High-Fidelity 3D Reconstruction of Space Bodies Using Machine Learning and Neural Radiance Fields

Timothy Jacob Huber
Florida Institute of Technology, thuber2019@my.fit.edu

Follow this and additional works at: https://repository.fit.edu/etd

Part of the Aerospace Engineering Commons

Recommended Citation
Huber, Timothy Jacob, "High-Fidelity 3D Reconstruction of Space Bodies Using Machine Learning and Neural Radiance Fields" (2024). Theses and Dissertations. 1442.
https://repository.fit.edu/etd/1442

This Thesis is brought to you for free and open access by Scholarship Repository @ Florida Tech. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Scholarship Repository @ Florida Tech. For more information, please contact kheifner@fit.edu.
High-Fidelity 3D Reconstruction of Space Bodies Using Machine Learning and Neural Radiance Fields

by

Timothy Jacob Huber

A thesis submitted to the College of Engineering and Science of Florida Institute of Technology in partial fulfillment of the requirements for the degree of

Master of Science
in
Aerospace Engineering

Melbourne, Florida
May, 2024
We the undersigned committee hereby approve the attached thesis, “High-Fidelity 3D Reconstruction of Space Bodies Using Machine Learning and Neural Radiance Fields.”
by
Timothy Jacob Huber

Madhur Tiwari, Ph.D.
Assistant Professor
Aerospace, Physics, and Space Sciences
Major Advisor

Eric D. Swenson, Ph.D.
Associate Professor
Aerospace, Physics, and Space Sciences

Seong Hyeon Hong, Ph.D.
Assistant Professor
Mechanical and Civil Engineering

Ratneshwar Jha, Ph.D.
Professor and Department head
Aerospace, Physics, and Space Sciences
Abstract

Title: High-Fidelity 3D Reconstruction of Space Bodies Using Machine Learning and Neural Radiance Fields

Author: Timothy Jacob Huber

Advisor: Madhur Tiwari, Ph.D.

In the era of burgeoning space exploration, the growing population of spacecraft heightens the inevitability of collisions. While traditional imagery remains effective for damage assessment, its lack of 3D representation of the object necessitates more advanced approaches. This research delves into cutting-edge methodologies, with a primary focus on leveraging machine learning and computer vision technologies to enhance the precision and efficiency of damage assessment by taking imagery of a space body and creating a 3D model.

The study extensively investigates TransMVSNet, a neural network code employing classical computer vision techniques such as multi-vision stereo (MVS) and depth maps. This approach enables the construction of high-fidelity scene models based on motion dynamics.

Moving beyond traditional paradigms, the research explores Neural Radiance Fields (NeRF), an innovative solution that integrates positional encoding and ray integration for rapid and accurate scene reconstruction. NeRF demonstrates superior efficiency compared to conventional methods, significantly reducing reconstruction time while minimizing errors.

This paper aims to engage in a comprehensive analysis of advanced machine learning solutions in addressing the intricacies of space systems. By studying ways to implement machine learning and computer vision into satellite damage assessment.

Table of Contents

iii
Abstract .......................................................................................................................... iii
List of Figures .................................................................................................................. v
List of Tables ................................................................................................................... vi
Acknowledgment ........................................................................................................... vii
Dedication ....................................................................................................................... viii
Chapter 1 Introduction ................................................................................................. 1
  1.1 Statement of Investigation ................................................................................. 1
  1.2 Related Works ..................................................................................................... 1
  1.3 Principles of Computer Vision .......................................................................... 3
  1.3 The Dataset ........................................................................................................ 9
Chapter 2 Training And Testing .................................................................................. 17
  2.1 TransMVSNet .................................................................................................... 17
  2.2 Instant NeRF ..................................................................................................... 27
Chapter 3 Conclusion and Recommendations ......................................................... 36
  3.1 Conclusion ......................................................................................................... 36
  3.2 Recommendations ............................................................................................ 37
References ..................................................................................................................... 39
Appendix ......................................................................................................................... 41
  Custom Depth Map Maker Code (Python) ............................................................... 41
  Camera Extrinsic Checker and Plotter (Python) ..................................................... 49
  Blender Output to COLMAP Database Code (Python) ......................................... 55
  Depth Map Maker from input Images [MVS] (Python) ........................................ 57
List of Figures

Figure 1: Camera Frame to World Frame ................................................................. 4
Figure 2: Key Point Extraction from Camera Frame [7] ............................................. 5
Figure 3: Camera Intrinsic Matrix with Frames of Reference ..................................... 6
Figure 4: SPEED+ Raw Dataset Image ...................................................................... 10
Figure 5: SPEED+ Reconstruction with COLMAP .................................................. 11
Figure 6: SmallSat Image .......................................................................................... 12
Figure 7: Blender Light and Camera Layout ............................................................... 13
Figure 8: Reconstruction of SmallSat Using COLMAP ............................................. 13
Figure 9: Camera Orientation in World Frame .......................................................... 14
Figure 10: TransMVSNet Flow Chart ......................................................................... 17
Figure 11: Coordinate System Change Tests for TransMVSNet ................................. 22
Figure 12: Blender Depth Maps (Left) and Scripted Depth Maps (Right) ..................... 24
Figure 13: Contour Plot of Small Sat ......................................................................... 25
Figure 14: DTU (Right) Comparison to SmallSat (Left) for TransMVSNet ............... 26
Figure 15: NeRF Fox Model Mesh ........................................................................... 31
Figure 16: LabSat Satellite Image ............................................................................... 33
Figure 17: LabSat Results from Instant NeRF ............................................................ 34
List of Tables

Table 1: Research Results........................................................................................................36
Acknowledgment

An expression of gratitude is given to Dr Madhur Tiwari Ph. D. from Florida Institute of Technology’s Department of Aerospace, Physics, and Space Sciences, for his constant guidance in academia and research. As well as to everyone who contributed to this research. Finally, the United States Air Force for funding this research as part of the United States Air Force Contract No. FA875023C1012.
Dedication

This thesis is a dedication towards my parents for their never-ending support in all facets of life.
Chapter 1
Introduction

1.1 Statement of Investigation

In today's era of expanding space exploration, the increasing number of spacecraft raises concerns about potential collisions. Traditional methods for assessing damage, though effective, often lack the precision needed for accurate evaluations. This study explores new ways to improve damage assessment using advanced technologies like machine learning and computer vision.

This research focuses on evaluating two specific machine learning algorithms designed to reconstruct 3D images. One approach, called TransMVSNet [1], combines classic computer vision techniques to create detailed models of scenes based on motion. The other method, Neural Radiance Fields (NeRF) [2], uses innovative techniques to reconstruct scenes quickly and accurately.

This research compares these two methods carefully to see which one works best for reconstructing a space body with high degrees of accuracy and resolution. By understanding the differences between them, the goal is to develop ways to improve autonomy in space exploration and infrastructure development. This research aims to provide valuable insights for future projects, to make informed decisions, and to push the boundaries of autonomy in space exploration.

1.2 Related Works

The exploration of 3D reconstruction methodologies is not novel within the realm of research, with numerous networks dedicated to this purpose which will be discussed next. One such endeavor, led by Lim Wen Peng (2004), delves into 3-dimensional reconstruction through the utilization of an artificial neural network (ANN), bearing resemblances to the TransMVSNet framework. This particular approach employs a conventional system for
estimating the z-coordinate or distance from each camera to the object. However, distinct from TransMVSNet, this paper adopts a unique set of inputs—the [x,y,z] coordinates of a clockwise Cartesian system, normalized within the bounds of [-1, 1]. While this normalization simplifies the computational process, it compromises accuracy, resulting in differing dimensions from the real model. The network, equipped with only one output, determines whether a point belongs to the object surface, expressed through a binary Boolean value of "yes" (1) or "no" (0). Additionally, the paper explores a mathematical approach called the 3rd Order Polynomial Method, albeit facing limitations in estimating the z-coordinate for objects with more than 20 points when provided with unorganized surface data. Furthermore, the network's simplistic nature yields simplistic 3D models, lacking in high-fidelity qualities, rendering it inadequate for comprehensive reconstructions[3].

Another noteworthy contribution to the domain of 3-dimensional reconstruction, authored by Audrius Kulikajevas (2019), shares similarities with the aforementioned work. This approach entails reconstructing object structures smoothly with polygonal meshes derived from a single depth frame, akin to the previous methodology. Employing a prebuilt dataset named ShapeNetCore, this work reconstructs nine real-life models utilizing the described method. The model architecture consists of a single predictor node and multiple reconstructor nodes, each dedicated to recognizing specific objects or groups. This streamlined approach enables repeated testing without necessitating network retraining after each trial, thereby enhancing algorithm efficiency. Operating on a smaller network scale, the input comprises a singular depth frame of size [320 x 240]. Following image placement, the network performs voxel cloud post-processing to transform the reconstruction algorithm output into a polygonal mesh, thereby generating the final mesh representation. Despite producing fully rendered voxel representations of objects, similar to the prior approach, this method also falls short in achieving high-fidelity reconstructions[4].

Irrespective of the algorithm under exploration for 3D reconstruction, the acquisition of high-fidelity scene imagery remains paramount. Particularly in the realm of space imagery, acquiring precise datasets poses significant challenges. Existing datasets often rely on physics engines like Blender or Unreal Engine to generate synthetic imagery. Notable
examples include URSO (2019) and SPEED+ (2021), employing simulation software to create photorealistic imagery and physical models for machine-learning purposes. While synthetic imagery offers scalability, it often lacks visual fidelity and illumination accuracy inherent in real-life space systems. To mitigate these approaches like SPEED+ (2021) incorporate physical models and testbeds to capture imagery under realistic lighting conditions, ensuring the fidelity and accuracy necessary for training machine learning algorithms. Simulation software such as Blender and Unreal Engine play pivotal roles in this process, facilitating the creation of high-fidelity imagery for space systems through meticulous top-down simulations.

