Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

William Eikrem

6D Synthetic Data Generation Pipeline with Digital Representation of Structured Light Sensor

Master's thesis in Mechanical Engineering
Supervisor: Lars Tingelstad
Co-supervisor: Sebastian Grans
June 2021

NTNU
Norwegian University of Science and Technology

William Eikrem

6D Synthetic Data Generation Pipeline with Digital Representation of Structured Light Sensor

Master's thesis in Mechanical Engineering
Supervisor: Lars Tingelstad
Co-supervisor: Sebastian Grans
June 2021

Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

Master's Thesis

6D Synthetic Data Generation Pipeline with Digital Representation of Structured Light Sensor

William Skeide Eikrem

2021-06-10

Preface

This report is the result of the Master's thesis for the Department of Mechanical and Industrial Engineering at the Norwegian University of Science and Technology in the field of robotics and automation.

The subject of the report stems from my interest in computer vision technologies, and the use of neural networks to solve complex problems with technology inspired by biological principles. I also firmly believe in the power of sharing knowledge, and thus it was a major motivation to potentially create a pipeline that could contribute to making datasets available for a wide range of research areas.

First of all I want to express my gratitude towards my supervisor Lars Tingelstad, for guidance throughout the research and development relating to my Master's thesis. I also want to direct my gratitude towards my co-supervisor, Sebastian Grans, for guidance through meetings and discussions about the field of synthetic data generation, as well as being a sparring partner when exploring the vast software that is blender. I also want to thank Martin Ingvaldsen for insights and introduction to many solutions regarding the theory around structured light algorithms.

Lastly I want to thank my co-students for support and a lot of memories throughout the years at NTNU.

Summary

The need for 3D datasets for use in computer vision tasks is growing with the increased use of neural networks on computer vision tasks such as 6D pose estimation and and segmentation. These neural network need large amounts of data to train models for solving the problems at hand. Traditional methods for acquiring such datasets involve manual labeling and annotation of ground truth values. These tasks are cumbersome and repetitive, and has therefore made synthetic data generation a popular approach to the dataset problem. Synthetic 3D datasets are generated through the use of computer rendered images and spatial information. The advantage of using computer generated datasets is that the ground truth can be collected directly from the rendering software used. However, an inherent problem with using synthetic datasets, is the "reality gap", which is the difference between real captured data and computer generated data.

For this report the aim is to create a pipeline process that generate synthetic data for use in 6D pose estimation for bin picking scenarios. The pipeline also seek to model realistic noise, as created by real 3D scanners, by creating a digital version of a structured light camera in Blender. The aim was to find out if such a 3D sensor could be created, and applied to the data generation pipeline.

Developing the structured light camera was done through exploring the theoretical concepts of structured light algorithms and patterns and applying them to the Blender software. The resulting pipeline was created as a Blender add-on, which contained a selectable structured light sensor modeling the properties of a real structured light scanner creating realistic noise properties and missing values.

Sammendrag

Behovet for 3D datasett for bruk i datasynsoppgaver øker med den økte bruken av nevrale nettverk på datasynsoppgaver som 6D-estimering og og segmentering. Disse nevrale nettverkene trenger store mengder data for å trene modeller for å løse. Tradisjonelle metoder for å anskaffe slike datasett innebærer manuell merking og annotering av sanne verdier. Disse oppgavene er tungvint og repeterende, og har derfor gjort syntetisk datagenerering til en populær tilnærming til datasettproblemet. Syntetiske 3D-datasett genereres ved hjelp av datarendererte bilder og dybdeinformasjon. Fordelen med å bruke datamaskingenererte datasett er at sanne vardier kan samles direkte fra visualiseringsprogramvaren som brukes. Imidlertid er et iboende problem med bruk av syntetiske datasett, " reality gap ", som er forskjellen mellom ekte data og datagenererte datasett.

For denne rapporten er målet å lage en "pipeline process" som genererer syntetiske data for bruk i 6D-posisjonsestimering for bin picking scenarier. Pipelinen skal også prøve å modellere realistisk støy, som etterligner ekte 3D-sensorer, ved å modellere en digital versjon av en strukturert lys-sensor i Blender. Målet var å finne ut om en slik 3D-sensor kunne opprettes og brukes i datagenereringsprosessen.

Utviklingen av strukturert lys-sensoren ble gjort gjennom å utforske de teoretiske konseptene som omhandler strukturert lys-algoritmer og projiserte mønstre og anvende disse i Blender programvaren. Den resulterende pipelinen ble laget til å være en add-on i Blender, som inneholder en valgbar strukturert lys-sensor som søker å modellere egenskapene til en ekte strukturert lys-scanner som skaper realistiske støyegenskaper og manglende verdier.

Contents

List of Figures

Chapter 1.

Introduction

1.1. Problem statement

The use of neural networks are getting more common and popular for solving complex problems in a wide variety of research fields and industrial applications, such as robotics. Although the problems these networks aim to solve can differ on most levels, most neural networks have one thing in common, the need for data is substantial. 6D pose estimation tasks, i.e. rotation and translation, is no exception. However, the annotation and labeling of such datasets is often more cumbersome and difficult than for traditional computer vision tasks in 2D. This report presents a pipeline process to automatically generate, annotate and label such data through the use of synthetically produced 3D datasets, specifically for the robotic industry problem of bin picking. Bin picking is the task of detecting object instances from objects in bulk, and acquiring their respective 6D poses to enable robotic grasping of said objects. With 3D cameras becoming a more available technology, it is more widely used in different applications, thus data for training is, as a consequence in large demand.

Traditional methods of acquiring usable 3D datasets for pose estimation tasks involves scanning a scene with a 3D sensor and manual labeling and annotating captured real world data with ground truth poses and instances, which is a monotonous and time consuming task. Time spent on labeling and annotating data could be better spent elsewhere, on tasks that are in need of human interaction and reflection.

For dataset generation there are two main approaches, domain randomized renders and hyper realistic renders. The hyper realistic approach tries to bridge the reality gap by generating data as close to reality as possible. With developments in ray tracing, this is made possible by having realistic lighting and object textures. Real captured data is scanned by sensors that can retrieve 3D data of good quality

with imperfections stemming from the scanning process. Most synthetic data have one common property, which is the lack of imperfections in the 3D data collected from the 3D software's internal z-buffer depth. Hence, this report tries to add to the hyper realism with the addition of a digital representation of a structured light camera. The generated data should display noise characteristics and missing data points similar to a real sensor, where for instance shadows would prevent a structured light sensor from capturing data. In combination with other randomizing parameters, the aim is to generate datasets which neural networks can be trained on and thus also take into account the effects of imperfect data at training time.

1.2. Objectives

The objectives of the report is a combination of development and research goals throughout the project. The development goals focuses on the final result of the DataPipe pipeline as a whole, and the research is focused on the possibilities of the principles utilized for the digital structured light model. This results in the following research objectives:

- Creating a pipeline process for generating textured 3D datasets for bin picking applications.
- Making the pipeline easily accessed by users, with an intuitive user input.
- Exploring the possibilities of creating a working structured light camera inside Blender.

1.3. Related work

The use of synthetic 3D data in training of neural networks is not a new concept in machine learning. However, the fast development in computer graphics give rise to new technology, which enables more realistic rendering of datasets. At the same time the need for 3D datasets are steadily increasing, as the use of 3D data for neural networks become more popular in the computer vision field.

The field of research revolving around 6D pose estimation and segmentation, is indeed active. However, the state of the art of the problem bin picking is relatively unknown, and the industry solutions performances is shrouded by secrecy as stated by R. Brégier et al. [1]. Thus, the true state of the art in pose estimation and bin picking as well as synthetic data generation is difficult to know for sure.

As mentioned, computer rendered datasets are becoming more realistic as new advancements in computer graphics are made. There is however, still a gap between

real images and rendered images. In the field of synthetic data generation, this is referred to as the "reality gap". These differences in real and computer generated data will affect the training of neural networks that are exclusively trained on synthetic data, to not encapsulate the reality domain. This in turn see the drop in performance of such models [2].

Popular approaches to dealing with the reality gap includes "domain randomization" and "hyper realism". Domain randomization, elaborated on in Section 2.4, is an approach utilized by the pipelines [3], where the images are rendered with a large degree of randomized parameters such as random positions in front of background images not connected to the objects that are selected for. This type of dataset often utilize the OpenGL rendering approach [4].

The BlenderProc pipeline [5] is a procedural pipeline generating synthetic data. This is a well developed pipeline with a lot of functionalities. Its physics based rendering approach means that the datasets generated are in line with real life lighting conditions and physics through the use of Blender's ray tracing engine Cycles [6]. Domain randomization is utilized by altering textures, surface roughness and several other variables. BlenderProc incorporates video generation capabilities, randomized object positioning and several other randomizing aspects. The pipeline emphasises that it is to be opensource to enable a community of people generating datasets, building the basis of available datasets together.

It was considered to use the tried and tested pipeline created by BlenderProc to utilize the base functionalities, only implementing extensions to the existing software. However, the BlenderProc pipeline incorporates a lot of extensive functionalities, and it was ultimately decided not to use as a base, because of the difficulty of implementing new software that could integrate with the existing.

A quite recently started initiative is the BOP challenge, which is a series of challenges trying to observe the state of the art of 6D pose estimation[7]. The BlenderPRoc pipeline has been used in combination with real data as the pipeline for the creation of synthetic data for this challenge, which is a statement to the quality of their pipeline process.

An example of a texture-less dataset is the T-LESS dataset [8], which consists of a real captured models, which are annotated with ground truth poses, after scanning. A lot of datasets from industry applications are created to be textureless, because often industrial parts, for instance bolts and nuts, have low texture variation. The depth images of this dataset is somewhat similar to what is aimed for by creating the structured light camera model inside the pipeline. The objects are rough models that show signs of being captured by real sensors.

Chapter 2.

Preliminaries

The following chapter aims to provide the reader with theoretical background knowledge of key concepts used in the report. These concepts involve an introduction to artificial neural networks, rotational transforms in three dimensional space. The Blender node system is also explained, which is actively used in the design of the DataPipe pipeline, as well as a short presentation on domain randomization for synthetic data sets. Lastly the principles of intrinsic and extrinsic camera matrices are presented and explained for use in the structured light part of the report. A large part of the preliminaries is gathered form the project thesis [9].

2.1. Artificial neural networks

Artificial neural networks are a form of machine learning algorithm. These neural networks are inspired by biology in that they mimic the way a brain operates, with several interconnected nodes called neurons. There are a lot of different types of neural networks, but a large portion of them are so called supervised learning algorithms [10]. Supervised learning is supervised in the sense that the ground truth of the data is known. The known truth is used to train the neural networks by applying an error function on the prediction at the end, and use this error to correct weights in the neurons of the network, such that the model will better estimate the actual problem the next time it is run. This changing of the weights are known as training the neural network, and is typically done on very large datasets, to account for a very large variation in the possible inputs.

An example of a simple neural network structure with 5 layers can be viewed in Figure 2.1. At each neuron, the input is weighted and the output passed on down the network structure, with the next layer taking the output of the previous as input.

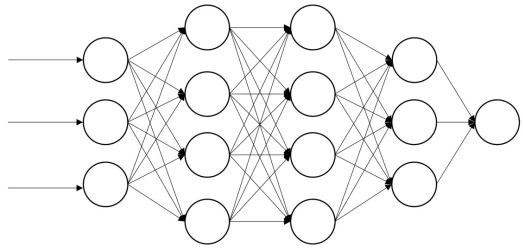


Figure 2.1.: A simple neural network structure with one input and output layer, and three hidden layers.

