to launch balls at the model. When the balls collide with the surface, the alignment between the photo and the 3D geometry makes it seem as if the image itself is reacting to virtual objects, reinforcing the effect of depth and realism in the scene.

Figure 8: Virtual object interacting with the photo

This test demonstrates that the technique effectively blends real-world imagery with interactive virtual content, making it a promising approach for VR cultural heritage applications.

5 Camera Pose Estimation Using Feature Matching

This step aims to estimate the pose of a camera that captures a new image of the subject by leveraging feature matching and Fiore's method. Unlike the dataset images used in the reconstruction process, the new image (target image) does not have precomputed intrinsic or extrinsic parameters from Zephyr. The goal is to determine its position and orientation as if it were taken at a later time, separate from the original scan.

5.1 Methodology

The camera pose estimation process follows these steps:

1. **Reference Image Selection**

   To begin, a reference image is chosen from the dataset. This image serves as a reliable starting point because the correspondences between its 2D points and 3D points of the model are known from the reconstruction process in Zephyr. By using this reference, a connection can be established between the new target image and the 3D model.

2. **Feature Matching**

   Once the reference image is selected, keypoints are extracted from both the reference image and the target image. These keypoints represent distinctive visual features that can be consistently identified across different views. The feature matching algorithm find correspondences between the two images, linking points in the target image to their counterparts in the reference image.

3. **Filtering Correspondences Using the Visibility File**

   Not all matched points are useful for our purposes. To refine the selection, Zephyr's visibility file was used; this file contains a precomputed mapping between 2D points in the reference image and their corresponding 3D points in the model. By applying a proximity threshold, irrelevant matches are filtered out and only those 2D points that are close enough to the known visibility data are retained.

4. **Mapping 2D Points to 3D Model Coordinates**

   After filtering, the 2D points that originally associated with the reference image are substituted with the newly matched points from the target image. This effectively establishes a new set of 2D-3D correspondences, where each selected point in the target image is now linked to a known 3D coordinate in the model.

5. **Pose Calculation Using Fiore's Method**

   With the 2D-3D correspondences finalized, the camera's extrinsic parameters using Fiore's method can be computed. This algorithm estimates the optimal rotation and translation that align the target image's keypoints with the corresponding 3D points, ultimately determining the pose of the camera as if it had been part of the original dataset.

By following this procedure, we can effectively estimate the camera pose of an image taken at a different time than the original dataset, allowing for further integration of new perspectives into the virtual environment.

5.2 SIFT-Based Approach

As a first attempt, SIFT was used as the feature matching algorithm. Through multiple trials and adjustments of parameters such as the distance threshold between keypoints and visibility points, and the SIFT peak threshold, the results progressively improved. While fine-tuning these parameters required some experimentation, the final outcome proved to be effective. In the final result the estimated pose was slightly off compared to the ground truth but the deviation remained small enough that it did not impact the perceived alignment in the Unity scene.

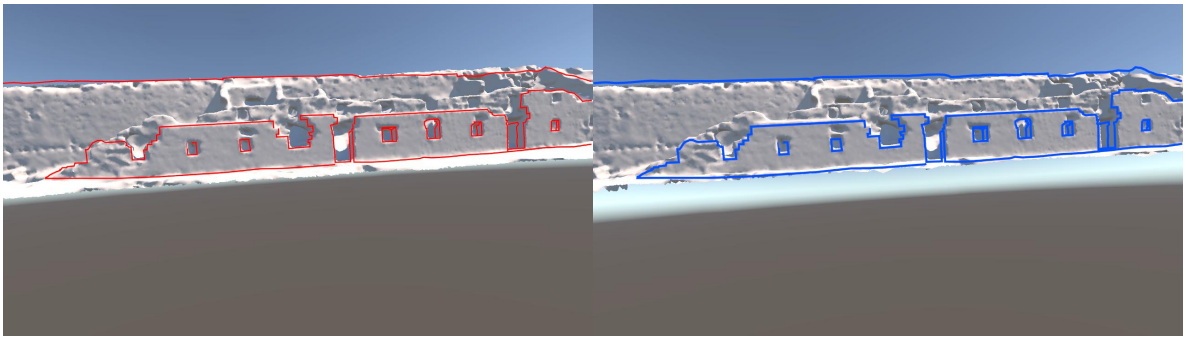


Figure 9: Comparison of Ground Truth (Red) and Calculated Result (Blue).