1.3 Principles of Computer Vision

This research is not only rooted in machine learning but also in computer vision. The large majority of this project was the exploration of coordinate systems used for the camera frame for these images. Since the underlying formulation of reconstructed scenes from imagery is, by its roots, the coordinate system changes to overlay a variety of static imagery from the cameras' frame of reference. Into a world frame where space and volume are taken into consideration. Figure 1 demonstrates the concept of taking imagery from the camera frame and overlapping key points of the scene into a world frame to create a 3D reconstruction of the scene. This reconstruction style depicted is typically referred to as sparse reconstruction.

In Figure 1, the red icons depict the pose and orientation of the camera in the world frame when the image was taken. For this example, the origin of the object is aligned with the origin of the world frame, as depicted in the Figure as the red, green, and blue axis lines [X, Y, Z]. This form of reconstruction is dubbed structure from motion (SfM) and is the process of estimation of the 3D structure of an object from a set of 2D imagery. This image in particular was curated from testing done with a custom dataset that will be discussed.
To reconstruct a static scene, a scene in which the camera is moving around the object and the object remains static, key points from the camera’s images need to be converted into the world frame. Meaning a coordinate system for the camera needs to be outlined as well. Figure 2 dictates how the images in each camera rotate about a scene, select key points between sets of image pairs and through this coordinate transfer, can place these points in 3D space in the world frame.

From Figure 2, better observations can be seen on how the imagery is taken from 2D to 3D. As well as the coordinate system now described from the camera’s frame of reference. In the camera frame, the z-axis points into the image, this is described as the depth of the image. This means that the z-axis for the camera always points towards the object being photographed. The y-axis of the camera also has an orientation that stays constant, this axis always points perpendicular from the line of the camera to the scene, as if the camera
was held right side up in the world frame. This dictates the top of the camera to ensure that imagery is not being confused for being taken upside down in the world frame. This leaves the x-axis in the orientation that completes the triad.

![Figure 2: Key Point Extraction from Camera Frame [7]](image)

The determination of depth of a key point in a scene is determined by the internal parameters of the camera. These internal parameters are typically stored in a matrix called the intrinsic matrix. The intrinsic matrix of a camera stores all the internal metrics of the camera. This includes the focal length of the camera \( f \); this is the distance between the point of convergence of the camera lens and the photometric sensor in the camera. This focal length is typically expressed in units of millimeters. The other piece of information that is stored in the intrinsic matrix is the principal point coordinates \( x \). These coordinates are the
location of the point on the surface of the camera’s lens where light initially comes into contact leading to the refraction process. This is typically the center of the image as most camera lenses are convex in shape [7]. This information is split into $x$ and $y$ components and then placed into the matrix in an upper triangular formation as shown in Figure 3. $S$ is the skew of the principal point from the true center of the image. This index is reserved for specialized camera lenses that refract the light differently.

![Camera Intrinsic Matrix with Frames of Reference](image)

Figure 3: Camera Intrinsic Matrix with Frames of Reference

The creation of the intrinsic matrix allows for the conversion from pixel coordinates in the image to estimated distance in the world frame. This means that the pixel grid of the image can be directly converted into distance, allowing the creation of a 3D volumetric representation of a scene from static 2D imagery. When a key point is observed in two image pairs, both pair’s instances in the image are located and compared to each other. This is when the orientation of the camera in the world frame is also considered. By finding the distance between the centers of each camera in the camera pairs and finding the corresponding key point in each image on the image, using triangulation, the distance from the camera can be estimated to the 3D volumetric scene as seen back in Figure 2. Through this triangulation, an estimation of where the key point in the images is located in 3D space can be visualized and then plotted in 3D space. This is how SfM pipelines can create high-fidelity models from a static scene.
For the neural networks in this research, both networks operate using the structure of motion software COLMAP [8] to dictate the locations of the cameras in 3D volumetric space relative to the world frame and the object being reconstructed. These positions of the cameras in the world frame are stored in an extrinsic matrix, a rotation matrix from the camera frame to the world frame. These matrices store the information on the camera’s position as well as its orientation and rotation.

COLMAP, an open-source software package for SfM and Multi-View Stereo (MVS), operates by reconstructing the 3D structure of a scene from a collection of 2D images taken from different viewpoints. It begins by extracting distinctive features from each input image using algorithms such as Scale-Invariant Feature Transform (SIFT) or Speeded-Up Robust Features (SURF). SIFT detects key points in the image that are invariant to scale, rotation, and illumination changes, making them suitable for matching across different views. SURF, on the other hand, computes local image descriptors based on gradients, providing a faster alternative to SIFT [9]. This means that SIFT is more suitable for complex scenarios and when the algorithm is looking for the whole object, while SURF is used for confirmation. This is exactly why COLMAP uses both SIFT and SURF in the reconstruction process.

These features are then matched between pairs of images to estimate geometric transformations and filter out incorrect matches. Using the matched feature correspondences, COLMAP computes camera poses and sparse 3D point clouds representing the scene's structure through triangulation. Employing an incremental SfM approach, COLMAP builds the reconstruction incrementally by adding images and refining the existing 3D structure.

The triangulation process in COLMAP involves registering a newly captured image to observe existing scene points and potentially extend scene coverage through triangulation. When a new scene point \( X_k \) is triangulated, it can be added to the set of points \( X \) once at least one more image, capturing the new scene part from a different viewpoint, is registered. Triangulation plays a critical role in SfM by enhancing the stability of the existing model through redundancy and facilitating the registration of new images by providing additional
2D-3D correspondence. While numerous methods for multi-view triangulation exist, many suffer from limited robustness or high computational cost for use in SfM [10].

The triangulation process in COLMAP involves determining the 3D coordinates of scene points by leveraging the information from multiple registered images. Let \((u_1^i, v_1^i)\) and \((u_2^i, v_2^i)\) represent the pixel coordinates of a feature point in two registered images \(I_1\) and \(I_1\), respectively. These correspondences are expressed in homogeneous image coordinates as \((x_1^i, x_2^i)\). Each registered image is associated with a camera projection matrix \(P\), which maps a 3D point \(X\) to its 2D image coordinates \(x\). The triangulation process aims to find the 3D coordinates of the point \(X\) by solving the equation \(P_1X = P_2X\), where \(P_1\) and \(P_2\) are the camera projection matrices for the two registered images. This equation can be solved using methods such as Direct Linear Transformation (DLT), which formulate the problem as a linear least squares optimization. Once the 3D coordinates of the point \(X\) are determined, it can be added to the scene reconstruction. This iterative process of triangulating new scene points from multiple viewpoints contributes to the refinement and expansion of the 3D model in COLMAP.

The projection matrix of a camera \(P\), is directly linked to the intrinsic matrix of a camera \(K\) by this mathematical formula:

\[
P = K[R|t]
\]  

(1)

Where \(R\) is the rotation matrix representing the camera’s orientation in the world coordinate system, and \(t\) is the translation vector. These two together create the extrinsic matrix. The camera projection matrix combines the intrinsic matrix with the extrinsic parameters of the camera (rotation and translation) to map 3D points in the world coordinate system to their corresponding 2D image coordinates.

Bundle adjustment is subsequently performed to optimize camera poses and 3D point positions jointly, improving the accuracy of the reconstruction by minimizing
reprojection errors. Optionally, COLMAP can perform dense reconstruction using MVS techniques to estimate depth information for each pixel in the images, resulting in a dense 3D representation of the scene. Finally, COLMAP applies texture mapping to the reconstructed 3D model, projecting the original images onto the 3D surface to create a realistic textured mesh. This is why the network explored in this research utilizes the MVS approach towards reconstruction and was chosen for this research. This comprehensive process enables users to efficiently reconstruct detailed 3D models from images, making COLMAP a valuable tool.

1.3 The Dataset

Datasets for neural networks for 3D reconstructions consist of sample imagery, usually in the magnitude of hundreds of images, and extrinsic and intrinsic data of the camera. The necessity of a large amount of sample images is to ensure that the scene is well represented, and all details of the object are persevered in the 3D reconstruction. The first dataset used in this research was from SPEED+, which involved thousands of images of a sample satellite in synthetic environments meant to mimic real-life space environments [6]. These images came complete with camera intrinsic and extrinsic that gave the location of the camera relative to the satellite. This means that when the reconstruction of the model is completed by the neural network, the satellite will be in the origin of the world frame. This practice is usually the case in these reconstructions due to its simplicity and the avoidance of a coordinate transfer. Before the dataset can be placed into the neural network, the images must be placed through the software COLMAP, to find strong image pairs and their associated transformation between one another in the world frame coordinates. This is to find the distance between the two images for the triangulation process between key points in the scene. Once these image pairs are found, usually the two images corresponding to images taken close to each other in world frame space, COLMAP creates three files used in the reconstruction of the scene in the neural networks. These files consist of an image pair that shows the target camera or source camera and the nine most strongly associated cameras
with the scene ranked in descending order of resemblance, the nine closest images in the world frame. The images are called the reference images. The rest of the files are individual camera intrinsic and extrinsic matrices used in the reconstruction process. This data accompanied by the images themselves make up most of the training data for the networks. This will allow for testing of the networks without the use of the camera intrinsic and extrinsic and create a model from only sample imagery which is the overarching goal of this research.

![Figure 4: SPEED+ Raw Dataset Image](image)

While the SPEED+ dataset has everything desired for the reconstruction of a satellite, there was one critical overarching issue that led to the discontinuance of this dataset in the research. This was the lack of details on the sides of the satellite. On the quarter panels of the satellite, there are instruments and hardware that are excellent key points for the reconstruction process. However, most of the quarter panels are the same shade and don’t
have imperfections due to the dataset being mostly synthetic imagery. This leads to holes in the reconstruction because the software and neural network cannot fill in those problem areas where there is extraordinarily little differentiation between one point on the surface to the other. Similar to a piece of paper where the entirety of it looks the same and is indifferentiable from one spot to another. Figure 5 is a reconstruction made in COLMAP from running the raw camera data and the images into the SfM software before testing on the network.