As mentioned, this is a simple neural network and its effectiveness is limited when working with 2D and 3D data. This data will have to be inputted as a vector where each pixel of a 2D image being an entry of that vector. This large vector is then sent in to the neural network by giving each entry of the vector to an input node, thus, loosing the spatial aspect of the image.

Overcoming this problem can be done by keeping the relation between image pixels. An example of such a neural network type is the Convolutional neural network (CNN) [11]. These networks can take 2D images or 3D point clouds as inputs and keep their spatial relations, in that they do not transform the input data into vectors. Instead they keep the data on its original form, and weight the inputs using local receptive fields as shown in Figure 2.2 for a 2D image. The CNN that is explained is for 2D images, but CNNs for 3D data is created in a similar fashion.

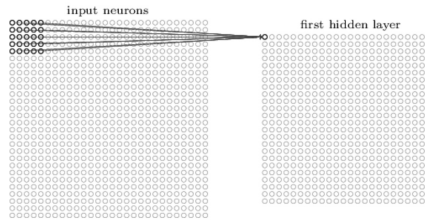


Figure 2.2.: Relation between the input layer and first hidden layer in a CNN for a 2D image. Source: [11].

The local receptive field is a small part of the image, often referred to as a kernel. For the next node in the hidden layer, the kernel is moved one step, typically one pixel, along the original image. Initially the kernel applies equal weights, or alternatively random weights, to the pixels in the convolution, such that no particular feature is selected for at first. Typically such a convolutional layers consist of several different feature maps, resulting in a hidden layer as in Figure

2.3, where the convolutional layer has 3 feature maps. In this case the neural network can activate for three different feature types. The kernel is the one determining what features that is searched for, and a typical kernel could for example be a gradient kernel, which will detect edges in an image as a feature.

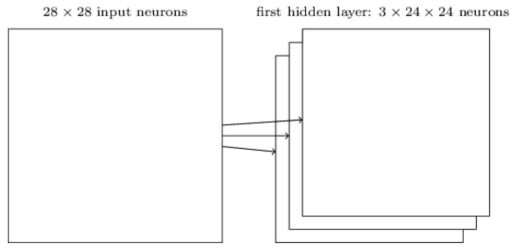


Figure 2.3.: Convolutional hidden layer consisting of 3 differt feature maps. Source: [11]

Typically there is a pooling layer following the convolutional layer, which means selecting between the nodes in the convolutional network based on some criterion. A common pooling layer is max-pooling, where some small region in the convolutional output is considered at a time. Max-pooling involves selecting the the node in this small region of the convolutional layer with the largest output, meaning the most considerable match to the feature selected for. This reduces the size of data considerably, but at the same time losing the exact position of the feature in the process.

The last layer is typically a fully connected layer, for tasks such as pose estimation, where the output of the fully connected layer is evaluated based on some error metric, compared to the ground truth. By using back-propagation, this error is used to correct the weights, in the form of kernels, of the network, and thus the neural network gets progressively better at modeling the problem.

As mentioned previously, the same concept is used for 3D CNNs, just with a three dimensional kernel moving in a large three dimensional matrix. This has, however until quite recent been impossible due to the dimensionality of such a 3D representation. But using a sparse representation of point clouds using a Minkowski Engine [12], where most of the entries become zero.

2.2. Coordinate frames and transformations

A rigid-body object's pose in 3D has 6 degrees of freedom, its position has three degrees of freedom and the rotation has three degrees of freedom. The positional argument of the pose can be represented by its x, y and z coordinates in space.

For the rotation there exists a number of different representations of rotation. In this report we use the XYZ-euler angles, rotation matrices and quaternion representation. The material used to compose this section about 3D rotations and transforms are gathered from textbooks and well respected papers [13], [14], [15]

2.2.1. Rotation matrix

The 3D rotation matrix represents a rotation between two coordinate frames, and consists of three unit column vectors. The elements in the rotation matrix is represented as follows

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}. \tag{2.1}$$

Since the rotation of a rigid body 3D object has three degrees of freedom, only three of the nine entries in the rotation matrix R can be chosen independently. This means that R must have six constraints. Three of the constraints comes from the three column vectors being unit vectors, and the last three constraints states that the vectors have to be orthogonal to each other. This can be expressed mathematically as

$$\begin{aligned} r_{11}^2 + r_{21}^2 + r_{31}^2 &= 1, \\ r_{12}^2 + r_{22}^2 + r_{32}^2 &= 1, \\ r_{13}^2 + r_{23}^2 + r_{33}^2 &= 1. \end{aligned} \tag{2.2}$$

And,

$$\begin{aligned} r_{11}r_{12} + r_{21}r_{22} + r_{31}r_{32} &= 0, \\ r_{11}r_{13} + r_{21}r_{23} + r_{31}r_{33} &= 0, \\ r_{12}r_{13} + r_{22}r_{23} + r_{32}r_{33} &= 0. \end{aligned} \tag{2.3}$$

A shorter notation for the constraints is that the rotation matrix has to fulfill the following equation,

$$R^{\mathrm{T}}R = I. \tag{2.4}$$

From this constraint it also follow that the inverse of rotation matrices R^{-1} is equal to its transpose R^{T}.

Further limiting the coordinate frames to be only right handed frames, introduces one last constraint where the determinant of R is equal to one,

$$\det(R) = 1. \tag{2.5}$$

The 3×3 rotation matrices adhering to these constraints forms the special orthogonal group $SO(3)$.

Using a rotation matrix to rotate a frame by some angle θ about a unit vector $\mathbf{w}$, taking the frame $\{a\}$ to frame $\{b\}$ as in Figure 2.4, is written as

$$\mathrm{Rot}(\mathbf{w}, \theta). \tag{2.6}$$

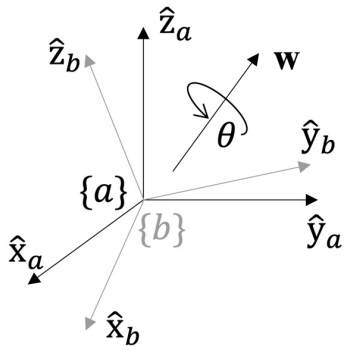


Figure 2.4.: Rotation of frame a to frame b, by an angle θ about the unit vector $\mathbf{w}$. Adapted from [14].

The rotation matrix Rot($\mathbf{w}$,θ), can then be written as R_{ab}, where the subscript indicates the rotation is applied to go from frame a to frame b. This rotation matrix, describing the rotation about the vector $\mathbf{w}$ can be described using the Rodrigues' formula

$$R_{ab} = \mathrm{Rot}(\mathbf{w}, \theta) = I + \sin\theta[\mathbf{w}] + (1 - \cos\theta)[\mathbf{w}]^2. \tag{2.7}$$

where I. is the identity matrix and $[\mathbf{w}]$ is the 3×3 skew symmetric matrix of the unit vector. The skew symmetric representation of a vector, represented as

$$[\mathbf{w}] = \begin{bmatrix} 0 & -w_3 & w_2 \\ w_3 & 0 & -w_1 \\ -w_2 & w_1 & 0 \end{bmatrix}. \tag{2.8}$$

2.2.2. XYZ-Euler angle representation

The rotation of a rigid body object can also be expressed as Euler angles, which can be represented as three rotation matrices, describing rotations about the three basis vectors **x**, **y** and **z**. The order of the rotations about these vectors can be altered, but in this section XYZ-Euler angles are used. The rigid body is then first rotated about its **x**-axis, before then rotation about its new **y**-axis, and lastly about its **z**-axis

The Euler angle representation takes three arguments in the form of three angles, α about the **x**-axis, β about the **y**-axis and γ about the **y**-axis. From Rodrigues' equation from equation (2.7) we can get the the three rotation matrices for each of the basis vectors, by substituting **w** with each of the vectors, and inserting their corresponding angles for θ. This gives us the three rotation matrices

$$\begin{aligned} \mathrm{Rot}(\mathbf{x}, \alpha) &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix}, \\ \mathrm{Rot}(\mathbf{y}, \beta) &= \begin{bmatrix} \cos\alpha & 0 & \sin\alpha \\ 0 & 1 & 0 \\ -\sin\alpha & 0 & \cos\alpha \end{bmatrix}, \\ \mathrm{Rot}(\mathbf{z}, \gamma) &= \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}. \end{aligned} \tag{2.9}$$

Using these three matrices in combination, we obtain the XYZ-Euler angle representation from the expression

$$\begin{aligned} R(\alpha, \beta, \gamma) &= \mathrm{Rot}(\mathbf{x}, \alpha)\,\mathrm{Rot}(\mathbf{y}, \beta)\,\mathrm{Rot}(\mathbf{z}, \gamma) \\ &= \begin{bmatrix} c_\beta c_\gamma & -c_\beta s_\gamma & s_\beta \\ c_\alpha s_\gamma + s_\alpha s_\beta c_\gamma & c_\alpha c_\gamma - s_\alpha s_\beta s_\gamma & -s_\alpha c_\beta \\ s_\alpha s_\gamma - c_\alpha s_\beta c_\gamma & s_\alpha c_\gamma + c_\alpha s_\beta s_\gamma & c_\alpha c_\beta \end{bmatrix}, \end{aligned} \tag{2.10}$$

where $\sin\alpha$ and $\cos\alpha$ is shortened to s_α and c_α, and so on, for the other angles as well.

2.2.3. Quaternion representation

In contrast to the Euler angle representation, which takes three arguments θ, α and β, the quaternion representation uses 4 parameters to define a 3D rotation. Using quaternions for rotation instead of Euler-angles or rotation matrices avoids the problem of gimbal lock. Gimbal lock occurs when two rotational axes coincide, an thus losing a degree of freedom.

A quaternion is a hypercomplex representation, where three of the four parameters are represented as complex numbers. A quaternion can be expressed as a four element vector,

$$\mathbf{q} = \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{bmatrix} = q_1 i + q_2 j + q_3 k + q_4, \tag{2.11}$$

where q_n, n = 1,2,3 and 4, are real numbers and i,j and k are imaginary units. The norm of a quaternion is

$$|\mathbf{q}| = \sqrt{q_1^2 + q_2^2 + q_3^2 + q_4^2}. \tag{2.12}$$

The quaternion conjugate is given as

$$\mathbf{q}^* = -q_1 i - q_2 j - q_3 k + q_4, \tag{2.13}$$

following from the multiplication and addition rules of complex numbers. Then, the quaternion product of $\mathbf{q}$ and its conjugate $\mathbf{q}^*$ gives the following relation,

$$\mathbf{q} \cdot \mathbf{q}^* = |\mathbf{q}|^2. \tag{2.14}$$

Following from this relation, the quaternion inverse is given by

$$\mathbf{q}^{-1} = \frac{\mathbf{q}^*}{|\mathbf{q}|^2} \tag{2.15}$$

If the quaternion is a unit quaternion it follows from the previous equation that

$\mathbf{q}^{-1} = \mathbf{q}^*$.