This image demonstrates the effectiveness of the final result by comparing the ground truth alignment with the calculated one. A visual representation of the original image highlights the building's silhouette, allowing a clear comparison. The building's silhouette remains consistently aligned with the model in both cases resulting visually accurate and satisfying. To quantify the alignment accuracy, the calculated error in translation has a magnitude of 2.54, while the rotational error is 6.03. In real-world terms, this corresponds to a translation shift of approximately 70 cm, with rotational deviations of approximately 5 degrees in pitch, 0.4 degrees in roll, and 0.1 degrees in yaw. Given the distance from the subject and its overall dimensions, these discrepancies are small enough to not significantly impact the visual quality of the alignment.

5.3 LoFTR-Based Approach

LoFTR (Local Feature TRansformer) is a detector-free framework that establishes dense pixel-wise correspondences between images using transformers. Unlike traditional methods that rely on keypoint detection and description, LoFTR directly predicts matches across images, making it effective in scenarios with low-texture regions, motion blur, or repetitive patterns.

5.3.1 LoFTR

The following sections will provide a detailed discussion of the specific algorithm used in this project, referencing the original authors and their contributions to the field [1].

The image processing pipeline described in the document can be briefly summarized in 4 key steps:

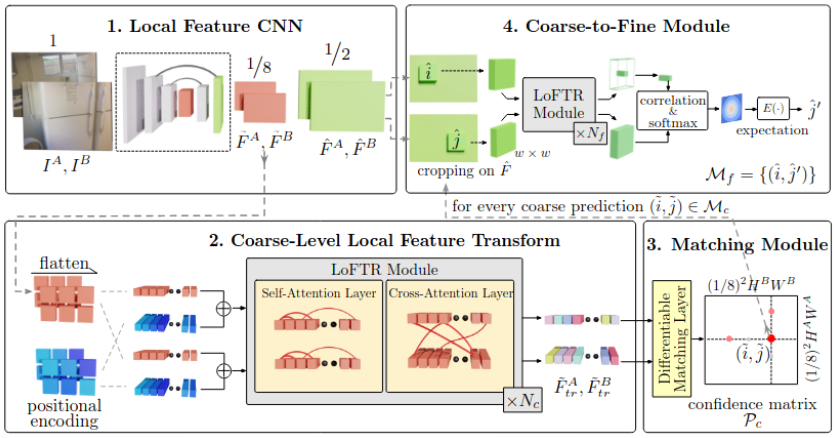


Figure 10: Overview of LoFTR pipeline

1. **Local Feature CNN:**
   A standard CNN with a Feature Pyramid Network (FPN) is used to extract multi-level features from the input image pair (I^A and I^B). In this step, two sets of features are produced:

   - Coarse-level features ($\tilde{F_A}$ and $\tilde{F_B}$): Extracted at 1/8 of the original image resolution, these features benefit from the CNN's built-in translation equivariance and locality, making them well suited for capturing the overall structure while reducing computational cost.
   - Fine-level features ($\hat{F_A}$and $\hat{F_A}$): Extracted at 1/2 of the original resolution, these features retain more spatial detail for later refinement.

2. **Coarse-Level Local Feature Transform:**
   The coarse feature maps are flattened into 1-D vectors and enriched with positional encodings. These enhanced features are then processed by the Local Feature Transformer (LoFTR) module, which consists of multiple self-attention and cross-attention layers. This transformer module adapts the features to be more context- and position-aware, making them easier to match across the two images.

3. **Matching:**
   A differentiable matching layer computes a confidence matrix ($\mathcal{P}_c$) by comparing the transformed features from the two images. From this matrix, coarse match predictions ($\mathcal{M}_c$) are selected based on a confidence threshold and mutual nearest-neighbor criteria. This yields a set of initial, but roughly localized, feature correspondences.

4. **Coarse-to-Fine Refinement:**
   For every coarse match in $\mathcal{M}_c$, a local window of size $w \times w$ is cropped from the corresponding fine-level feature maps. Within each local window, further refinement is carried out to adjust the match positions to a sub-pixel level, resulting in the final, high-precision match predictions ($\mathcal{M}_f$).

In summary, the processing pipeline starts with extracting coarse and fine features via a CNN, transforms the coarse features with a transformer-based module to incorporate global context, performs initial matching through a differentiable layer, and then refines these matches locally using the high-resolution fine features. This detector-free, coarse-to-fine approach allows for efficient and accurate feature matching between image pairs.

5.3.2 Application of LoFTR for camera pose estimation

The process for this approach involves the same key steps but some of them are revisited as follows:

For calculating the matched points was used the implementation provided by the developers in a Google Colab environment. The code was slightly modified to allow for the saving and exporting of the matched points calculated by LoFTR.

A new visibility file was then constructed using the matched points obtained from LoFTR. This file replaced the reference points of the original image with the matched points, using the same distance threshold as before. This step established the fundamental correspondences between the 2D image points and the 3D model points.

Finally, the newly created visibility file was input into Fiore's method to estimate the camera's position and orientation, following the same procedure as previously described.