![Figure 5: SPEED+ Reconstruction with COLMAP](image)

As one can observe, COLMAP does an excellent job in selecting and placing the solar panel of the SPEED+ satellite as well, and the general shape of the object is preserved. However, with the quarter panels of the satellite come the physical holes in the model. This is due to the lack of details for the key point detection software to select and piece together from other views. Furthermore, by doing the COLMAP step before the neural network training and testing. A baseline model can be viewed and compared after the training process. However, this model is not acceptable for the neural networks to reconstruct due to the lack
of details in the mesh. A mesh is the surface area made during the reconstruction process. It is because of this lack of details in the mesh that the SPEED+ dataset had to be abandoned for a better alternative. This is a custom dataset with the ability to change to fit the needs of each of the networks independently. This will also give the ability to change certain aspects of the dataset to ensure the best results.

With the creation of a custom dataset comes the flexibility to change the dataset as issues emerge. Coming out of the initial tests with the SPEED+ dataset with knowledge of what the new model should contain. A new model was created on Blender, a physics simulation software. By finding a pre-built satellite model and adding to the model to ensure that there are more textures and objects to be treated as key points. The assurance of a clean and full model can be fulfilled. The Blender model consists of a satellite model at the world origin of the Blender file and then a single camera that orbits the satellite in three distinct orbits. The first orbit is at zero elevation from the origin XY plane. Meaning that the camera is in the same plane as the satellite. The second and third orbit are elevated 12 meters above and below the satellite and the camera is pitched 45 degrees to look at the satellite. Along with the camera orbits, seven lighting conditions were constructed to ensure proper shadowing was displayed and even more key points could be detected. Figures 6 and 7 display the sample image of the satellite as well as the lighting orientation with the camera placement in the world frame. This dataset is dubbed SmallSat.

Figure 6: SmallSat Image
From Blender, the camera extrinsic data was extracted, and the camera intrinsic data was recorded. The images and data were then fed into COLMAP for the training data. The following resulted in Figure 8. As one can observe, the addition of foil-like textures and reinforcement beams to the backs of the solar panels aided in the reconstruction process and led to a clean and concise mesh. This reconstruction is ideal for being fed into the neural networks for reconstruction without the use of the camera orientation and data.
Multiple tests were performed to find the best orientation for the orbits as well as the number of pictures to take in each orbit. The first configuration done for this test was overlapping orbits as seen in Figure 1 and having seventy-five images taken per orbit. While this orientation gave decent results, one overarching issue was the curving of the solar panels in the mesh. This was a result of the reference image and the pairing image being associated with the key points being too similar to one another. Meaning that as the camera rotated around the spacecraft the images did not change enough for the algorithm, causing the resulting mesh to warp in places. The solution for this was the decrease in the number of images taken in each orbit down to twenty-five. Another issue was the fact that some cameras were in the same place in the world frame as others in the overlapping orbits, this caused issues with the pairing of cameras and resulted in more warping of the mesh. As a result, the solution to this issue was to make the orbits layer on top of one another and not overlap in any position. The resulting orientation of the camera frames in the world frame can be seen in Figure 9. From the resulting tests to the dataset, the data collection process is complete and is prepared to be tested and trained on the neural networks. This will be discussed in the next coming chapters.

![Coordinate Orientations for Cameras](image)

**Figure 9: Camera Orientation in World Frame**
Other issues that arose were the backs of the solar panels having missing textures in the model. Giving similar results to that on SPEED+. This issue was resolved with the creation of a heat-shielding texture used in most satellites used today. This foil textured shielding on the backs of the solar panels as well as the addition of supporting structures to the model. Helped break up the missing space on the back of the solar panels. Allowing for a more complete mesh to be constructed before the neural networks were applied.

Extrinsic data for the cameras is of the utmost importance for the creation of 3-dimensional models. The correct placement of cameras is critical for the completeness of a mesh. This meant that the extrinsic data collected from Blender needed to be precise. For the collection of images in Blender. The camera was rotated around the object, similar to how most 3-dimensional datasets collect images of real-life objects. In DTU’s case, a bucket was captured in the same respect. This in turn meant that the camera orientation was constantly changing as the orbit of the camera progressed. As testing with the dataset on COLMAP progressed issues were arising with the quaternions created by the Python script developed in Blender to create and save the images and quaternions of the camera for each image. This issue arose from Blender using a different coordinate system than COLMAP. While COLMAP’s sparse reconstruction, like the one that can be seen in Figure 1, was in the world frame of reference, the cameras for the reconstruction were in a different frame of reference, this frame was the camera frame. Meaning that the cameras’ positions in the COLMAP sparse reconstruction were positioned backward or facing away from the satellite being reconstructed. To fix this issue a custom script was coded in Python to hardcode the positions of the cameras in each orbit in the correct frame of reference, the world frame. This could have been achieved as well using a coordinate transfer, however, during the experimentation of the coordinate system of the Blender dataset, the original coordinate system that the quaternions were expressed in was not clear. Meaning that the creation of a custom script to hardcode these camera positions was the better alternative. The code for the Blender dataset can be viewed in the appendix.

By using the COLMAP SfM software before the neural networks. The insurance of a proper mesh could be visualized to ensure that the resulting mesh could be compared to
the input data that would be given to the neural networks. Make sure that the model is complete before implementation.
Chapter 2
Training And Testing

2.1 TransMVSNet

The first network to be evaluated in this research was the TransMVSNet algorithm. An algorithm that is based on more traditional methods of 3D reconstruction of a scene via the use of depth maps and input images as well as the image pairs with extrinsic and intrinsic data. However, this network was not readily available to have imagery sent through it and needed to be further investigated to ensure that the results were as high-fidelity as expected.

The process in which the TransMVSNet algorithm trains is that it requires the input training images to be fed through a SfM network, in this case, COLMAP was used. From this fed-through approach, COLMAP produces the image pairs as well as the estimated camera extrinsic and intrinsic. For this case, COLMAP did not need to estimate the camera extrinsic data due to the Blender custom dataset having already known poses. From this camera, extrinsic data, and known intrinsic data from Blender, the only need for COLMAP was to find the image pairs that closely associate with one another. These outputs from COLMAP as well as the raw input images and the depth maps created in Blender are then fed through the network for the training process. After training, the testing of the raw images without the use of extrinsic data can be commenced. Finally, output mesh is produced of the scene. A diagram of the network flow can be viewed in Figure 10.

Figure 10: TransMVSNet Flow Chart
TransMVSNet architecture begins with a feature pyramid network (FPN) to extract deep image features. The network does this three times during training at different intervals of coarse-to-fine resolution levels. This in turn represents patterns in the mesh and addresses areas of the object where textures may be less than optimal, and the need for global context information and inter-image feature interaction is needed to reduce the uncertainty of matching. [1]. After the FPN, the input is then passed to an adaptive receptive field (ARF) used to refine the local feature extraction and ensure a smooth transit to the feature-matching transformer (FMT). This FMT performs the attention-based algorithm discussed in the next few paragraphs. FPN's primary function lies in generating a hierarchical pyramid of feature maps from input images, capturing information at diverse levels of abstraction and resolution. By amalgamating both local and global context information, FPN facilitates a comprehensive understanding of the input, crucial for tasks such as object detection and semantic segmentation. Its scale-invariant representation ensures robust handling of objects at varying distances and sizes within scenes, ultimately enhancing localization accuracy. Moreover, FPN achieves computational efficiency by reusing features across different resolutions, minimizing redundant computations.

TransMVSNet architecture integrates sophisticated mechanisms like Scaled Dot-Product Attention, drawing parallels with information retrieval practices. Here, features are categorized into query (Q), key (K), and value (V) groups. Q interacts with V based on the attention weight derived from the dot product of Q and K corresponding to each V. Mathematically, the attention layer is represented as depicted in equation 2. This mechanism evaluates feature-wise similarity between Q and K, extracting information from V accordingly. Embracing the multi-head attention approach, the network divides feature channels into $N_h$ groups, effectively enhancing feature interaction.

$$\text{Attention}(Q,K,V) = \text{stmax}(QK^T)V$$  \hspace{1cm} (2)
While multi-head attention computes attention through the dot product of Q and K, it leads to quadratic growth in computational costs concerning the input sequence length. To mitigate this, TransMVSNet leverages a linear transformer, to compute attention efficiently. This linear transformer replaces the conventional kernel function with the properties shown in equation 3, with $elu(\cdot)$ symbolizing the activation function of exponential linear units. This substitution drastically reduces computational complexity to a linear scale, enabling attention computation on high-resolution images, despite the smaller number of channels compared to the input sequence length.

$$Attention(Q, K, V) = \Phi(Q)\Phi(K^T)V, \quad \Phi(\cdot) = elu(\cdot) + 1$$  \hspace{1cm} (3)

The $elu(\cdot)$ can be view from the relationship as shown here:

$$elu(x) = \begin{cases} x, & \text{if } x \geq 0 \text{ or } \alpha \ast (\exp(x) - 1), & \text{if } x < 0 \end{cases}$$  \hspace{1cm} (4)

where $\alpha$ is a hyperparameter controlling the saturation value for negative inputs. $elu(x)$ aims to address the vanishing gradient problem encountered in training deep neural networks, particularly in regions where traditional activation functions such as sigmoid or tanh saturate. By allowing negative values, $elu(x)$ ensures a smooth gradient flow during backpropagation, which can lead to faster and more stable training. This substitution helps to significantly reduce computational complexity, enabling efficient attention computation on high-resolution images while maintaining a linear scale of complexity.