When using quaternions to describe three dimensional rotations, the unit quaternion elements are described by an angle θ and a unit vector $\mathbf{w}$. The quaternion can then be written as

$$\mathbf{q} = iw_1 \sin\frac{\theta}{2} + jw_2 \sin\frac{\theta}{2} + kw_3 \sin\frac{\theta}{2} + \cos\frac{\theta}{2}, \tag{2.16}$$

where $\mathbf{w}$ is the axis of rotation for the angle θ. A rotation of vector $\mathbf{x}$ about the angle and rotation vector can then be written as

$$\mathbf{y} = \mathbf{q}\mathbf{x}\mathbf{q}^{-1} \tag{2.17}$$

Where $\mathbf{x}$ and $\mathbf{y}$ is the quaternion representation with zero real parts, of three dimensional vectors. This expression, when $\mathbf{q}$ an $\mathbf{q}^{-1}$ is represented in its trigonometric form, can then be compared to the rotation matrix R given by the Rodrigues' formula in equation (2.7). This gives the folloeing relation

$$R\mathbf{x} = \mathbf{q}\mathbf{x}\mathbf{q}^{-1}, \tag{2.18}$$

which shows that the quaternion $\mathbf{q}$ represents the same rotation as the matrix R. Solving the equation for the elements of R expressed as the elements of q gives a way of converting between quaternion representation and rotation matrix representation. A rotation matrix described by the elements of $\mathbf{q}$ is then given as

$$R = \begin{bmatrix} q_1^2 - q_2^2 - q_3^2 + q_4^2 & 2(q_1q_2 - q_3q_4) & 2(q_1q_3 + q_2q_4) \\ 2(q_1q_2 + q_3q_4) & -q_1^2 + q_2^2 - q_3^2 + q_4^2 & 2(q_2q_3 - q_1q_4) \\ 2(q_1q_3 - q_2q_4) & 2(q_2q_3 + q_1q_4) & -q_1^2 - q_2^2 + q_3^2 + q_4^2 \end{bmatrix}. \tag{2.19}$$

2.2.4. Transformation matrices

The previous sections have been focusing on the rotation of a rigid body only. In order to describe both rotation and translation between two frames in three dimensional space, one can use the matrix T, which is a 4×4 matrix on the form

$$T = \begin{bmatrix} R & \mathbf{p} \\ \mathbf{0} & 1 \end{bmatrix}, \tag{2.20}$$

where R is a 3×3 matrix as in Section 2.2.1, and $\mathbf{p}$ is a vector describing a translation. Matrices on this form makes up the special euclidian group $SE(3)$, also known as homogeneous transformation matrices. These transformation matrices can be used to describe the pose of coordinate frames in space. Given three coordinate frames in three dimensional space $\{a\}$, $\{b\}$ and $\{c\}$, and the transformations T_{ab}, T_{bc} and T_{ac}, shown in Figure 2.5, one can obtain T_{ac} by the expression

$$T_{ac} = T_{ab}T_{bc}. \tag{2.21}$$

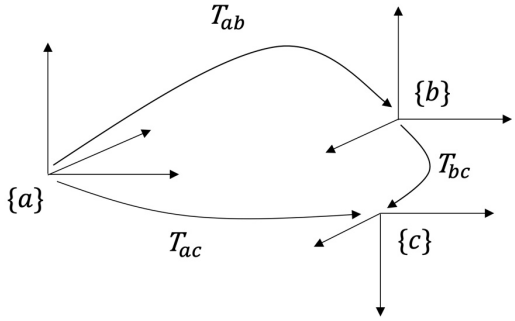


Figure 2.5.: Transformations between the frames $\{a\}$, $\{b\}$ and $\{c\}$.

Using the inverse of transformation matrices reverses the direction of the rotation and translation such that $T_{ac}^{-1} = T_{ca}$. Thus, T_{cb} can be described as

$$T_{cb} = T_{ac}^{-1}T_{ab} = T_{ca}T_{ab}. \tag{2.22}$$

2.3. Blenders node system

The Blender application incorporates a node based system for editing its objects, rendering settings, lighting, etc. In blender, two of the main application for this node system is “compositing” and “material creation”. For material creation, nodes are combined to create different textures, colours, surfaces and so on. Compositing in Blender is the modification of the rendered results that is passed as output from Blender. From compositor nodes, the rendering of the scene can be altered to for example render the depth pass of an image or change images to black and white. An example of such a node tree can be viewed in Figure 2.6. This node system is an example of node based programming, where different nodes represents actions that are carried out on different inputs or as outputs.

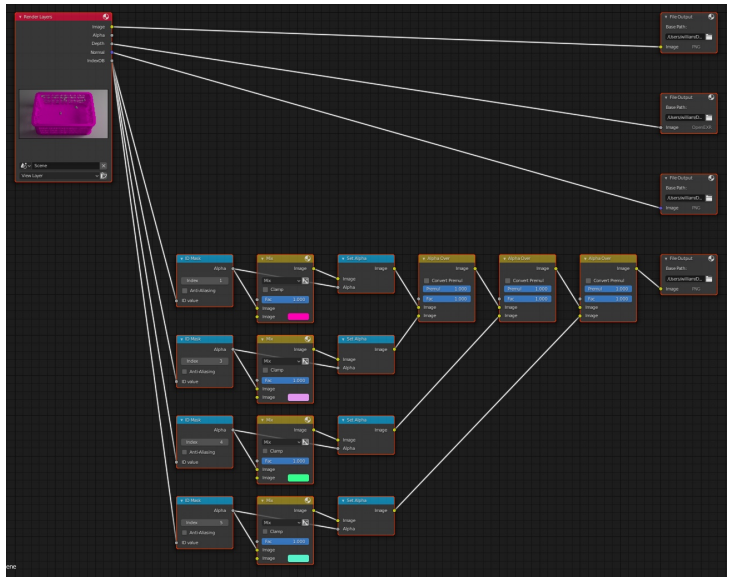


Figure 2.6.: Example of compositor node tree for rendering z-buffer, normals, RGB and masked images.

Nodes can have input sockets, output sockets or both, depending on the node type. Nodes that only have output sockets are used as inputs to other nodes in the "node tree", which is what a network of node is called in Blender. Examples of these nodes are called "input nodes", since they are used to enable the node tree to take inputs from different parts of Blender. An example of such a node could be the "RGB node", which outputs a specified RGB colour, which then can be connected to a 3D objects "material output node", to be displayed as the surface colour of the object. The material output node is then said to be the output node for the node tree.

2.4. Domain randomization

Using synthetic datasets to train neural networks that are aimed at modeling a real life problem, is not an easy task. Even though computer graphics and rendering has come a long way, the gap between artificially produces images and real images is still present. This presents problems when lighting conditions, reflections, surface textures, etc. introduces large variability in the data that is produced.

An approach to overcome this challenge is domain randomization [16]. The method of domain randomization is focused on producing large amounts of sim-

ulated variability in the training data, such as for example large variations in lighting conditions, randomizing textures, varying noise in images and so on. This is to train the neural networks on data with large variation, and trying to make the model generalize these variations when applied to the real-world data. The concept is focused on trying to encapsulate the real-world domain inside a larger domain of randomized data as illustrated by the overlapping green are in Figure 2.7.

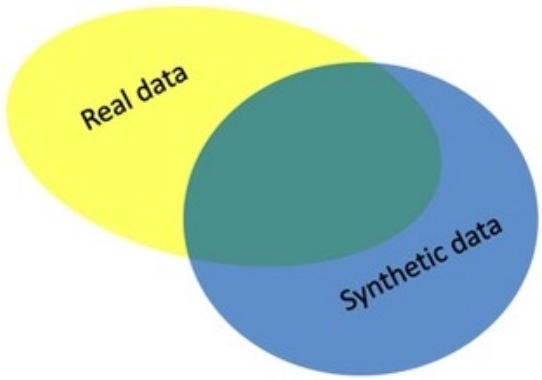


Figure 2.7.: Reality domain versus synthetic domain

There are a lot of examples on pipelines using domain randomization, for example in the article by M. Jalal et al. [3], where the generated data is used for pose recognition. For 3D data, domain randomization can involve placing objects in front of unrelated backgrounds, leaving the objects floating, seemingly in mid air, ignoring real world physics. As well as applying textures to objects that are not related to the shapes in any way, making textures metallic, roughen the surfaces or in other ways introduce randomness to these generated images.

2.5. Camera matrices and transforms

A camera can be described by the usual pinhole model , and the coordinate frames and parameters related to the model are illustrated in Figure 2.8, where the point Q is a 3D point in world coordinates.

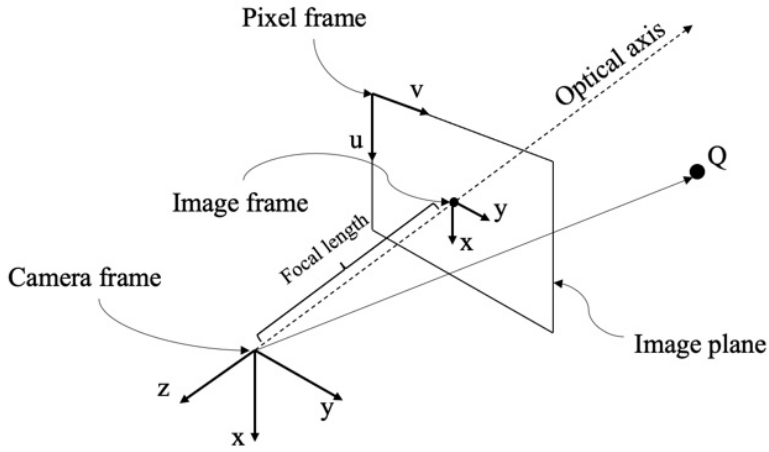


Figure 2.8.: Illustration of pin hole camera model and coordinate frames

The transformation between the point Q in homogeneous world coordinates to the a point $\mathbf{p} = \lambda[u, v, 1]^{\mathrm{T}}$ in homogeneous coordinates performed through the matrix multiplication

$$\mathbf{p} = \mathbf{K}[\mathbf{R}|\mathbf{t}]Q, \tag{2.23}$$

taking the point homogeneous 3D world coordinates to homogeneous 2D camera coordinates. The camera matrix $\mathbf{K}$ contains the camera's intrinsic parameters, and $[\mathbf{R}|\mathbf{t}]$ is the extrinsic rotation and translation of the camera in the world coordinate system. The intrinsic parameters for the camera contained in $\mathbf{K}$ is presented as the 3×3 matrix

$$\mathbf{K} = \begin{bmatrix} \rho_u & 0 & u_0 \\ 0 & \rho_v & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f & \beta & 0 \\ 0 & f\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.24}$$

where f is the focal length ρ_u and ρ_v are the pixel height and width respectively. The u_0 and v_0 is the coordinate frame shifts, which moves the image frame to the upper corner, to where the pixel frame is located. The entries α and β is the skewness parameters for the pixels. Usually the pixels are right angled and square such that α and β is set to be 1 and 0 respectively.

Chapter 3.

3D scanning and structured light algorithms

This chapter provides the reader with the theoretical concepts used throughout the report. The theory revolves around the

3.1. Scanning methods

Recovering 3D geometries through use of computer vision is an extensive field of research, with applications such as manufacturing, human-computer interaction, medicine and biology and entertainment [17]. 3D scanning, in general, can be divided in two main categories, contact- and non-contact methods. Contact methods are often very precise, but has its weaknesses. For example, the scanning speed of a contact method is limited by the speed of the measuring device on the surface of an object [18].