This approach was applied to the same reference and target images used previously. The results were significantly improved, with an estimated camera pose error of 0.80 in translation and 0.50 in rotation, demonstrating better accuracy compared to the previous method.

5.4 Performance Comparison

5.4.1 Feature Matching Differences

The first analysis focused on the results of applying the feature matching algorithms. As previously mentioned, the purpose of these algorithms is to identify corresponding points between two images.

SIFT detects keypoints based on areas of high contrast, leading to a sparse and uneven distribution across the image. These keypoints often cluster in regions with strong texture variations. In contrast, LoFTR selects keypoints only for the reference image and in a more uniformly distributed manner, covering the entire image more effectively.

Figure 11: SIFT keypoints

As shown in the figure, SIFT keypoints tend to be scattered irregularly, forming clusters in high-contrast areas. LoFTR, on the other hand, produces a more structured distribution, resembling a regular grid. Additionally, blue circles highlight the points that are sufficiently close to visibility points, indicating their potential usefulness in subsequent pose estimation steps.

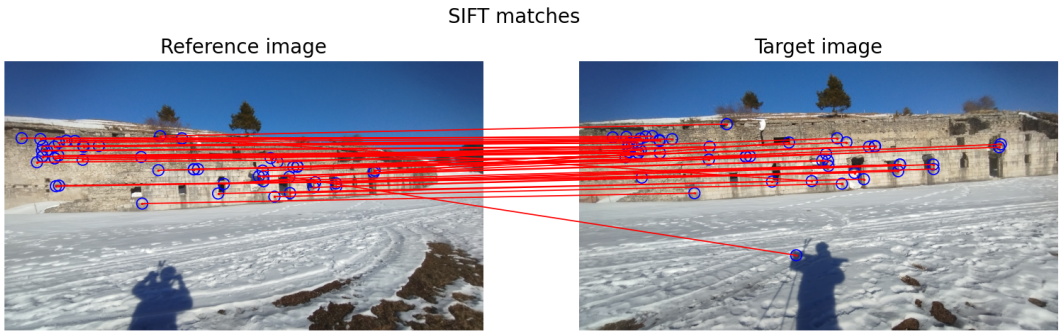


Figure 13: Enter Caption

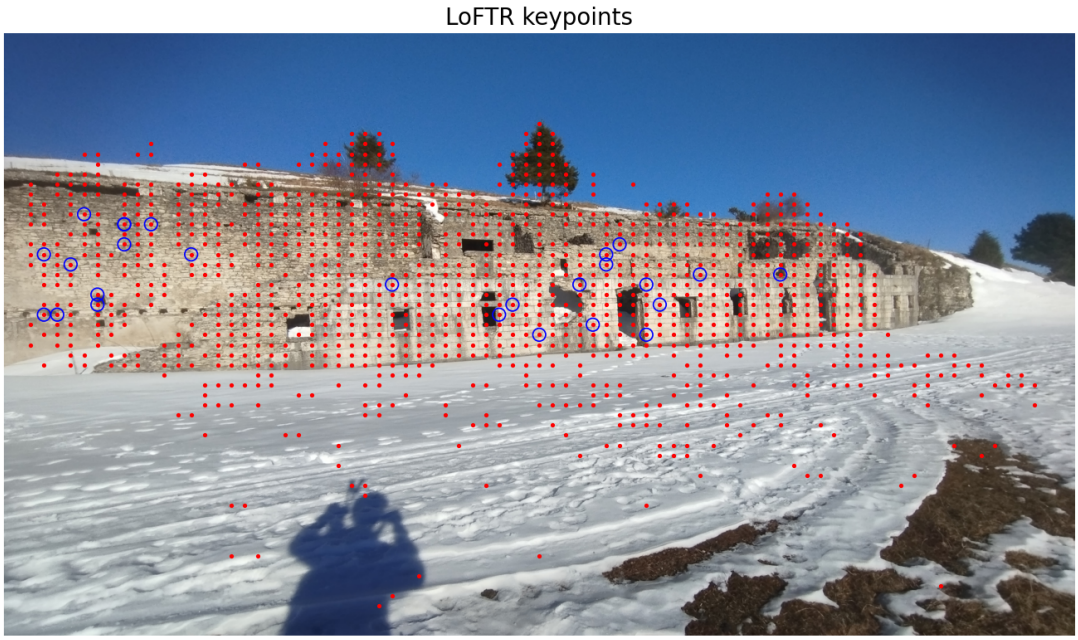


Figure 12: LoFTR keypoints

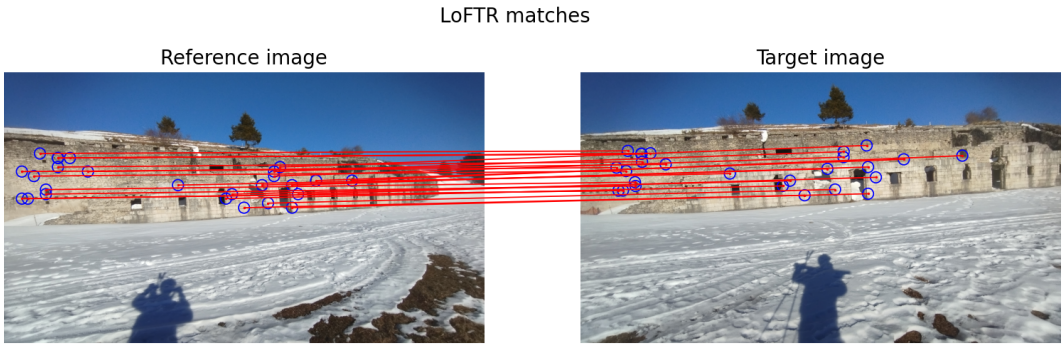


Figure 14: Enter Caption

In the last two figures, the matched points shown are those that are close to the visibility points, meaning they are the ones effectively used in the pose estimation process. Both methods produced visually good correspondences, though the evaluation was performed qualitatively by inspection. The points appeared to match well between the reference and target images, confirming the effectiveness of both algorithms in establishing correspondences.

However, in the case of SIFT, at least one of the matched points was clearly misplaced, significantly deviating from its expected location. This misalignment could introduce errors that propagate through the pose estimation process, potentially affecting the final results. The impact of these inaccuracies will become more evident in the next section, where the alignment performance of the estimated camera poses is analyzed.

5.4.2 Pose Estimation Accuracy

To evaluate the impact of these differences on camera pose estimation, an experiment was conducted using 10 different reference images and measuring the error in the resulting camera pose. The results are summarized in the table below:

Reference Image	Translation Error (SIFT)	Translation Error (LoFTR)	Rotation Error (SIFT)	Rotation Error (LoFTR)
Image 1	2.54	**0.80**	6.03	**0.50**
Image 2	89.88	**8.95**	28.04	**4.87**
Image 3	23.81	**1.07**	13.72	**0.62**
Image 4	7.68	**1.65**	3.07	**0.96**
Image 5	11.01	**1.28**	5.25	**0.74**