This approach to handling attention in the neural network allows for two forms of attention-based calculations to be derived, those being intra-attention and inter-attention.
When the Q and K vectors are features from the same input image, the attention layers retrieve relevant information about the scene within a given view. This is intra-attention when the Q and K are present in the same view. As a result, when the Q and K vectors are present in two different images or different views, attention layers then capture information in a multi-view feature extraction. This is known as inter-attention. TransMVSNet requires multiple images to create a reconstruction comparison between two image pairs is needed using this method. TransMVSNet does this by creating image titles such as reference images and source images. The source images refer to the images captured by the camera, typically representing different viewpoints of the same scene or object. These images serve as input to the network for the task of multi-view stereo (MVS), which aims to reconstruct the 3D geometry of the scene or object from multiple 2D images. On the other hand, reference images are specific images chosen from the source images that serve as a reference during the MVS reconstruction process. These reference images are often selected based on criteria such as image quality, viewpoint coverage, or other considerations relevant to the reconstruction task. Source images would include all available images capturing the scene, while reference images may be selected based on their suitability for guiding the reconstruction process or for comparison with the reconstructed 3D geometry.

In the dataset for TransMVSNet, these images are described in the pairs text file that is created from the COLMAP output. This pairs text describes one image and its nine mostly associated pairs, usually the nine closest images in the reconstruction. This first image is the source image. While the other nine are deemed the reference images. TransMVSNet then does intra-attention on the first pair of source images and reference images to ensure a solid baseline for stereo reconstruction. Then inter-attention is applied to gather the global scene information and ensure a high-fidelity model.

After these modules, the TransMVSNet model is prepared to make depth estimations. Due to the earlier coarse-to-fine resolution passes made in the FPN. The model has many points to estimate depth. Since TransMVSNet requires depth maps as an input the model has ground truth values for reconstruction. This means that the depth estimation process can be treated as a loss function as shown in equation 5.
\[ L = \sum_{p \in \{pv\}} \gamma \log (p^d) \]  

where \( L \) denotes the total loss, \( p \) iterates over all pixels \( p \) in the set of pixels \( pv \), \( p^d(p) \) signifies the predicted probability of the depth \( d \) at pixel \( p \), \( \gamma \) stands for a hyperparameter usually set to 2 for complicated scenarios and 0 for simplistic scenarios, and \( \log \) denotes the natural logarithm function. This formulation calculates the loss for each pixel based on the negative logarithm of the predicted depth probability, weighted by the uncertainty of the prediction. The parameter \( \gamma \) adjusts the emphasis on uncertainty, enabling fine-tuning of the loss function to suit specific depth estimation requirements. From this loss function depth maps can be created, and a full mesh can be created.

Setbacks were experienced via the SmallSat dataset and the software COLMAP. This was due to the overlapping in the orbits and the number of images taken in each orbit as discussed in the previous chapter. However, another critical issue was experienced in the deduction of the coordinate system that the cameras images followed in the TransMVSNet algorithm. The first test and training run of the TransMVSNet system, resulted in few points or no points at all in the mesh as can be viewed in the images in Figure 11, which is a few of the later tests done with TransMVSNet when trying to deduce which axis should be pointing in the cameras frame. These tests were done when the lack of results for the first few test runs gave the hypothesis that the cameras were not pointing correctly. Upon further investigation, before the hardcoded cameras extrinsic were implemented it was shown that the cameras positions were in fact skewed in random directions. It was after this that the hardcoded camera extrinsic was implemented and tested for the remainder of this algorithms testing cycle. Further looking into the TransMVSNet algorithm discovered that the algorithm used a majority of the library PyTorch, a widely used library for computer vision. This led to the assumption that PyTorch was instrumental in the algorithm, and then the algorithm must follow the library's built-in coordinate system for cameras. This was the traditional coordinate system discussed in the previous chapter. Before this deduction was made, the hypothesis was that the TransMVSNet algorithm operated in the image frame of reference which is different from the camera frame. Tests were done with this hypothesis
with little success as with the image frame the camera would be tilted. As the x-axis would be facing towards the right, y-axis would be down, and z-axis would be pointing. This means that the camera would have to undergo a rotation around the z-axis of 180 degrees to regain the correct orientation.

Another hypothesis was that the TransMVSNet algorithm required the extrinsic matrix needed to be how the world frame saw the camera. However, looking into the code and the documentation this was incorrect as a coordinate transfer from camera to world was done in the calculations of the algorithm already.

Figure 11: Coordinate System Change Tests for TransMVSNet
All tests done in these images were done in the image frame, z point (leftmost image), y point (middle image), and x point (rightmost image)

After the coordinate system was changed into the correct system, that of the frame of the camera, there still were some issues in the algorithm. The largest issue was the creation of accurate depth maps for the scene. Since this algorithm required depth maps for the reconstruction, depth maps for the SmallSat dataset had to be curated. This was a challenging task due to the complete synthetic nature of the SmallSat dataset. Many approaches were taken to convey accurate depth in the depth maps. One system of approach was the creation of a custom script in Python to develop these maps. This script went through many iterations and developed depth maps from one camera, or a monocular, to the traditional two-camera
approach. All the Python scripts for this project can be found in the appendix. The two-camera approach is more complex and involves the two cameras to take an image of the scene at the same time, these two cameras are separated by a pre-determined distance apart from one another. This distance is called the interaxial distance, increasing or decreasing this interaxial distance will increase or decrease the depth of the scene [12]. For this test, the interaxial distance was set to 0.1 meters and the distance from the scene remained at 45 meters for all tests. Both the single camera and the stereo camera python scripts resulted in identical depth maps which can be viewed in Figure 11. The scripted depth maps were quantized and did not contain accurate measurements of depth. In the Python script as well as in all depth map making software the depth values are constrained to a [1,255] scale. This is to represent the upper most value from the camera and the closest point towards the camera. Since the Blender satellite was spanned at 16 meters. The closest distance to the camera was 37 meters, when the satellite was being viewed from the sides and the solar panels were sideways towards the camera. This meant that in the python code the closet point out of all the pictures was to be white at a pixel value of 255 and the furthest distance was to always be set at 1 or black, typically reserved for the background as it was always furthest. Upon investigation into the depth maps it was shown that the constraints of the depth values were not accurate, and a new avenue needed to be addressed for the curation of the depth maps.

This resulted in the second avenue of the depth map curation since the imagery was captured in Blender the depth maps would also come from the same software. Blender as a physics software has known distances from each point on the scene to the camera. From this data an accurate depth map can be created by creating a custom filter on a camera in Blender to output depth maps. These resulting depth maps can be seen in the comparison figure in Figure 12. As one can observe the Blender depth maps are of higher resolution than the scripted depth maps. Thus, the Blender curated depth maps are used for the remainder of the tests. This custom filter was created using the built-in nodes used to generate camera effects in Blender. Form these nodes the output image was normalized and brought from RGB to
monochrome, these depth maps were able to convey correct scaling as well as having a smoother gradient to them allowing for better reading from the algorithm.

![Figure 12: Blender Depth Maps (Left) and Scripted Depth Maps (Right)](image)

One of the tests done to ensure that the depth maps were conveying the correct values for depth, the maps was transferred over to Printer Font Metrics files (pfm). This file format allows for complex data to be transferred efficiently. The pfm files were then viewed using a custom Python script to ensure that the correct depth data was being fed into the network. The custom Python script reads in the pfm and then plots the data into a contour map that shows the values in meters from the camera. For the remainder of the tests the distance to the camera was a consistent 45 meters away when the satellite was facing straight on the camera. Figure 13 shows the pfm data contour plots. As one can observe, the contour plots display accurate depth data which ensures that the depth maps are now accurate and able to be run through the network. The closer the lines are in the contour plot the more condensed the image is in that area meaning that since the lines of the contour are clustered on the front most solar panel the depth maps are accurate in stating that the front solar panel is the closest object to the camera.
Before testing was done with the SmallSat dataset, the TransMVSNet algorithm was tested on a preset dataset called DTU. This DTU dataset came complete with depth maps and camera extrinsic and intrinsic data. This was tested and trained on the algorithm and resulted in the mesh in Figure 14. This mesh was used to test the capabilities of the algorithm. After ensuring that the TransMVSNet network was functional, the SmallSat dataset was run through the algorithm. However, there was not a single test that resulted in a 3D model. The hypothesis is that the nature of the SmallSat dataset was static but since the dataset was placed on a black background the satellite was the only item with key points. This means that the algorithm was trying to match the black background in the mesh and was treating the background as an item to model as well as the satellite. This means that the TransMVSNet algorithm would not be a suitable network for space applications. The best results for the SmallSat dataset are also shown in Figure 14 alongside the result from the
DTU test. As one can observe, the center of the disk-like mesh that was produced from the SmallSat testing is the satellite mesh while the outside of the disk remains the black background. This means that the algorithm was trying to key point match the background as well. For most static 3D reconstruction algorithms, the background of the scene also has to move to allow for proper key-point matching. Without the moving background, the scene cannot be properly placed in space and the mesh fails. This means that the TransMVSNet algorithm would never work in a space scenario.

![Figure 14: DTU (Right) Comparison to SmallSat (Left) for TransMVSNet](image)

Through multiple experiments with the TransMVSNet algorithm, going through every possible data point and structure. The conclusion is that this algorithm can create high-fidelity meshes of scenes. However, for a space scenario, this network would not be able to handle the lack of background. Furthermore, this algorithm would not work in a space scenario as well due to the system being mostly dynamic. Meaning that a dynamic algorithm would need to be implemented eventually. However, in the case of this study, both of the networks tested are static algorithms.
2.2 Instant NeRF

The second algorithm investigated in this research embodies cutting-edge methodologies for reconstructing 3D scenes, known as neural radiance fields (NeRF). NeRF's premise revolves around projecting a ray from each camera through a pixel into the scene, subsequently integrating density and color data along the ray in predetermined increments. Leveraging spatial coordinates (x, y, z) and the camera's orientation—pitch and yaw—NeRF orchestrates a sophisticated integration process, meticulously capturing scene dynamics. This data serves as the foundation for NeRF's built-in Convolutional Neural Network (CNN) to estimate scene color and density, facilitating concise training and modeling from video input.