One of the most widely used non-contact scanning methods is structured light [19]. Structured light is a 3D scanning technique where a pattern is projected across the target object. A simple form of non-contact 3D scanning, similar in some principles to structured light, is laser line scanning [20]. The laser projects a sharp laser line across the target object, and the laser line viewed by the camera is deformed by the contours of the object. The camera then captures the image with the laser line and then triangulate between the camera's optical center and the captured laser line and the optical center of the laser, forming what is essentially a laser plane, as shown in Figure 3.1. The downside to laser line scanning is that the laser and camera have to be moved to capture the full surface and not just a cross section.

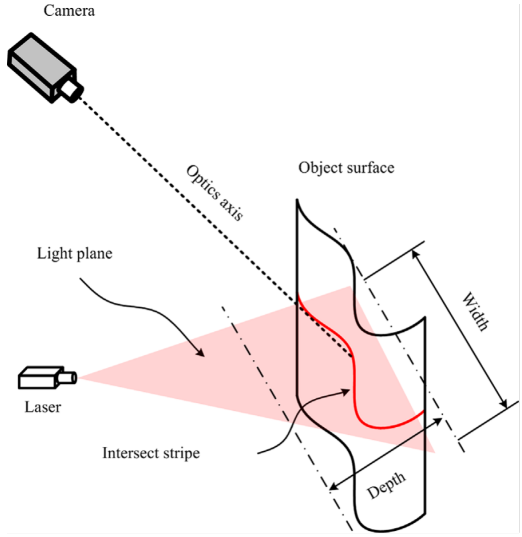


Figure 3.1.: Laser line scanning principle. Source: [20]

Structured light approaches such as Fringe Projection Profilometry and binary encoding work in a similar manner in terms of triangulating between the light source and the camera capturing an image of the object. However, the structured light technique involves projecting an pattern image instead of a laser line, enabling rendering of almost the full view of the camera instead of only the cross section. For a structured light approach called fringe projection profilometry (FPP), explained more in-depth in Section 3.2, the projector pattern is a series of sine waves alternating in one direction of the image. These sine waves code the surface such that the camera can use the reference points created by the intensity of the sine waves, also referred to as "fringes". These fringes can be projected horizontally, vertically as in Figure 3.2 or at an angle. The projected fringes create several planes orthogonal to the sine wave which can be triangulated against. These planes exist for each captured phase value in the image, in a similar way to the one plane created by the laser line.

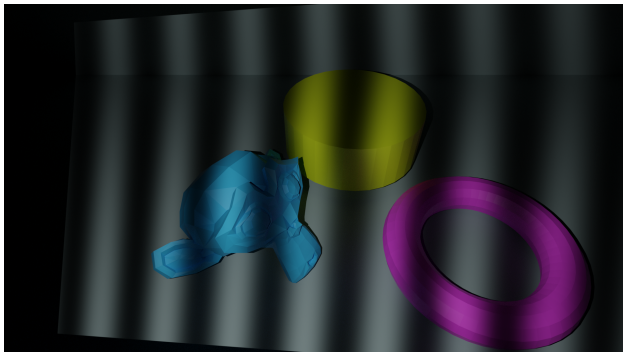


Figure 3.2.: Vertical fringes projected to a scene in Blender.

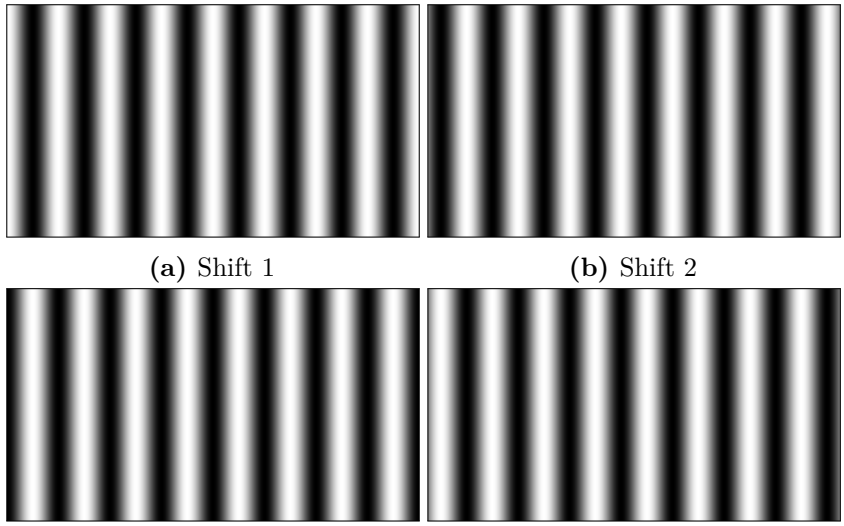
3.2. Structured light

Structured light is a popular choice for 3D scanners. It is used for many industrial applications due to its accuracy and dense point clouds. Structured light scanning can be achieved through numerous approaches, with more techniques being developed continuously as it is an active field of research. The approaches to structured light vary in many different ways. The projected patterns can, for example, be color encoding, binary, phase shifting and step-phases [17].

3.2.1. Structured light phase shifting patterns patterns

The patterns used in different structured light techniques have different properties, some patterns are binary, and code the surface in discrete sections, and others project continuous patterns. Generally discrete patters are less accurate than continuous patterns, since they cannot reach camera pixel level accuracy due to the resolutions of the binary stripes having to be greater than pixel width.

Figure 3.3 shows an example of continuous patterns. The patterns are sinusoidal waves varying in the horizontal direction of the image coding the surface by the amplitude of the sine waves.



(a) Shift 1 **(b)** Shift 2

(c) Shift 3 **(d)** Shift 4

Figure 3.3.: Fringe patterns for 4 shift algorithm with equal shifts

3.2.2. Phase shifting algorithm

The projected sinusoidal patterns of an N-step phase shifting method [17] consist of multiple periods of one wavelength in the horizontal direction of the image. An example of such patterns, for a 4-step algorithm, can be viewed in Figure 3.3. Different phase shifting algorithms vary in the number of phase shifts are applied, and therefore the number of images to capture. For the general N-step case with equal phase jumps, the intensity in pixel x, y for shift n is given as

$$I_n(x, y) = I'(x, y) + I''(x, y) \cos(\phi(x, y) + 2\pi n/N), \tag{3.1}$$

where $n = 1, 2, \ldots, N$ and the phase shift applied for each projected image is the $2\pi n/N$ expression. The terms $I'(x, y)$ and $I''(x, y)$ is the average intensity and the intensity modulation respectively, for the pixel in the pixel coordinate x, y. The average intensity $I'(x, y)$ is given by

$$I'(x, y) = \frac{\sum_{n=1}^{N} I_n}{N}, \tag{3.2}$$

and the intensity modulation $I''(x, y)$ is given by

$$I''(x,y) = \frac{\sqrt{(\sum_{n=1}^{N} I_n \cos(2\pi n/N))^2 + (\sum_{n=1}^{N} I_n \sin(2\pi n/N))^2}}{N}. \tag{3.3}$$

The output of the algorithm is the phase $\phi(x,y)$ in each pixel, from Equation (3.1). The phase can be derived through the expression

$$\phi(x,y) = -\arctan 2\left(\frac{\sum_{n=1}^{N} I_n \sin(2\pi n/N)}{\sum_{n=1}^{N} I_n \cos(2\pi n/N)}\right), \tag{3.4}$$

which outputs the "wrapped phase", $\phi(x,y)$, in the range $[-\pi, \pi]$. From Equation (3.1), there are three unknowns, and thus, the number of images needed to solve for the wrapped phase is three, or more. The general N-step expression for ϕ in Equation (3.4) can be obtained obtained through performing the least squares algorithm [21] on Equation (3.1), solving for $\phi(x,y)$. First, it is re written as

$$I_n = I'(x,y) + I''(x,y)\cos(\phi(x,y)\cos(2\pi n/N) - \sin(\phi(x,y)\sin(2\pi n/N), \tag{3.5}$$

from the angle sum identity for cosine. Easing the notation, this expression becomes

$$I_n(x,y) = \alpha_0(x,y) + \alpha_1(x,y)\cos(2\pi n/N) + \alpha_2(x,y)\sin(2\pi n/N), \tag{3.6}$$

where

$$\begin{aligned} \alpha_0 &= I'(x,y) \\ \alpha_1 &= I''(x,y)\cos(\phi(x,y)) \\ \alpha_2 &= -I''(x,y)\sin(\phi(x,y)). \end{aligned} \tag{3.7}$$

The least squares method then uses the squared difference between the measured intensity I_n, and the predicted intensity, from Equation (3.6). Thus, obtaining the expression

$$E^2 = \sum_{n=1}^{N} [I_n(x,y) - \alpha_0(x,y) - \alpha_1(x,y)\cos(2\pi/N) - \alpha_2(x,y)\sin(2\pi/N)]^2. \tag{3.8}$$

The minimum error is found when derevating the expression with respect to the unknowns α_0, α_1 and α_2, and setting the expressions equal to zero. To simplify the notation, δ_n is used for the $2\pi/n$ phase shifts, and the summation limits are kept out. This derivation gives the three equations

$$\begin{aligned}
\frac{dE^2}{d\alpha_0} &= \sum I_n - \alpha_0 N - \alpha_1 \sum \cos(\delta_n) - \alpha_2 \sum \sin(\delta_n) = 0 \\
\frac{dE^2}{d\alpha_1} &= \sum I_n \cos(\delta_n) - \alpha_0 \cos(\delta_n) - \alpha_1 \sum \cos^2(\delta_n) - \alpha_2 \sum \cos(\delta_n)\sin(\delta_n) = 0 \\
\frac{dE^2}{d\alpha_2} &= \sum I_n \sin(\delta_n) - \alpha_0 \sin(\delta_n) - \alpha_1 \sum \sin(\delta_n)\cos(\delta_n) - \alpha_2 \sum \sin^2(\delta_n) = 0
\end{aligned} \tag{3.9}$$

.