Image 6	0.34	**0.23**	1.90	**0.13**
Image 7	0.61	**0.44**	1.63	**0.25**
Image 8	18.80	**0.55**	8.65	**0.31**
Image 9	34.42	**5.40**	15.47	**2.93**
Image 10	**1.73**	1.79	2.15	**1.02**
Mean	19.08	**2.21**	8.59	**1.23**

Table 1: Error comparison between SIFT and LoFTR methods.
The better results (lower error) are highlighted in bold.

This table shows that LoFTR consistently provides more reliable and stable results, with lower error measurements across every translations except for one. For rotation error, LoFTR outperforms SIFT in all case, making it the preferred method for feature matching in this project.

There were a few outliers during the experiment. For example, the second reference image exhibited a significantly large error in both translation and rotation for the SIFT-based approach. Upon investigating the reason for this anomaly, it was found that some of the matched points were placed in completely incorrect positions, propagating the error through the subsequent steps of the method. Despite this outlier, which can be considered an exception, SIFT exhibits greater variability in its results and requires more careful tuning and attention to achieve optimal performance.

6 Final Scene Interaction

For the augmented reality application, a newly created model was used compared to the previous experiments. This one is a fusion of the drone dataset model and the photogrammetry model made with the camera. The fusion was done in Blender by aligning the two models as accurately as possible and removing parts of the drone model that were already present in the other model but with a more detailed appearance. This ensured a seamless combination, preserving both the broader landscape coverage and the high level of detail in the barracks. The most important thing to take into account in this step was to keep the origin of the more detailed model fixed to avoid any alignment issues later when integrating the images. In addiction, for this model, the textures generated by Zephyr were also retained to achieve the highest level of realism possible.

Figure 15: 3D model used in the game.

The model was imported into Unity, where a simple movement system was implemented to allow

users to freely explore the reconstructed environment.

For the interactive component, a "photo matching challenge" was developed. In this challenge, a photo appears in front of the player's view, and the objective is to align it with the 3D model as accurately as possible. The closer the player gets to the actual position from which the photo was taken, the higher the score. This mechanic encourages careful observation and engagement with the virtual space while demonstrating the accuracy of the photogrammetric reconstruction.

Figure 16: Gameplay

References

[1] LoFTR: Detector-Free Local Feature Matching with Transformers: Jiaming Sun and Zehong Shen and Yuang Wang and Hujun Bao and Xiaowei Zhou