Expanding upon this, the Neural Radiance Fields (NeRF) algorithm represents a pinnacle of sophistication in 3D scene reconstruction. Operating on the principle of ray casting, NeRF seamlessly projects rays from each camera pixel into the scene, initiating an intricate integration of the radiance field along the ray at specified intervals. Harnessing spatial coordinates (x, y, z) and camera orientation (pitch and yaw), NeRF's fusion of spatial and directional data enables its built-in Convolutional Neural Network (CNN) to discern scene color and density with unparalleled precision. This holistic methodology empowers NeRF to adeptly model and train from video input, laying the groundwork for the precise reconstruction of dynamic scenes with exceptional fidelity. Ray casting begins with computing a ray direction vector $r$, given a pixel's location $(u, v)$ in the image plane, utilizing the camera's intrinsic and extrinsic parameters. This ray is represented as equation 6.

$$r(t) = o + td$$  \hspace{1cm} (6)$$

Where $o$ denotes the ray origin (the camera center), $d$ is the normalized ray direction, and $t$ represents a parameter that defines the distance along the ray from the ray origin $o$ (the camera center). By varying the parameter $t$ the integration can move along the ray $d$. NeRF’s ray integration involves approximating an integral along the ray to estimate radiance values at discrete points. This approximation is represented as a sum over discrete steps along the
ray, incorporating accumulated density, step lengths, density, and color values at each step. The radiance and density values along the ray are estimated by a neural network, given a query point along the ray, which takes as input the point’s spatial coordinates and camera orientations. The neural network is trained using supervised learning with input-output pairs of spatial coordinates and camera orientations, and ground-truth radiance and density values obtained from training images. By amalgamating these mathematical principles with the algorithmic process, NeRF demonstrates exceptional capabilities in reconstructing 3D scenes from 2D images, heralding advancements across diverse fields such as computer vision, graphics, and virtual reality. The ray integration equation employed by NeRF is shown in equation 6.

\[ C(r) = \sum_{i=1}^{N} T_i (1 - \exp(-\sigma_i \delta_i)) c_i \quad \text{where} \quad T_i = \exp(-\sum_{j=1}^{i-1} \sigma_j \delta_j) \quad (6) \]

The variables represent specific aspects of the reconstruction process. \( N \) is the number of bins. A bin refers to a discrete interval or partition of the input space meaning a certain quantized amount of input. \( \sigma_i \) is the density of the \( i^{th} \) bin, \( \delta_i \) is the width of the \( i^{th} \) bin, and \( c_i \) is the color of the \( i^{th} \) bin. This whole equation is used to estimate the expected color \( C(r) \) when tracing along ray \( r \) [13]. The expression being multiplied by \( T_i \) is the probability the ray hits somewhere along the \( i^{th} \) bin independence of the previous collisions, this can be derived from discrete Poisson distribution. The expression of \( T_i \) approximates Beer’s law, or the probability to not collided with any previous bins.

The structure of the Instant NeRF neural network is intricately designed to facilitate the precise reconstruction of 3D scenes with remarkable accuracy and fidelity. Operating on a foundation of advanced machine learning principles, Instant NeRF leverages a convolutional neural network (CNN) architecture to discern intricate details and nuances within captured imagery. The neural network’s architecture comprises multiple layers, each meticulously engineered to extract and analyze features essential for scene reconstruction. At the heart of the Instant NeRF neural network lies its ability to seamlessly integrate spatial coordinates and camera orientations into the reconstruction process. This integration enables
the network to effectively capture the geometric layout of the scene, facilitating precise depth estimation and volumetric reconstruction. By incorporating spatial and directional data, Instant NeRF ensures that every aspect of the scene is faithfully represented in the reconstructed model. This structure of the network also allows for the integration of a small convolutional neural network for the main neural network for the reconstruction. Using this smaller network allows for faster processing time and mesh generation. This CUDA version also allows for higher resolution from the output. The paper for Instant NeRF compares their network output to that of gigapixel representation. “ACONS have shown impressive results when fitting very large images—up to a billion pixels—with high fidelity at even the smallest scales. We target our multiresolution hash encoding at the same task and converge to high-fidelity images in seconds to minutes” [2].

Furthermore, Instant NeRF's neural network is trained using supervised learning techniques, wherein input-output pairs of spatial coordinates, camera orientations, and ground-truth radiance and density values are utilized to optimize the network's parameters. This rigorous training process ensures that the network can accurately estimate scene color and density from a given viewpoint, laying the groundwork for robust and reliable scene reconstruction. In addition to its robust neural network architecture, Instant NeRF also employs sophisticated ray tracing methodologies to refine scene reconstruction. By projecting rays from each camera pixel into the scene and integrating radiance values along these rays, Instant NeRF captures intricate lighting effects and surface details with unparalleled precision. This holistic approach to scene reconstruction ensures that the final output is not only visually compelling but also faithful to the underlying scene geometry and lighting conditions.

Instant NeRF utilizes a fairly small CNN to ensure timely training. The CNN consists of a Multi-Layer Perceptron (MLP) with two hidden layers, each layer comprised of 64 neurons and a rectified linear unit (ReLU) activation function. The output layer in turn is linear. The initialization of the algorithm uses Goldfard and Bengio. The initialization can be viewed in equation 7.
\[
\text{variance} = \frac{2}{n_{in} + n_{out}}
\]  

(7)

Where \(n_{in}\) is the number of input units and \(n_{out}\) is the number of output units. This use of initialization is used to solve issues with vanishing or exploding gradients during training by ensuring the initial weights are set to appropriate values. Leading to more stable and effective training. By setting the hash table, the storage of the input data, entries to a uniform distribution \(U(-10^{-4}, 10^{-4})\) allow for randomness to be initialized while keeping initial predictions close to zero. The use of ADAM optimization is also implemented to ensure that the parameters are being updated effectively during training through the use of momentum in training.

Before embarking on testing the SmallSat dataset, it was imperative to conduct preliminary assessments of the algorithm's efficacy using a sample dataset. In the case of the Instant NeRF algorithm, a curated sample dataset featuring a video capturing the movements of a fox was provided alongside the algorithm for initial testing purposes. This dataset served as a litmus test to ensure the algorithm's functionality and integrity before scaling up to more complex datasets. The outcomes of these preliminary tests are delineated in Figure 15, offering a visual representation of the algorithm's performance. Notably, the results unveil a discernible superiority of the Instant NeRF model over its TransMVSNet counterpart in terms of fidelity and completeness of the reconstructed scene. While TransMVSNet excels in generating a mesh of the scene by selectively focusing on specific objects within a predetermined range from the cameras, Instant NeRF surpasses this limitation by recreating the entire scene, encompassing even the background objects with remarkable detail. However, it's essential to acknowledge that the efficacy of the NeRF model is contingent upon the environment maintaining static conditions, where the scene remains stationary while allowing for camera movement. This constraint underscores the need for further exploration and refinement to ensure the algorithm's adaptability to dynamic scenarios,
thereby maximizing its utility across diverse applications within the realm of 3D scene reconstruction and beyond.

![NeRF Fox Model Mesh](image)

**Figure 15: NeRF Fox Model Mesh**

The progression from initial tests to comprehensive evaluation with the SmallSat dataset marked a pivotal phase in this research endeavor. Following the preliminary assessments of the Instant NeRF algorithm, the testing and training processes transitioned to encompass the broader scope of the SmallSat dataset. Adhering to standardized guidelines akin to those employed in the TransMVSNet algorithm, both networks leveraged the COLMAP software to ascertain and estimate camera extrinsic matrices alongside camera pairs. Although the data format varied slightly from that utilized in TransMVSNet, it laid the groundwork for commencing training on the SmallSat images. Notably, the unique methodology of the NeRF algorithm obviated the necessity for depth maps, given its reliance on ray tracing and point matching systems, thus streamlining the training process by solely requiring imagery for scene reconstruction.
However, upon integrating the SmallSat dataset into the NeRF algorithm, similar challenges emerged as those encountered with preceding networks. The initial hypothesis posited that the ray tracing mechanism of the NeRF algorithm would inherently identify the satellite as the focal point within the imagery, attributed to the distinct black background simulating a space environment. Contrary to expectations, while the NeRF algorithm yielded superior results compared to the TransMVSNet counterpart, it still failed to produce a mesh representation of the satellite. This outcome prompted a critical reevaluation, leading to the inference that the dataset's efficacy may be compromised. This hypothesis, albeit tentative, raised pertinent questions regarding the dataset's complexity and the potential absence of intricate features within the background, thus undermining the algorithms' efficacy. With the nature of the SmallSat dataset being a three-hundred and sixty encompass of the object with a black background the NeRF algorithm struggles to reconstruct the entire scene due to similar issues the MVS systems. The struggle to place background points with respect to the space given for the object.