Modifying these expressions by setting the entries with the unknowns α_{0-2} on one side of the equation, enables us to put the equations on matrix form

$$\mathbf{A}(\delta_n)\alpha(x,y) = \mathbf{B}(x,y,\delta_n), \tag{3.10}$$

where

$$\mathbf{A}(\delta_n) = \begin{bmatrix} N & \sum \cos(\delta_n & \sum \sin(\delta_n) \\ \sum \cos(\delta_n) & \sum \cos^2(\delta_n) & \sum \cos(\delta_n)\sin(\delta_n) \\ \sum \sin(\delta_n) & \sum \cos(\delta_n)\sin(\delta_n) & \sum \sin^2(\delta_n) \end{bmatrix}, \alpha(x,y) = \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \end{bmatrix} \textit{ and } \quad \mathbf{B}(x,y,\delta_n \tag{3.11}$$

The problem of obtaining the unknowns now become a matrix calculation problem, where the inverse of $\mathbf{A}(\delta_n)$ is applied to matrix $\mathbf{B}(x,y,\delta_n)$, to obtain α_{0-2}

$$\begin{bmatrix} \alpha_0(x,y) \\ \alpha_1(x,y) \\ \alpha_2(x,y) \end{bmatrix} = \mathbf{A}^{-1}(\delta_n)\mathbf{B}(x,y,\delta_n). \tag{3.12}$$

Once the three unknowns are obtained, for each pixel in the image, the phase data can be extracted from equations (3.11) and (3.12)

$$\phi(x,y) = \arctan 2\left(\frac{-\alpha_2(x,y)}{\alpha_1(x,y)}\right). \tag{3.13}$$

In the case of evenly spaced phase shifts, where $\delta_n = 2\pi n/N$ The off diagonal elements of matrix $\mathbf{A}(\delta_n)$ become zero, and the solution for the phase $\phi(x, y)$ then becomes

$$\phi(x, y) = \arctan\left(\frac{-\sum_{n=1}^{N} I_n \sin(\delta_n)}{\sum_{n=1}^{N} I_n \cos(\delta_n)}\right), \tag{3.14}$$

which is the same expression as Equation (3.4). The resulting wrapped phase image from this equation contains 2π phase jumps, resulting in a "sawtooth" patterned image as in Figure 3.4, where the mod (2π) function has been applied to the phase. This is to remap the phase from the range $[-\pi, \pi]$ to $[0, 2\pi]$

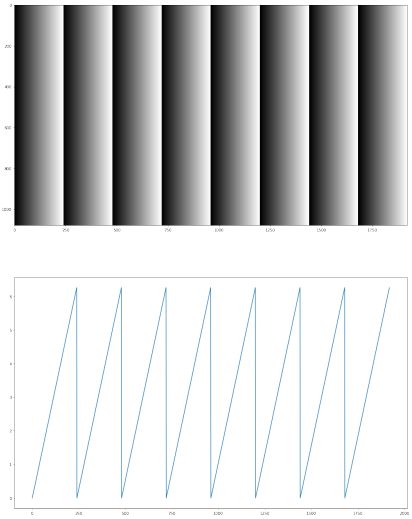


Figure 3.4.: Wrapped phase image with 8 periods (top), and its intensity cross section (bottom)

As Figure 3.4 shows, the phase is uniquely defined inside each of the 8 periods. However, the phase is not unique across the periods, and to obtain the full spatial relation between the periods, a phase unwrapping algorithm has to be applied.

3.2.3. Phase unwrapping

Unwrapping the phase is the process of uniquely identifying the phase, and thus the y-coordinate, of each pixel in the captured images. The target of phase unwrapping is to eliminate the phase jumps in the wrapped phase image, to create a continuous phase across the image as in Figure 3.5. This is called the absolute phase $\Phi(x, y)$ Phase unwrapping algorithms can be classified in three main

categories [22]; temporal algorithms, colour encoding algorithms and spatial algorithms.

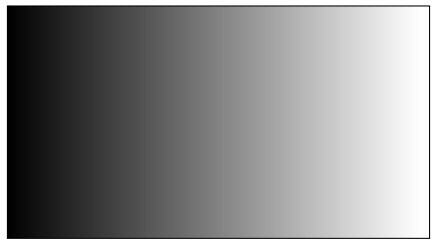


Figure 3.5.: Absolute phase, with intensity ranging from 0 to 2π

Spatial unwrapping algorithms works under the assumption that the scanned surfaces are smooth with no large discontinuities. The main idea behind such approaches is that neighbouring pixels in the wrapped phase image dictates the phase value in the unwrapped phase image. This is done by adding $2k\pi$ to the wrapped phase in the event that a discontinuity in phase value is detected in the wrapped phase image, and the unwrapped phase then become

$$\Phi(x, y) = \phi(x, y) + 2\pi k, \tag{3.15}$$

where k ranges from 0 to $K - 1$. Where K is the number of fringes in the projected pattern. Such algorithms are widely used, mostly on surfaces without large discontinuities [22].

Temporal algorithms typically use phase images of different wavelengths, resulting in their corresponding wrapped phase images, through phase detection as described in Section 3.2.2. To obtain the absolute phase more than one fringe pattern is typically required, and often a temporal phase unwrapping algorithm is applied [17]. Instead of utilizing the spatial relations to unwrap the phase, the temporal algorithms utilizes the phase information of each pixel, from more than one projected wavelength pattern. The "absolute" phase is then recovered through the relationship between the phases of the different wavelength fringes.

As a simple example for what is meant by the relation between the phase in fringe patterns, one can look at the heterodyne approach or two-wavelength approach [23]. The algorithm involves subtracting the phase measurements in the wrapped phase image from each of the wavelengths

$$\phi_{eq} = \phi_1 - \phi_2, \tag{3.16}$$

where ϕ_{eq} is the equivalent phase. This is the phase yielded by the equivalent

wavelength

$$\lambda_{eq} = \frac{\lambda_2 \lambda_1}{\lambda_2 - \lambda_1} \tag{3.17}$$

where λ_{eq} is also called the synthetic wavelength at beat frequency. If the wavelengths are related as $\lambda_1 < \lambda_2 < 2\lambda_1$, it follows that $\lambda_1 < \lambda_2 < \lambda_{eq}$. The beat frequency refers to the frequency of the wave produced by the absolute value of two interacting waves. If the two wavelengths λ_1 and λ_2 are close, one can completely eliminate the 2π phase jumps, and end up with the the absolute phase from only applying Equation (3.16) and using the mod (2π) function on the result. However, such choice of algorithm means that the signal to noise ratio become small, and the absolute phase image will be prone to noise. In Figure 3.6 the heterodyne principle is used on two waves, wave 1 with 8 periods (upper left), and wave 2 with 7 periods across (upper right). The resulting absolute phase (lower left) is smooth and continuous across the image. However, the algortihm is run perfect fringe images, and therefore no noise effects are visible, as opposed to if these patterns were projected onto an object.

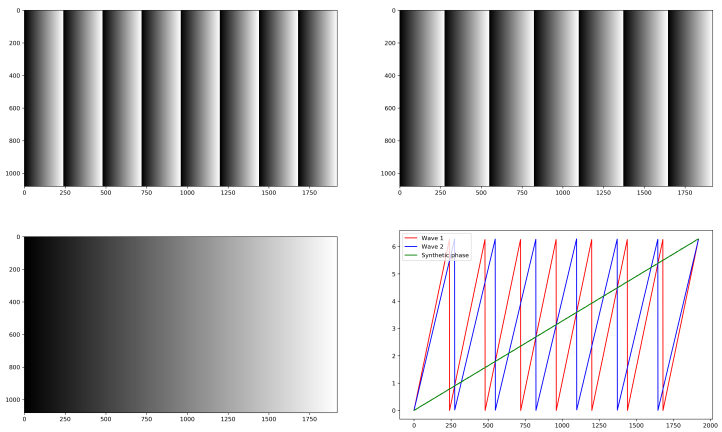


Figure 3.6.: Heterodyne principle with two fringes, phase 1 with 8 periods and phase 2 with 7 periods combined into the absolute phase

Because the two-wave heterodyne approach is so prone to noise interaction, the synthetic phase map Φ_{eq} is most often used as a reference phase to assist further phase unwrapping [24]. The ratio of synthetic wavelength λ_{eq} to the shortest wavelength λ_1 is combined to a scaling factor to create a step image $k_1(x, y)$ defining the fringe order for the longer wavelength image

$$k_1(x,y) = Round\left[\frac{(\lambda_{eq}/\lambda_1)\phi_{eq}(x,y) - \phi_1(x,y)}{2\pi}\right], \tag{3.18}$$

The $k_1(x,y)$ scaling image is applied to the captured fringe image by adding the two captured images together. The cross section of two ideal images of k_1 and ϕ_1 can be viewed in Figure 3.7, which illustrates the principle. Each pixel of the step image, $k_1(x,y)$ is multiplied by 2π and added to the unwrapped phase $\phi_1(x,y)$ pixels. Thus, the absolute phase image is obtained and the phase jumps eliminated.

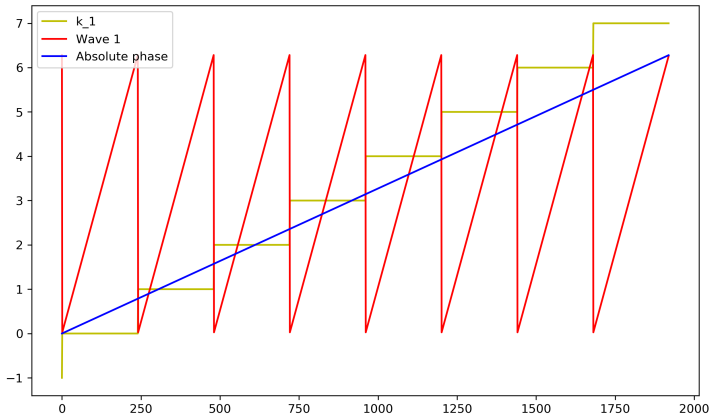


Figure 3.7.: k_1 step image, ϕ_1 and absolute phase cross section for phase images with 8 and 7 periods.

The absolute phase in Figure 3.7 is found through

$$\Phi(x,y) = \phi_1(x,y) + k_1(x,y)2\pi \mod (2\pi), \tag{3.19}$$

where the mod (2π) function is applied to take the abolute phase $\Phi(x,y)$ from the range $[0, 8 \cdot 2\pi]$ to the range$[0, 2\pi]$.

3.3. Triangulation

3.3.1. Point triangulation

Depth measurements can be extracted by observing the same point in two or more images, captured from different camera angles with a known transformation between their optical centers. The method replicates how depth perception functions for humans, in that both eyes are looking at the same scene, with a slightly

different point of view. The principle also applies to what is known as active stereo vision [25],[26]. Active stereo vision is a form of structured light approach, where patterns are projected onto an object, and thus coding its surface in phase values, such as the structured light approaches explained in Section 3.2.2. The two stereo cameras can then use the coded surface to find point correspondence between their image frames, and performing point triangulation explained below.

Extracting the depth value of a 3D point Q is done by triangulating its image projections [27], $\mathbf{q}_i$, where $i = 1, 2, ..., n$ is the camera number. Each camera has a projection matrix $\mathbf{P}_i$, associated with it, which maps a 3D point Q in homogeneous world coordinates, to pixel coordinates in the image plane. The matrix $\mathbf{P}_i$ is described by

$$\mathbf{P}_i = \mathbf{K}_i[\mathbf{R}|\mathbf{t}], \tag{3.20}$$

where $\mathbf{K_i}$ is the camera matrix, describing the intrinsic parameters of the calibrated camera, and $[\mathbf{R}|\mathbf{t}]$ describing the extrinsic camera parameters, i.e. the pose of the camera center in world coordinates.

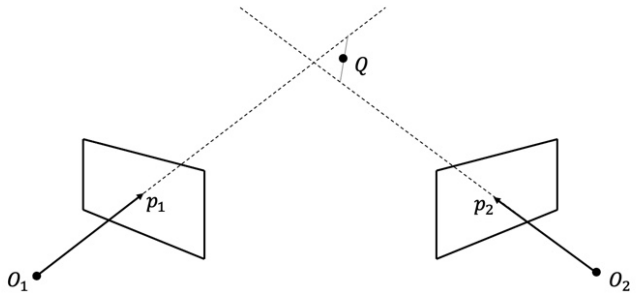


Figure 3.8.: Point triangulation with a stereo camera setup of point Q, in the presence of noise. Where O_1 and O_2 is the optical centers of the two cameras and $\mathbf{p}_1$ and $\mathbf{p}_2$ are the in-homogeneous image coordinates.

Without noisy observations, the problem of triangulation would simply be to find the intersection of the lines going through the image coordinate p_i from each optical center. However such observations does generally not occur in real world problems, and the lines therefore do not intersect, as can be viewed in Figure 3.8. Therefore the triangulation becomes a task of finding the point at minimal distance between these lines.

Siter Hartley and Zissermann, section "12.2 Linear triangulation methods", hvor finnes artikkelen Obtaining the triangulated value of a 3D point

Q is done though a linear algorithm involving "Single Value Decomposition" [28], hereafter referred to as SVD. In this linear algorithm, the rows of $\mathbf{P}_i$ are denoted by superscript,

$$\mathbf{P}_i = \begin{bmatrix} \mathbf{P}_i^1 \\ \mathbf{P}_i^2 \\ \mathbf{P}_i^3 \end{bmatrix}. \tag{3.21}$$

Using this notation, the camera model can be expanded as

$$\begin{aligned} \mathbf{q_i} &= \begin{bmatrix} s_i x_i \\ s_i y_i \\ s_i \end{bmatrix} = \begin{bmatrix} \mathbf{P}_i^1 \\ \mathbf{P}_i^2 \\ \mathbf{P}_i^3 \end{bmatrix} Q \\ s_i x_i = \mathbf{P}_i^1 Q\,, \quad s_i y_i &= \mathbf{P}_i^2 Q\,, \quad s_i = \mathbf{P}_i^3 Q \\ x_i = \frac{s_i x_i}{s_i} = \frac{\mathbf{P}_i^1 Q}{\mathbf{P}_i^3 Q}\,, \quad y_i &= \frac{s_i y_i}{s_i} = \frac{\mathbf{P}_i^2 Q}{\mathbf{P}_i^3 Q}. \end{aligned} \tag{3.22}$$

From these equations for pixel values of x_i and y_i, an expression for Q can be developed as follows

$$\begin{aligned} x_i = \frac{\mathbf{P}_i^1 Q}{\mathbf{P}_i^3 Q}\,, \quad & y_i = \frac{\mathbf{P}_i^2 Q}{\mathbf{P}_i^3 Q} \\ x_i \mathbf{P}_i^3 Q = \mathbf{P}_i^1 Q\,, \quad & y_i \mathbf{P}_i^3 Q = \mathbf{P}_i^2 Q \\ x_i \mathbf{P}_i^3 Q - \mathbf{P}_i^1 Q = 0\,, \quad & y_i \mathbf{P}_i^3 Q - \mathbf{P}_i^2 Q = 0 \\ (x_i \mathbf{P}_i^3 - \mathbf{P}_i^1) Q = 0\,, \quad & (y_i \mathbf{P}_i^3 - \mathbf{P}_i^2) Q = 0. \end{aligned} \tag{3.23}$$

Since Q has three degrees of freedom from its coordinates X, Y and Z, the triangulation algorithm needs three constraints from Equation (3.23). Thus, the minimum number of camera poses required is two, such that Q has four constraints. Looking at Equation (3.23), the x_i and y_i parts correspond to planes. The x and y coordinates of Q's image defines a plane with coefficients $P_i^3 x_i - P_i^1$, which Q lies on. Where the planes intersect, is the line stretching from the optical center of the camera to the 3D point, corresponding to the back projection of the 2D point $\mathbf{q_i}$. From these constraints, Q can be calculated by transferring them to matrix form

$$\mathbf{B} = \begin{bmatrix} P_1^3 x_1 - P_1^1 \\ P_1^3 y_1 - P_1^2 \\ P_2^3 x_2 - P_2^1 \\ P_2^3 y_2 - P_2^2 \\ \vdots P_n^3 x_n - P_n^1 \\ P_n^3 y_n - P_n^2 \end{bmatrix}, \tag{3.24}$$

where $\mathbf{B}$ is a $n \times 4$ matrix. Using $\mathbf{B}$ Equation (3.23) can be transformed into

$$\mathbf{B}Q = \mathbf{0}. \tag{3.25}$$

From the Equation (3.25) it is seen that a trivial solution of $Q = \mathbf{0}$ exists, and thus an additional constraint $||Q|| = 1$ is needed to ensure that the homogeneous representation of Q is not scaled to be zero.

As mentioned, noisy observations mean that Equation (3.25) will not hold perfectly, and that the solution is found by

$$\min_Q ||\mathbf{B}Q||_2^2 \quad where \quad ||Q|| = 1. \tag{3.26}$$

Which is solved through applying SVD [28] to $\mathbf{B}$, resulting in

$$\mathbf{B} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{T}}, \tag{3.27}$$

where the minimal solution for Equation (3.26), and thus the Q, is found to be the last column vector $\mathbf{v}_n$ of the matrix $\mathbf{V}$.

3.3.2. Active structured light triangulation

Contrary to the point triangulation explained in Section 3.3.1, the triangulation for the fringe projection profilometry algorithm, are between the projector and a camera. This approach is called active structured light, and is dependant on a properly calibrated projector, because the intrinsic and extrinsic projector matrices are used for triangulation. In the case of phase shifting and binary encoding, which are commonly used structured light techniques, the patterns are only variant in one direction. As mentioned this can be in the horizontal direction, vertical direction or at an angle. If the fringes or binary stripes are vertical, the patterns vary in the horizontal direction. This means that the triangulation process are performed using three coordinate entries [29]. From the camera this is the u_c

and v_c pixel coordinates, but for the projector, assuming the pattern stripes are vertical, only the v_p coordinate holds any information useful for the triangulation.

These three coordinates are linked through the absolute phase $\Phi(x, y)$ obtained though the phase detection and unwrapping explained in Sections 3.2.2 and 3.2.3 by the constraint

$$\Phi(u_c, v^c) = \Phi(u_p), \tag{3.28}$$

where u_c and v_c are the camera pixel coordinates, and u_p is the projector pixel coordinate in the horizontal direction. Each phase's pixel coordinates in the camera frame is matched with the corresponding phase coordinate in the projected pattern. The camera pixel coordinates and projector pixel coordinates' relation to world coordinates of a 3D point $Q = [X_w, Y_w, Z_w]$ are defined as

$$\lambda_c \begin{bmatrix} u_c \\ v_c \\ 1 \end{bmatrix} = \mathbf{P}_c \begin{bmatrix} X_w \\ Y_w \\ Z_w \end{bmatrix} \quad and \quad \lambda_P \begin{bmatrix} u_p \\ v_p \\ 1 \end{bmatrix} = \mathbf{P}_p \begin{bmatrix} X_w \\ Y_w \\ Z_w \end{bmatrix}, \tag{3.29}$$

where λ_c and λ_p are the scaling factors for the homogeneous pixel coordinates and $\mathbf{P}_c$ and $\mathbf{P}_p$ are the camera and projector projection matrices. In the following equation, their matrix elements are denoted with c and p superscript, the matrices are defined as

$$P_c = \begin{bmatrix} p^c_{11} & p^c_{12} & p^c_{13} & p^c_{14} \\ p^c_{21} & p^c_{22} & p^c_{23} & p^c_{24} \\ p^c_{31} & p^c_{32} & p^c_{33} & p^c_{34} \end{bmatrix} \quad and \quad P_p = \begin{bmatrix} p^p_{11} & p^p_{12} & p^p_{13} & p^p_{14} \\ p^p_{21} & p^p_{22} & p^p_{23} & p^p_{24} \\ p^p_{31} & p^p_{32} & p^p_{33} & p^p_{34} \end{bmatrix}. \tag{3.30}$$

From the Equations 3.28 and 3.29, the resulting 3D word coordinates for Q is obtained from triangulation

$$\begin{bmatrix} X_w \\ Y_w \\ Z_w \end{bmatrix} = \begin{bmatrix} p^c_{11} - p^c_{31}u_c & p^c_{12} - p^c_{32}u_c & p^c13 - p^c_{33}u_c \\ p^c_{21} - p^c_{31}v_c & p^c_{22} - p^c_{32}v_c & p^c23 - p^c_{33}v_c \\ p^p_{21} - p^p_{31}v_p & p^p_{22} - p^p_{32}v_p & p^p23 - p^p_{33}v_p \end{bmatrix}^{-1} \begin{bmatrix} p^c_{14} - p^c_{34}u_c \\ p^c_{24} - p^c_{34}v_c \\ p^p_{24} - p^p_{34}v_c \end{bmatrix}. \tag{3.31}$$

Chapter 4.

Method

This chapter discusses the choices made and methods used when researching, the designing and testing the pipeline and structured light concepts used throughout the research and development related to this report. The main focus points are the design choices for the pipeline, as well as testing of different structured light techniques in combination with Blender.

4.1. Choice of software

When the development of the pipeline was initialized, the first priority was to select a suitable 3D software platform that incorporated all features needed to render, randomize and was somewhat easily programmable. A literature study on similar work was performed, to get an overivew of different approaches and possible software. Parts of the following section is adapted from the project thesis [9], about the same subject.

The idea of a pipeline producing synthetic data is not new, and there are several approaches to the problem. The focus of each of the pipelines vary by what type of problem is aimed to optimize. Some pipelines are focused on rendering speeds and utilize domain randomization on the data to try and bridge the reality gap. Such data sets apply random lighting conditions and place objects often in unrealistic positions. Sundemeyer et.al. [30] used the OpenGL [4] rendering pipeline to produce randomized data. The resulting images are not in line with physics, due to OpenGLs lack of physics simulation possibilities.

Another commonly used type of software for generating synthetic data is game engines such as the Unity engine [31]. These are highly capable and relevant software, with a large variety in functionality. However, they are often expensive and not accessible to everyone. Thus, it was decided not to use such software, since one of the main focus points of this project was to make the pipeline open-source.

By making pipelines, datasets and other tools open-source, the community could then benefit from a large number of contributors generating data. Such a software was found in Blender [32].

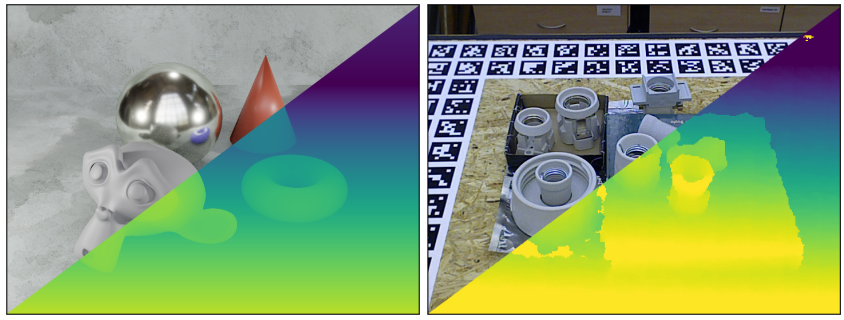
Blender is a powerful open-source 3D creation software, with a large community. It incorporates a large variety of functionalities. Most important for the pipeline is its physics simulation, light simulation and rendering capabilities. With Blender, it is possible to create realistic 3D environments, due to its node based material editor, physics simulation and ray tracing capabilities.

Blender also has an embedded Python interpreter [33]. This allows users to initialize actions, extract data and modify 3D objects in Blender through Python scripts. This means that the repetitive actions associated with data generation can be automated, and output data for the pipeline can be collected and presented in the format that is needed. Using Python alongside Blender in the pipeline also enables the utilization of Python's large selection of "modules". A Python module is a library of classes in Python that can be imported to extend functionality.

To access Blender through Python, the "bpy" module is used. The module allows python script to do actions in almost every part of Blenders functionality base. Settings of the "scenes", which is a concept of a 3D environment inside blender, which is elaborated on in Section 4.3.