With time constraints looming over the research timeline, a final test was devised to validate the conjecture pertaining to the dataset's composition. Leveraging the smaller Instant NeRF algorithm for its expedited training speed and user-friendly interface, the final test was orchestrated on a novel dataset coined LabSat. This dataset entailed a comprehensive three-hundred and sixty-degree scan of an in-lab scale satellite housed within the Florida Institute of Technology Autonomy Lab. Spanning three feet and eleven inches, the satellite boasted a square frame enveloped in foil, facilitating the replication of complex textures for feature matching. Additionally, the model thruster and antenna, alongside solar panels, augmented the dataset's complexity. An image of the satellite is encapsulated in Figure 16, providing a visual reference for the ensuing analysis. This exhaustive evaluation of the LabSat dataset promises to unveil insights pivotal to elucidating the intricate dynamics underlying 3D scene reconstruction, thereby culminating this research endeavor on a note of comprehensive understanding and potential avenues for further exploration.
Taking a video of the lab satellite and then exporting two frames every second of the video left the dataset with one hundred and ten images. These images were then applied to the instant NeRF pipeline. The results were as the hypothesis stated. Due to the more complex background, a mesh was recovered from this dataset and with high fidelity. This was then tested to the limits by turning off all the lights in the lab and allowing only a singular light source coming from an artificial sun. This was done to simulate the SmallSat dataset as much as possible. These two tests are compared in Figure 17.
Figure 17: LabSat Results from Instant NeRF

Images a.1 and a.2 are reconstructions taken with a single light source, images b.1 and b.2 are with the lab lights on (ambient light)

Upon closer examination, it becomes evident that the mesh generated under the influence of a single light source exhibits a comprehensive rendition of the satellite, albeit with minor discrepancies such as the missing mesh section on the leftmost solar panel. This anomaly can be attributed to the intricate interplay of light and shadow cast by the artificial sun, which inadvertently creates highlights and shadows, thereby affecting the completeness of the mesh. Conversely, the mesh representation of the satellite's rear, as depicted in Figure 17 image a.2, remains incomplete owing to the shadows cast by the artificial sun, obscuring certain details. Contrasting this scenario, the ambient light tests showcased in Figure 17 images b.1 and b.2 yield meshes that are perceptibly complete, albeit lacking the nuanced
intricacies observed in the single light source scenario. This indicates that the inspector camera should have a light source associated with it to decrease this noise found in image a.2. By incorporating a light source with the inspector camera, shadows cast by a single light source, like the sun, will be a non-factor. Notably, the software utilized for mesh visualization facilitates meticulous measurements, thereby enabling a comparative analysis of the accuracy between the two lighting conditions. Span measurements derived from these assessments reveal a substantial disparity, with the ambient light source yielding an estimated measurement of 2.43 feet, whereas the single light source yields a measurement of 3.97 feet. This stark contrast underscores the inherent inaccuracies associated with the ambient light source, exhibiting a 37.94% error compared to the 1.38% error observed in the single light source scenario. Despite the incomplete mesh, the single light source scenario affords a more detailed representation of the satellite, coupled with a comparatively accurate span measurement. This elucidates the algorithm's efficacy in generating high-fidelity meshes of space objects, thereby accentuating its potential utility in diverse applications within the realm of space exploration and beyond.
Chapter 3
Conclusion and Recommendations

3.1 Conclusion

In both of the networks explored, one network produced acceptable results. The instant NeRF algorithm was able to produce high-fidelity, high resolution and true to form meshes of a spacecraft. This was also done at a fraction of the training time as the TransMVSNet algorithm as shown in the table below comparing the results of the research.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Train Time</th>
<th>Depth Maps</th>
<th>Complete Mesh</th>
</tr>
</thead>
<tbody>
<tr>
<td>TransMVSNet</td>
<td>12-16 hours</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Instant NeRF</td>
<td>0.5-1 hours</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

While the TransMVSNet algorithm has demonstrated its effectiveness in generating high-fidelity meshes of scenes and capturing intricate details in the completed reconstructions, its limited versatility poses a challenge when it comes to handling more complex imagery. Specifically, this limitation becomes evident when attempting to reconstruct space bodies within a realistic space environment, complete with dynamic lighting effects. The algorithm's static nature inhibits its ability to adapt to the complexities inherent in such scenes, thereby restricting its applicability.

In contrast, a more sophisticated approach, such as Instant NeRF, which harnesses radiance fields and employs ray tracing techniques, showcases a greater capacity to express and reconstruct complex scenes. By leveraging these advanced methods, Instant NeRF is capable of capturing the intricate interplay of light and shadow, thus enabling the faithful
representation of dynamic environments. However, it's worth noting that even with Instant NeRF's enhanced capabilities, the successful reconstruction of highly dynamic scenes is contingent upon the complexity of the background and the movement of the camera.

Despite the advancements represented by both TransMVSNet and Instant NeRF, their inherent static nature introduces certain limitations and errors in the reconstruction process. A notable constraint lies in the absence of a dynamic framework, which would undoubtedly yield superior and more consistent results. The implementation of a dynamic system holds the promise of mitigating these limitations, allowing for greater adaptability and accuracy in capturing the nuances of dynamic scenes. Thus, the exploration of dynamic methodologies stands as a crucial avenue for further enhancing the efficacy and reliability of 3D reconstruction algorithms in diverse and dynamic environments.

3.2 Recommendations

As the research continued, more information was gathered on the topic of reconstruction of scenes from imagery. There are many recommendations as to how the continuation of this research can be explored. Firstly, a more complex dataset with more expressions on the limitations of space imagery can be explored. These topics include more realistic lighting with imperfections in the imagery such as lens flares, and spotty or grainy imagery from low-light images. Different types of imagery can also be expired from thermal to strict depth-related imagery to investigate if those are better alternatives for reconstruction than a typical camera system. There are many avenues to explore in the imagery process that would drastically improve the functionality of these networks as the resulting meshes are only ever as high fidelity as the input imagery.

Secondly, investigation into a dynamic model can also be beneficial. There exist dynamic models for NeRF algorithms that consider the time step in which the image was taken in sequence with the other images in the dataset. This would allow for images to be taken in a “fly-by” manner from another satellite. Allowing for a more realistic scenario in
which this research can be applied. Another avenue is the reduction of allotted memory these algorithms use. The reconstruction of scenes requires complex hardware with memory storage to allow for these complex triangulations to occur and the microprocessor that a neural network does to reconstruct a scene. Exploration into how these networks can be decreased in size with minimal reflection of the output mesh integrity can also improve the feasibility of the implementation of an architecture such as this on a real mission.

This research explored the topic of computer vision and machine learning and their feasibility of implementing a reconstruction algorithm onto a space mission. Limitations and results show that this type of implementation is possible but would require more in-depth topics and computing to make it into a real fully functional system.
References


Appendix

Custom Depth Map Maker Code (Python)

```python
import cv2
import numpy as np
import os
import struct

def create_depth_map(image_left_folder, image_right_folder, camera_matrix_left,
camera_matrix_right, baseline, output_folder, start_indexes):
    if not os.path.exists(output_folder):
        os.makedirs(output_folder)

    left_image_files = sorted(os.listdir(image_left_folder))

    name_index = 0

    for start_index in start_indexes:
        end_index = start_index + 24
```

image_files_range = left_image_files[start_index:end_index]

if start_index == 0:
    baselineCam = baseline[0]
else:
    baselineCam = baseline[1]

for i in range(len(image_files_range)):
    left_image_path = os.path.join(image_left_folder, image_files_range[i])
    if i == 22:
        right_image_path = os.path.join(image_right_folder, image_files_range[0])
    elif i == 23:
        right_image_path = os.path.join(image_right_folder, image_files_range[1])
    else:
        right_image_path = os.path.join(image_right_folder, image_files_range[i+2])

    img_left = cv2.imread(left_image_path)
    img_right = cv2.imread(right_image_path)
gray_left = cv2.cvtColor(img_left, cv2.COLOR_BGR2GRAY)

gray_right = cv2.cvtColor(img_right, cv2.COLOR_BGR2GRAY)

# Rectify the images using the camera matrices

R1, R2, P1, P2, Q, _, _ = cv2.stereoRectify(
    cameraMatrix1=camera_matrix_left,
    cameraMatrix2=camera_matrix_right,
    distCoeffs1=None,
    distCoeffs2=None,
    imageSize=(img_left.shape[1], img_left.shape[0]),
    R=np.eye(3),  # Identity rotation matrix
    T=np.array([baselineCam, 0, 0]),  # Baseline translation
    flags=cv2.CALIB_ZERO_DISPARITY,
    alpha=0
)

map1, map2 = cv2.initUndistortRectifyMap(camera_matrix_left, None, R1, P1,
                                (img_left.shape[1], img_left.shape[0]), cv2.CV_16SC2)
rectified_left = cv2.remap(gray_left, map1, map2, interpolation=cv2.INTER_LINEAR)

rectified_right = cv2.remap(gray_right, map1, map2, interpolation=cv2.INTER_LINEAR)

# Compute disparity map

stereo = cv2.StereoSGBM_create(
    minDisparity=0,
    numDisparities=16,
    blockSize=5,
    P1=8 * 3 * 5 ** 2,
    P2=32 * 3 * 5 ** 2,
)

disparity = stereo.compute(rectified_left, rectified_right)

resized_disparity = cv2.resize(disparity, (1600, 1200))

# Normalize the disparity map

normalized_disparity = cv2.normalize(disparity, None, alpha=0, beta=255, norm_type=cv2.NORM_MINMAX, dtype=cv2.CV_8U)

# Resize
resized = cv2.resize(normalized_disparity, (1600, 1200))

print(disparity.shape)
print(resized_disparity.shape)

# Name Files

file_index = str(name_index).zfill(5)
output_filepng = f"depthvisual_{file_index}.png"
output_filepfm = f"depthmap_{file_index}.pfm"

# Save the depth map

output_pathpng = os.path.join(output_folder, output_filepng)
cv2.imwrite(output_pathpng, resized)

# Optionally, save the disparity map in PFM format

pfm_file = os.path.join(output_folder, output_filepfm)
# write_pfm(pfm_file, resized_disparity)
cv2.imwrite(pfm_file, disparity)
print(f"Depth map created and saved for {image_files_range[i]}")

name_index += 1

print("Depth map creation completed!")