4.2. Structured light camera in Blender

Blender is an extensive platform that laid the foundation for making a working pipeline for realistic synthetic datasets. The large amount of functionalities incorporated in Blender, enabled the creation of a projector that could be controlled though python and Blender, and therefore making a digital structured light camera model. The inspiration to create such a camera in Blender was taken from a Zivid blog post [34], where the author created a binary pattern structured light setup, to understand the technique and algorithms of the method. The thought behind then incorporating this in DataPipe, was to help bridge the "reality gap" discussed in Section 2.4. The difference in real captured data and synthetic data produced by rendering the z-buffer depth is shown in Figure 4.1, where the synthetic depth data 4.1a is near perfect, in contrast to the uneven surfaces in the real captured T-LESS data 4.1b. By mimicking the effects that noise has on depth sensing with real cameras, the aim was to create more realistic datasets for training neural networks. However, such a camera would mean a lot more images would have to be rendered per image created which was an important consideration throughout the research.



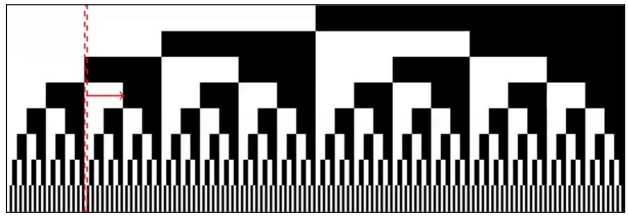
(a) Blender z-buffer depth data. **(b)** T-LESS captured depth data.

Figure 4.1.: Comparison of synthetic rendered depth data from z-buffer and real captured data from T-LESS dataset [8]. Images were received from my co-supervisor, Sebastian Grans.

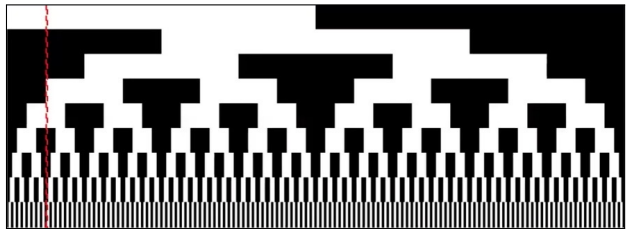
4.2.1. Binary structured light

Although the main focus of this project is not the data generation speed, some considerations had to be made regarding the number of images rendered per scene. The rendering process of the images captured in Blender is the largest time consumption in the pipeline with rendering time of approximately one minute for an image with resolution 1920×1080, on a standard Macbook Pro model. Since datasets for neural networks often need to be in the range of several tens of thousands of entries and often upwards, depending on the task, this sort of time spent on rendering implies the need to minimize images generated per scene render for the structured light camera in Blender.

As mentioned in Section 3.2.1, there are a number of different patterns that can be used for structured light. At the start of the project, the search for the most applicable method for structured light scanning, was the main research area. The aim was to utilize a method that enabled high accuracy, was easy to implement and required as few patterns as possible to be projected. The first type of patterns considered was the binary patterns, more specifically, graycode patterns [17], because of the simplicity of the implementation itself. Graycode patterns were more favorable than ordinary binary patterns, due to their robustness against misinterpreted bits as illustrated in Figure 4.2. Gray code patterns only change one bit at a time, such that is one bit is misinterpreted, the spatial displacement between the correct and misinterpreted bit will always be small. For binary patterns however, this spatial displacement is much larger, as shown by the arrow in Figure 4.2a, where the value of one bit in the sequence is misinterpreted.



(a) Binary pattern sequence.



(b) Gray code pattern sequence.

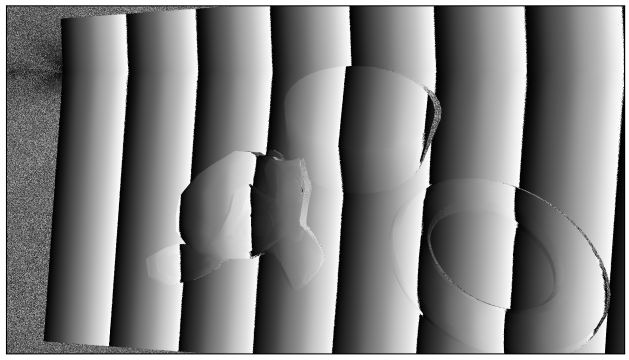
Figure 4.2.: Comparison image of binary and gray code patter sequences.

The downside found to using binary approach of structured light however, is that the patterns limit the resolution of the point-cloud in that the projected patterns can not be divided into camera pixel size, to avoid miss interpretation of the camera pixel values. In addition, the approach can be very costly in terms of images that has to be taken to obtain the wanted accuracy. For graycode, each image projected doubles the number number of stripes in the pattern such that to divide a pattern with, for example, dimensions 1280×720 into stripes with 2 pixels width, one would need to solve $n = \log_2(1280) - 1$, and round up the result, which is 10, but two additional images would have to be projected as well, one completely lit and one completely dark. This is to correct for color variance in each pixel. Thus, the number of patterns to project and render would be 12, which would affect the run time of the pipeline to a large degree. In addition, binary structured light not being of the most accurate techniques, the approach was deemed unfavorable at an early stage.

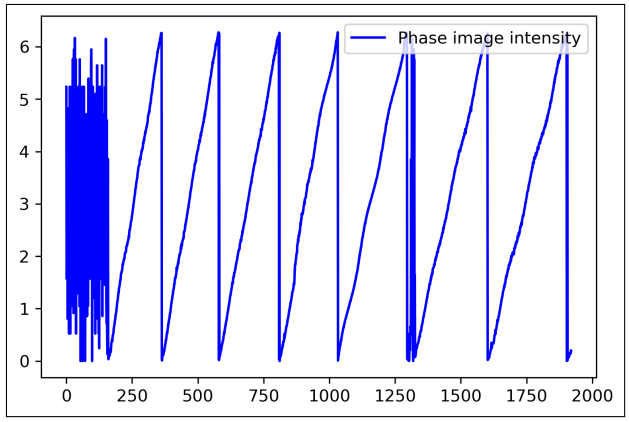
4.2.2. Phase detection

Fringe projection profilometry, explained in Section 3.2.1, as a structured light technique was of interest since the properties of the sinusoidal patterns allowed for camera pixel accuracy, contrary to binary pattern methods, it is not affected by the resolution of the projected patterns, due to the ability to interpolate between the sine values captured. Different approaches utilizing fringe projection was considered, and the initial experimentation was testing different approaches to phase detection, where the wrapped phase was solved for.

At the start of the testing and research, the initial approach was to use Equation (3.4) with a three step heterodyne algorithm. The phase detection algorithm was applied to every pixel to obtain the full phase image. In the early stages, this involved looping the image pixel by pixel, calculating the pixel's corresponding phase value from three phase shifts. Time consumption of the approach was between 40 seconds, and upwards to 1 minute 10 seconds for an image of resolution 1920 by 1080. However, the phase image was obtained, and the wrapped phase in Figure 4.3 looked promising at such an early stage.



(a) Wrapped phase image



(b) Wrapped phase intensity

Figure 4.3.: Wrapped phase image and phase image intensity for 8 fringe pattern, captured in Blender.

As the phase detection speed was slow due the algorithm having to loop and separately compute two million pixels' phase value, at resolution 1920 by 1080 pixels, a new method was proposed. In the Section 3.2.2, Fourier profilometry was presented, which was a solution to the run time problem, as the numpy library contains a function called FFT [35], short for "Fast Fourier Transform", which

can be performed on every pixel in the shifted patterns, without going through a loop. The FFT function compute the Fourier transform and outputs the signal in frequency domain. With signal, it is refered to each pixel's phase values at each phase shifted image, together illustrating a partial sine wave with entries for each phase shift, as in Figure 4.4. These sampled sine values were taken as input to the numpy FFT which output a list of complex representation of the waves, $z = a + ib$ at the different frequencies incorporated in the signal. The first output list entry is the average value of the wave, and the second entry is the largest wavelength, which is the projected pattern. By extracting this complex representation of the wave, the phase can be computed by finding the angle of the complex number. This was done through the numpy.angle function which computes the angle for the complex number through the expression $\phi(x, y) = \arctan(b/a)$.

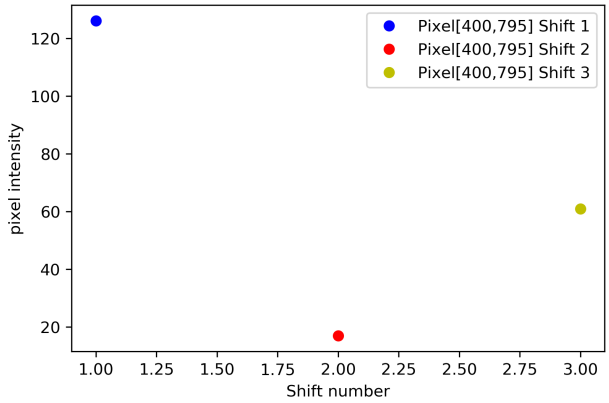
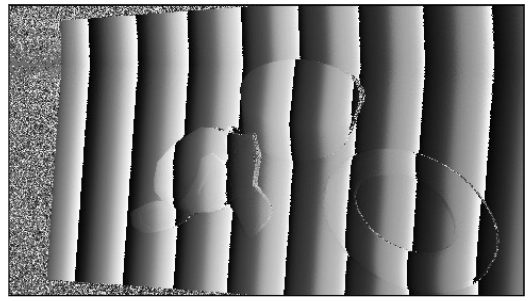
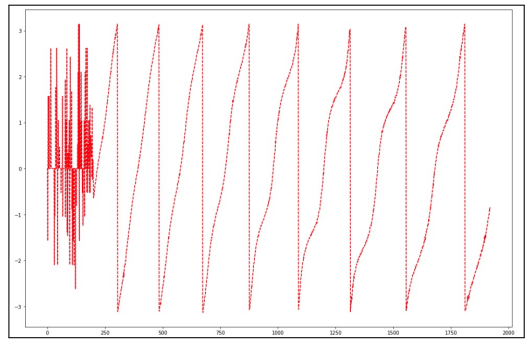


Figure 4.4.: Intensity value of pixel $u = 400$, $v = 795$, in three phase shifted images.

The obtained wrapped phase image was, at first glance, similar to that of the previous approach using the arctan function to obtain the phase value. However, what looked like periodic misinterpreted phase values as shown in Figure 4.5, started to emerge. The misinterpreted phase values are shown as waves forming on the saw tooth cross section of the wrapped phase. At first this was suspected to be caused by low light strength from the projector, causing the projected phase patterns to be saturated, causing the sine waves to be flattened at the peak intensities. Tho test if this was the case, a range of strength values was tested as projector output, but yielded no results, and the wrapped phase images showed similar misinterpreted phase values.



(a) Wrapped phase image from FFT approach, from 5 step algorithm



(b) Cross section of wrapped phase obtained from FFT approach

Figure 4.5.: Wrapped phase image and cross section from FFT appraoch.

After eliminating the light strength as cause to the misinterpretation, it was suspected that the cause of the periodic errors could be caused by the Cycles ray tracing engine [6] incorporated in Blender. The Blender ray tracing is a stochastic algorithm [36], where light rays are reflected off surfaces in a randomly selected direction. It was therefore suspected that the lack of light in the low lit intensities of the fringe patterns, could cause noisy measurements in these places. To test if this was the cause of the periodic errors in the wrapped phase image, the sampling of rays per camera pixel was set to 1000, in contrast to 128 which is the default setting used before. Another approach was to raise the lowest pixel intensity in the projected patterns from 0 to 50, which was the range where most noisy measurements was captured.