# def write_pfm(filename, data):

#     # with open(filename, 'wb') as f:

#         # PFM header

#             header = "PF\n%d %d\n%d\n" % (data.shape[1], data.shape[0], -1)

#         f.write(header.encode('utf-8'))

#     # PFM data

#         data.tofile(f)

def write_pfm(filename, depth_map):

    
    Write a grayscale depth map to a PFM file.

    Parameters:

    - filename: The path to the output PFM file.
- depth_map: A 2D numpy array representing the depth map.

```python
height, width = depth_map.shape
scale = -1.0 / np.max(depth_map)  # Scale factor for PFM format

with open(filename, "wb") as file:
    # Write the PFM header
    file.write(b'PF

file.write(f'{width} {height}

file.write(f'{scale}

# Write the data as binary

depth_map = np.flipud(depth_map)  # PFM stores the rows in reverse order

data = depth_map.astype(np.float32)
data.tofile(file)

# Example usage

image_left_folder = "C:\Users\AutonomyLab\Desktop\DataSet\Camera Right"
```
image_right_folder = "C:\\Users\\AutonomyLab\\Desktop\\DataSet\\Camera Left"

output_folder = "C:\\Users\\AutonomyLab\\Documents\\STTR-Creare\\TransMVSNet\\datasets\\SmallSat_Training\\Depths_raw_made2\\scan1_train"

## Camera intrinsic parameters for left and right cameras

scale1 = 1 \#800/960

scale2 = 1 \#600/540

camera_matrix_left = np.array([[2666.667*scale1, 0, 960*scale1], [0, 2250*scale2, 540*scale2], [0, 0, 1]], dtype = float)

camera_matrix_right = np.array([[2666.667*scale1, 0, 960*scale1], [0, 2250*scale2, 540*scale2], [0, 0, 1]], dtype = float)

baseline1 = 45 * 15 * (180/3.14) \# Provide the baseline value

baseline2 = 46.57 * 15 * (180/3.14)

baseline = [baseline1, baseline2]

# Specify the starting indexes for reading images

start_indexes = [0, 24, 48]

create_depth_map(image_left_folder, image_right_folder, camera_matrix_left, camera_matrix_right, baseline, output_folder,
Camera Extrinsic Checker and Plotter (Python)

```python
import re
import os
import numpy as np
import quaternion
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Function to convert rotation matrix to quaternion

def matrix_to_quaternion(matrix):
    return quaternion.from_rotation_matrix(matrix[:3, :3])

# # Directory containing the files

# directory = r'C:\Users\AutonomyLab\Documents\STTR-Creare\TransMVSNet\datasets\SmallSat_Training\Cameras\train' # What is training now
```
# directory = r'C:\Users\AutonomyLab\Documents\STTR-Creare\TransMVSNet\outputs\SmallSat_ThisIsIt\scan1\cams'  # ThisIsIt

# directory = r'C:\Users\AutonomyLab\Documents\STTR-Creare\TransMVSNet\outputs\SmallSat_Zpoint\scan1\cams'  # Donut

# directory = r'C:\Users\AutonomyLab\Documents\STTR-Creare\TransMVSNet\outputs\SmallSat_ThisIsIt2\scan1\cams'  # three rings

directory = r'C:\Users\AutonomyLab\Documents\STTR-Creare\TransMVSNet\datasets\DTU_Training\Cameras'  # DTU Training

# directory = r"D:\casmvsnet\cams"

# Get a list of files in the directory

file_list = sorted([f for f in os.listdir(directory) if re.match(r'd+_cam\..txt', f)])

# Coordinates of the single point

#single_point = np.array([45, 0, 0])

fig = plt.figure()

ax = fig.add_subplot(111, projection='3d')

# Set the desired scale to make the vectors longer

scale = 18
counter = 0

for file_name in file_list:
    file_path = os.path.join(directory, file_name)

    with open(file_path, 'r') as file:
        file_content = file.read()

    matrix_strings = re.findall(r'extrinsic\n.*?\nintrinsic', file_content, re.DOTALL)

    rotation_matrices = []
    translations = []

    for matrix_str in matrix_strings:
        matrix_values = matrix_str.strip().split('n')
        rotation_matrix = [list(map(float, row.split()[:3])) for row in matrix_values]
        translation = [float(matrix_values[0].split()[3]), float(matrix_values[1].split()[3]),
                       float(matrix_values[2].split()[3])]

    # Process rotation_matrices and translations as needed
rotation_matrices.append(np.array(rotation_matrix))
translations.append(np.array(translation))

if translation[0] > 0:
    print("X is not Negative": file_name)

quaternion_list = \[matrix_to_quaternion(matrix) for matrix in rotation_matrices\]

break_iteration = 90
for i, (quat, translation) in enumerate(zip(quaternion_list, translations)):
    if counter >= break_iteration:
        break

rotation_matrix = quaternion.as_rotation_matrix(quat)

#rotation_matrix = rotation_matrix.transpose()

# Orthogonal
if rotation_matrix.shape != (3, 3):
    print("False, Not square")
if not np.allclose(np.dot(rotation_matrix, rotation_matrix.T), np.eye(3)):
    print("False, transpose is not inverse")

# Check if the determinant is approximately +1
det = np.linalg.det(rotation_matrix)
if not np.isclose(det, 1.0):
    print("False, determinate not +1")

print("True")

translation = np.matmul(rotation_matrix, translation)

# Plot the x-axis
ax.quiver(translation[0], translation[1], translation[2],
          rotation_matrix[0, 0], rotation_matrix[0, 1], rotation_matrix[0, 2],
          color='r', length=scale)

# Plot the y-axis
ax.quiver(translation[0], translation[1], translation[2],

        rotation_matrix[1, 0], rotation_matrix[1, 1], rotation_matrix[1, 2],
        color='g', length=scale)

# Plot the z-axis
ax.quiver(translation[0], translation[1], translation[2],

        rotation_matrix[2, 0], rotation_matrix[2, 1], rotation_matrix[2, 2],
        color='b', length=scale)

# Plot the single point
#ax.scatter(single_point[0], single_point[1], single_point[2], color='k', s=50)

counter += 1

# ax.axes.set_xlim3d(-50, 50)
# ax.axes.set_ylim3d(-50, 50)
# ax.axes.set_zlim3d(-50, 50)

ax.set_xlabel('X')
ax.set_ylabel('Y')

ax.set_zlabel('Z')

ax.set_title('Coordinate Orientations for Cameras')

plt.show()

Blender Output to COLMAP Database Code (Python)

import os

def reformat_quaternion_data(input_folder, filename, output_file, image_indexes, num_images):
    output = []

    index_number = 1

    with open(filename, 'r') as file:

        lines = file.readlines()

        for index in image_indexes:

            for i in range(index, index + num_images):

                line = lines[i]

                if not line.strip():
continue

parts = line.strip().split(':

quaternion_data = parts[2].strip().split(',

qw, qx, qy, qz = quaternion_data

# Generate the image filename based on the index
image_filename = f"image_{(index_number

with leading zeros

image_path = os.path.join(input_folder, image_filename)

# Create reformatted data line with the image name
reformatted_data = f"{index_number} {qw} {qx} {qy} {qz} 26 -30 -6 1 {image_filename}\n"

output.append(reformatted_data)

index_number += 1

with open(output_file, 'w') as outfile:
    outfile.write('n'.join(output))

# Usage example:
input_folder = 'C:\Users\AutonomyLab\Desktop\Database'

filename = 'quaternions.txt'

output_file = 'images.txt'

image_indexes = [0, 504, 1008]

num_images = 72

reformat_quaternion_data(input_folder, filename, output_file, image_indexes, num_images)

**Depth Map Maker from input Images [MVS] (Python)**

# Depth Map Maker

# Timothy Huber

# May 18th 2023

import os

import pandas as pd

import numpy as np

import math

import re

import cv2

from statistics import mode
import shutil

from scipy.spatial.transform import Rotation

# File Directories

colmapFile = "C:\Users\AutonomyLab\Desktop\colmapOutputs\"
# Directory for Colmap outputs

trainFile = "train.json"  # Directory for Train.json [input]

pairFile = "PairData.txt"  # Directory for pairs data log [written in code]

pairRectFile = "PairDataRect.txt"  # Directory for new pairs data after rectification and easing of bad pairs

speedFile = "C:\Users\AutonomyLab\Desktop\667\667_Export\images\"
# Directory for SPEED+ images [input]

recLargeFile = "C:\Users\AutonomyLab\Desktop\Rectified_Large\"
# Directory for large Rectified images [made in code]

recTransFile = "C:\Users\AutonomyLab\Documents\STTR-Creare\" \
    "TransMVSNet\datasets\SPEED_Training\667\Rectified\scan1_train\"
# Directory for rectified images for transMVSNet [made in code]

depthFile = "C:\Users\AutonomyLab\Documents\STTR-Creare\" \

# Directory for depth maps for TransMVSNet [made in code]

"\TransMVSNet\datasets\SPEED_Training\667\Depths_raw\scan1_train\"

# Read Train.json

readFile = pd.read_json(trainFile ,lines=False)

x = readFile.values.tolist()

# Loop for data pulling

List = readFile.values.tolist()

nameArray = []

quartArray = np.zeros([len(x),4],list)

transArray = np.zeros([len(x),3],list)

for i in range(len(readFile)):

    # splitting up the index

    line = List[i]

    imageName = line[0] # str

    trans = np.array(line[1], float) # list to array

    quart = np.array(line[2], float) # list to array
# restructuring
	namenameArray.append(imageName)

	quartArray[i,:]=quart

	transArray[i,:]=trans

# Finding Pairs Block

	pairs = []

	pairMatrix = []

	minValues = []

	minAngles = []


dataLength = len(os.listdir(speedFile))  # Length of Training data scope

relativeDistances = np.zeros(dataLength,float)

angle = np.zeros(dataLength,float)

for i in range(dataLength):

	for j in range(dataLength):
relativeDistances[j] = np.linalg.norm(transArray[j, :] - transArray[i, :])  # Find the minimum difference in translation data (Translation from SpaceCraft to Camera)

r1 = Rotation.from_quat(quartArray[i, :])  # The rotation of the baseline camera

r2 = Rotation.from_quat(quartArray[j, :])  # The rotation of the second camera

relativeRotation = r1.inv() * r2  # Find the rotation between the two cameras

angle[j] = relativeRotation.magnitude()  # Find the angle from rotation (Rads)

if i == j:
    relativeDistances[j] = float('NaN')  # Remove itself
    angle[j] = float('NaN')  # Remove itself

minValue = min(relativeDistances[~np.isnan(relativeDistances)])

indexArray = np.where(relativeDistances == minValue)  # Returns tuple

index = indexArray[0].item()
minAngle = min(angle[~np.isnan(angle)])

indexArrayAngle = np.where(angle == minAngle) # Returns tuple

indexAngle = indexArrayAngle[0].item()

pairs.append(i), pairs.append(index), minValues.append(minValue), minAngles.append(minAngle)

if i == 0:
    pairMatrix = np.array([i, indexAngle], int)
else:
    pairMatrix = np.vstack((pairMatrix, np.array([i, indexAngle], int)))

# Exporting Data Block

with open(pairFile, 'w') as f:
    for i in range(len(pairMatrix)):  # Outputting pairs
        f.write(str(i) + ', ' + str(pairMatrix[i, 0]) + ', ' + str(pairMatrix[i, 1]) + '
')
and other data into a .txt file called pairsData.txt
inputName = "indexs: (" + str(pairMatrix[i][0]) + "," + str(pairMatrix[i][1]) + "): " + nameArray[pairMatrix[i][0]] + "," + nameArray[pairMatrix[i][1]] + " Baseline: " + str(minValues[i])

f.write(inputName)

f.write('n')

f.close

# # Rectified_Large Block (Dont have to run this everytime)

# def rectify_images(image_path1, image_path2):

#     image1 = cv2.imread(image_path1)                                                               # Load the images
#     image2 = cv2.imread(image_path2)

#     gray1 = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY)  # Convert the images to grayscale
#     gray2 = cv2.cvtColor(image2, cv2.COLOR_BGR2GRAY)

#     # Load the images
# sift = cv2.SIFT_create()

# keypoints1, descriptors1 = sift.detectAndCompute(gray1, None)  # Find keypoints and descriptors using SIFT

# keypoints2, descriptors2 = sift.detectAndCompute(gray2, None)

# descriptors1 = descriptors1.astype(np.uint8)  # Convert to appropriate data

# descriptors2 = descriptors2.astype(np.uint8)

# # Match keypoints using FLANN

# bf = cv2.BFMatcher()  # Find matches between keypoints in images

# matches = bf.knnMatch(descriptors1, descriptors2, k=2)

# # good_matches = []  # Filter good matches using the Lowe's ratio test
# if m.distance < 0.80 * n.distance:    # Tweak the weighting parameter for different results
#
  good_matches.append(m)
#
#
# src_pts = np.float32([keypoints1[m.queryIdx].pt for m in good_matches]).reshape(-1, 1, 2)    # Get matched keypoints' coordinates
#
# dst_pts = np.float32([keypoints2[m.trainIdx].pt for m in good_matches]).reshape(-1, 1, 2)
#
#
# F, mask = cv2.findFundamentalMat(src_pts, dst_pts, cv2.RANSAC, 3, 0.99)    # Calculate the fundamental matrix and rectification transformation
#
# _, H1, H2 = cv2.stereoRectifyUncalibrated(src_pts, dst_pts, F, image1.shape[:2], threshold=5)
#
#
# rectified_image1 = cv2.warpPerspective(image1, H1, image1.shape[:2][:,::-1])    # Rectify the images
#
# rectified_image2 = cv2.warpPerspective(image2, H2, image2.shape[:2][:,::-1])

65
#
# _, threshold1 = cv2.threshold(rectified_image1, 10, 255, cv2.THRESH_BINARY)
# Check the black and white levels of rectified images
#
# percentage_of_black1 = np.mean(threshold1 == 0)
#
# percentage_of_white1 = np.mean(threshold1 == 255)
#
# average_weight1 = np.mean(rectified_image1)
#
# percentage = (average_weight1/225) * 100
#
#_
#
# _, threshold2 = cv2.threshold(rectified_image2, 10, 255, cv2.THRESH_BINARY)
# Check the black and white levels of rectified images
#
# percentage_of_black2 = np.mean(threshold2 == 0)
#
# percentage_of_white2 = np.mean(threshold2 == 255)
#
# ref_gray = image1
#
# ssim1 = cv2.matchTemplate(ref_gray, rectified_image1, cv2.TM_CCORR_NORMED)[0][0]                 # Check rectified images based on reference initial image for filtering
#
# ssim2 = cv2.matchTemplate(ref_gray, rectified_image2, cv2.TM_CCORR_NORMED)[0][0]
#
# thresholdSim = 0.3
#
#
# con1 = (percentage_of_black1 >= 0.95 or percentage_of_white1 >= 0.20)
# con2 = (percentage_of_black2 >= 0.95 or percentage_of_white2 >= 0.20)
# con3 = (percentage_of_white1 <= 0.07 or percentage_of_white2 <= 0.07)
# con4 = (ssim1 <= thresholdSim and ssim2 <= thresholdSim)
#
# if con1 == True or con2 == True or con3 == True:
#     # Check aspect ratio and levels of black and white
#     return None, None
#
# else:
#
#     outputFolder = recLargeFile
#     # Return Rectified Filtered Images
#
#     if not os.path.exists(outputFolder):
#         os.makedirs(outputFolder)
#
#     foundation = 'P00000'  # Save the rectified images
#
#     Left = foundation[::len(str(i))] + str(i) + '_01.png'
#
#     Right = foundation[::len(str(i))] + str(i) + '_02.png'
#
#
# cv2.imwrite(outputFolder + Left, rectified_image1)
# cv2.imwrite(outputFolder + Right, rectified_image2)
#
# return rectified_image1, rectified_image2
#
# for i in range(len(pairMatrix)):
#    #print("Printing Rectified Image Number... ", i , ", ", str(len(pairMatrix)))
#    # Counter
#    image1_path = speedFile + nameArray[pairMatrix[i][0]]
#    Specify the paths to images
#    image2_path = speedFile + nameArray[pairMatrix[i][1]]
#
#    rectified_image1, rectified_image2 = rectify_images(image1_path, image2_path)
#    # call the function to rectify the images
#
### Rectified Small Block for transMVSNet
# i = 0
#
# rectifiedFolder = os.listdir(recLargeFile)
#
# for i in range(0,len(rectifiedFolder),2):
# print("Printing Rectified Small Image Number... ", int(i/2), ", ",
str(len(rectifiedFolder)/2))  # Counter

# pathLeft = recLargeFile + rectifiedFolder[i]

# pathRight = recLargeFile + rectifiedFolder[i+1]

#

# image1 = cv2.imread(pathLeft)  
# Load the images

# image2 = cv2.imread(pathRight)

#

# rectified_image1 = cv2.resize(image1, (640,512))  
# Resize the images for TransMVSNet

# rectified_image2 = cv2.resize(image2, (640,512))

# outputFolder = recTransFile

# if not os.path.exists(outputFolder):

#   os.makedirs(outputFolder)

#

#

# foundation = 'P00000'  
# Save the rectified images

# Left = foundation[:-len(str(int((i)/2)))] +str(int((i)/2)) + '_01.png'
# Right = foundation[:len(str(int((i)/2)))] + str(int((i)/2)) + '_02.png'

#

# cv2.imwrite(outputFolder + Left, rectified_image1)

# cv2.imwrite(outputFolder + Right, rectified_image2)

# Make Depth Maps

outputFolder = depthFile

rectifiedFolder = os.listdir(recLargeFile)

for i in range(0,len(rectifiedFolder),2):
    print("Printing Depth Map Number... ", int(i/2), "/", str(len(rectifiedFolder)/2))
    pathLeft = recLargeFile + rectifiedFolder[i]
    pathRight = recLargeFile + rectifiedFolder[i+1]

    if os.path.exists(pathLeft):
        imgLeft = cv2.imread(pathLeft,0)
    else:
        print("Path Left does not exist.", pathLeft)
if os.path.exists(pathRight):

    imgRight = cv2.imread(pathRight,0)

else:

    print("Path Left does not exist.", pathRight)

max_disparity = 128
# Parameters for Depth Maps (Tweak these for different results)

window_size = 5

if imgRight.shape == imgLeft.shape:
    # Add the images together pixel-wise

    imgMerge = cv2.add(imgLeft, imgRight)

else:

    print("Error: Images have different dimensions.")

_, binary_mask = cv2.threshold(imgMerge, 1, 255, cv2.THRESH_OTSU)
# Apply binary mask

indexName = int((i)/2)
foundationMap = 'depthmap00000'  # Save the Images as pfm and png

foundationVis = 'depthvisual00000'

depthFilePNG = foundationVis[-len(str(indexName))] + str(indexName) + '.png'  # PNG

depthFilePFM = foundationMap[-len(str(indexName))] + str(indexName) + '.pfm'  # PFM

if not os.path.exists(outputFolder):
    os.makedirs(outputFolder)

depthMap = cv2.resize(binary_mask, (1600,1200))  # Resize images to fit into TranMVSNet

outputPathPNG = os.path.join(outputFolder, depthFilePNG)
outputPathPFM = os.path.join(outputFolder, depthFilePFM)
cv2.imwrite(outputPathPNG, depthMap)
cv2.imwrite(outputPathPFM, depthMap)