In Figure 4.6 the captured images from the three approaches can be viewed. The blue line correspond to the default number of samples per pixel, and it is clearly very noisy in the lower intensities of the projected fringe patterns. Compared with the red line, which is the 1000 samples image, a large difference in noise can be observed. This is indicating that the noise could be affecting the unwrapped image.

Since the 7 images with 1000 samples took 45 minutes to render, it is thus not an approach that is valid for the full pipeline. However, when using 128 samples with the intensity shifted patterns the noise was reduced to almost the same level as the 1000 sample image, illustrated by the green line. However, the wrapped phase image still had periodic phase misinterpretations despite minimizing the noise.

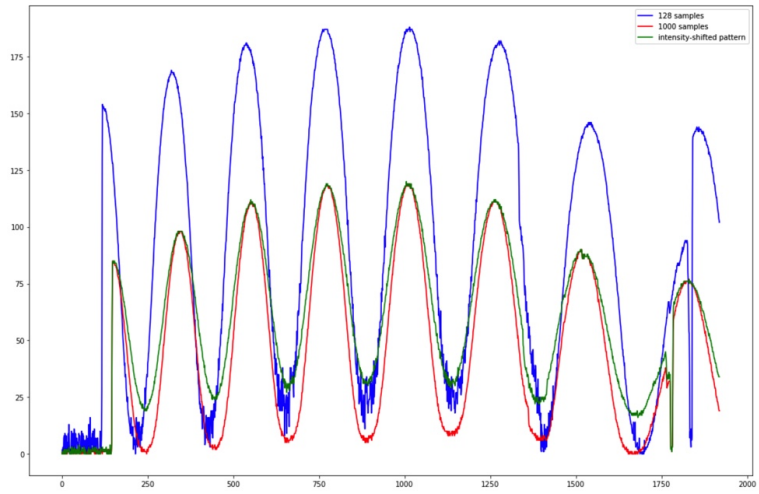


Figure 4.6.: Intensity cross-section of the captured images projected by fringe patterns. The blue graph shows the captured image rendered with 128 samples per pixel, the red graph shows 1000 samples and the green graph shows 128 samples with a higher minimum intensity in the projected pattern.

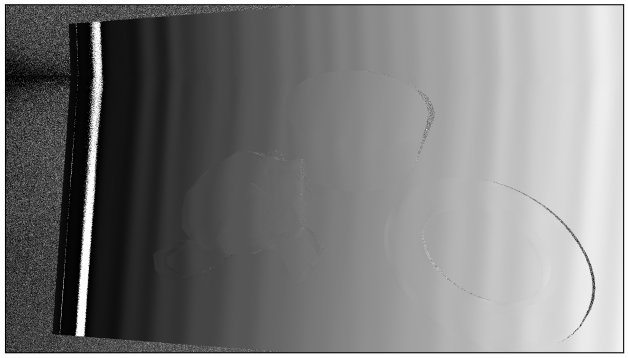
Further experimentation was done by projecting patterns of different wavelengths on the same scene, checking for a link between the misinterpreted phase values and wavelength. The experimentation showed that the misinterpretations became more prominent for the shorter wavelengths, but still also present in the longer wavelength tested. From the tests, it was presumed that the misinterpretations was due to the FFT causing aliasing due to the number of phase shifts being too low, which gave the FFT too few data points to determine the phase.

Because the FFT approach resulted in wrapped phase images with periodic misinterpreted phase values, the arctan approach was again the favoured. However, the approach of computing each phase changed from looping every pixel to applying a matrix multiplication algorithm using the Numpy Python library, improving the computation time and outputing cleaner wrapped phase images.

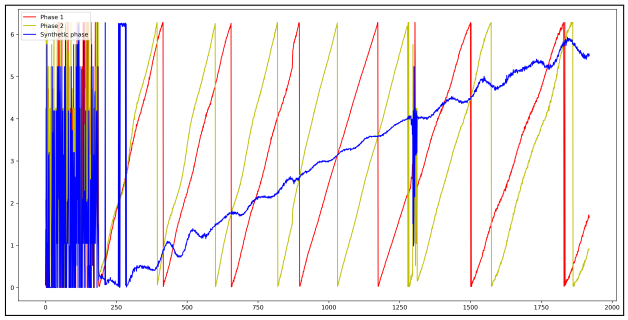
4.2.3. Phase unwrapping

Obtaining the absolute phase of the fringe images is completely dependent on the quality of the wrapped phase images, when using a temporal algorithm. The temporal algorithm from Section 3.2.2, using the heterodyne approach with two wavelengths was first tested. Two projected patterns with 9 and 8 fringes was first projected. The image projected with the shortest wavelength pattern was subtracted by the longer wavelength and the modulo operation was applied to their difference. The resulting image is their synthetic phase, with pixel values from 0 to 2π, because the 8 fringe pattern is exactly one wavelength less than the 9 fringe pattern.

The synthetic phase can be viewed in Figure 4.7, showing the synthetic phase image 4.7a and the cross section of the synthetic phase and the two corresponding wrapped phases 4.7b. The synthetic phase was at an early stage of the research mistakenly believed to be the output absolute phase result, which it seldom is, as explained in Section 3.2.3.



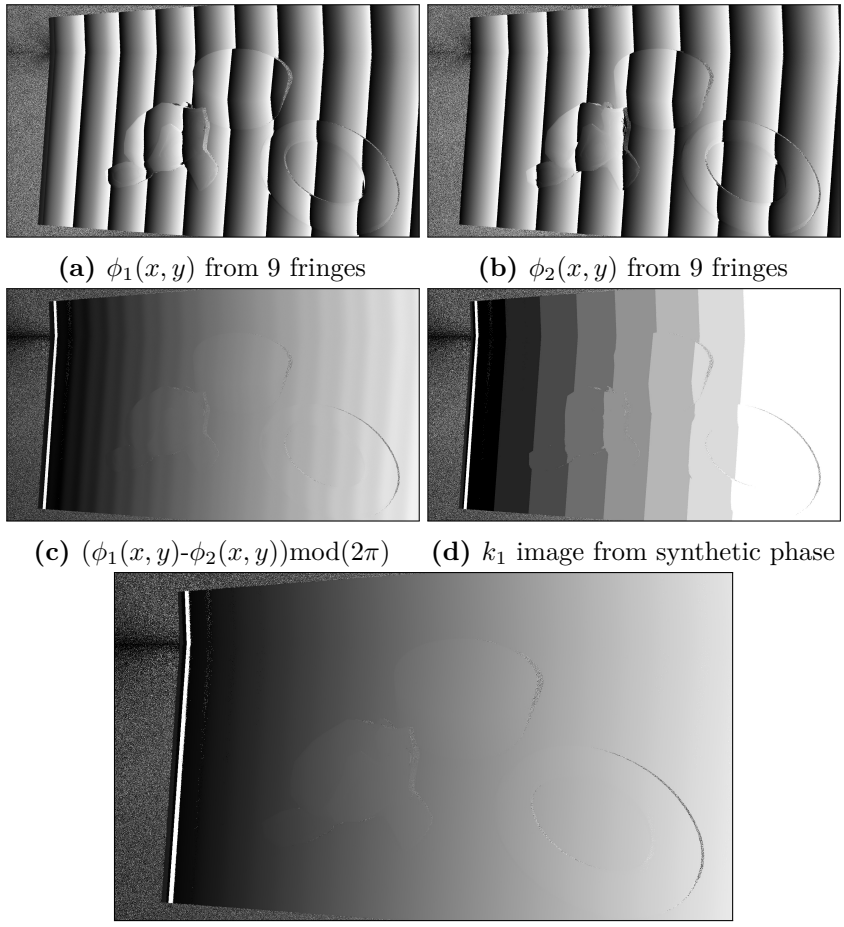
(a) Synthetic phase image from 9 fringes and 8 fringes patterns.



(b) Cross section of wrapped phases and synthetic phase.

Figure 4.7.: Phase unwrapping with 10 and 9 period image patterns, with the use of heterodyne synthetic phase.

The synthetic phase was found to most often be used as a reference phase in the process of further unwrapping the phase. As the phase values from the synthetic phase is very susceptible to noise, it is not applicable for use in triangulating between the projector and camera. However, the scaling factor assisted algorithm from Section 3.2.3, where the shortest and more sensitive wavelength projected is scaled by a step image k_1. The image k_1 determines the fringe order of the discontinuous wrapped phase from said wavelength by being added to the wrapped phase image. The absolute phase is thus transformed to the range $[0,\ 2\pi K - 1]$ where K is the number of fringes in the smallest wavelength image. The images obtained from the algorithm can be viewed in Figure 4.8.



(a) $\phi_1(x, y)$ from 9 fringes **(b)** $\phi_2(x, y)$ from 9 fringes

(c) $(\phi_1(x, y)\text{-}\phi_2(x, y))\text{mod}(2\pi)$ **(d)** k_1 image from synthetic phase

(e) Absolute phase, from k_1 and ϕ_1

Figure 4.8.: Phase unwrapping with 10 and 9 period image patterns, with the use of heterodyne synthetic phase.

4.2.4. Triangulation

Triangulation between the projector and camera is done through the algorithm from Section 3.3.2. For the triangulation the intrinsic matrices from both the projector and camera was needed. Because the camera and projector intrinsic parameters are determined from user input, these matrices are not the same for every run of the pipeline. Thus they had to be calculated at run time. For the camera matrix, the algorithm from [37] was used, which gather the matrix entries directly from the camera object in Blender and compute the same camera model as described in Section 2.5. The projector, a similar algorithm is used, but the matrix inout is collected from the input variables defined by the user, since

the projector is created from a light source object, and therefor have no Blender specific properties to collect. The camera extrinsic transformation matrix is set to be the unit 3×4 matrix, and thus the coordinate frame the depth vales are referenced from. The transformation between the projector is collected directly from Blender, with the camera object being a parent object to the projector. These operations are performed by the `Algorithm` class, which can be viewed in Appendix A.13.

4.3. DataPipe architecture

From the previous report [9] and the creation of the first pipeline in Blender, experience and new ideas on how to improve the quality and structure of a pipeline was obtained. For instance, the earlier version of the pipeline was designed in a way that made expanding and improving functionalities difficult. The class design was cluttered and inconsistent. Thus, the need to build a new pipeline from ground up was the easiest approach for improving. The new pipeline, DataPipe, was build using proper instancing and object oriented techniques, instead of using static classes handling too many tasks and variables. Classes were now designed to have fewer responsibilities, more closely related to the Blender objects they handle. This made for a better overview of the classes' functionalities and responsibilities, and making the process of expanding and changing the pipeline Python classes more agile. The code for all DataPipe classes and scripts can be viewed in Appendix A.1-A.14.

Figure 4.9 shows a simplified overview of the main components and processes included in the pipeline. The pipeline revolves around two loops, the scene creation loop and the render loop. In the scene creation loop the objects are imported based on the user input described later, in Section 4.3.7. These objects are given physics properties, and their physics are simulated, to mimic a bin-picking scenario, with randomly positioned objects in the scene. This scene is then rendered from one or multiple camera poses. The output information related to each rendered camera angle is then sent to the output file and the render loop continues until all selected camera poses are rendered. This launches the scene creation loop, where the process repeats itself until the total number of renders, from user input, is reached. The pipeline then terminates and resets to be ready for new runs.

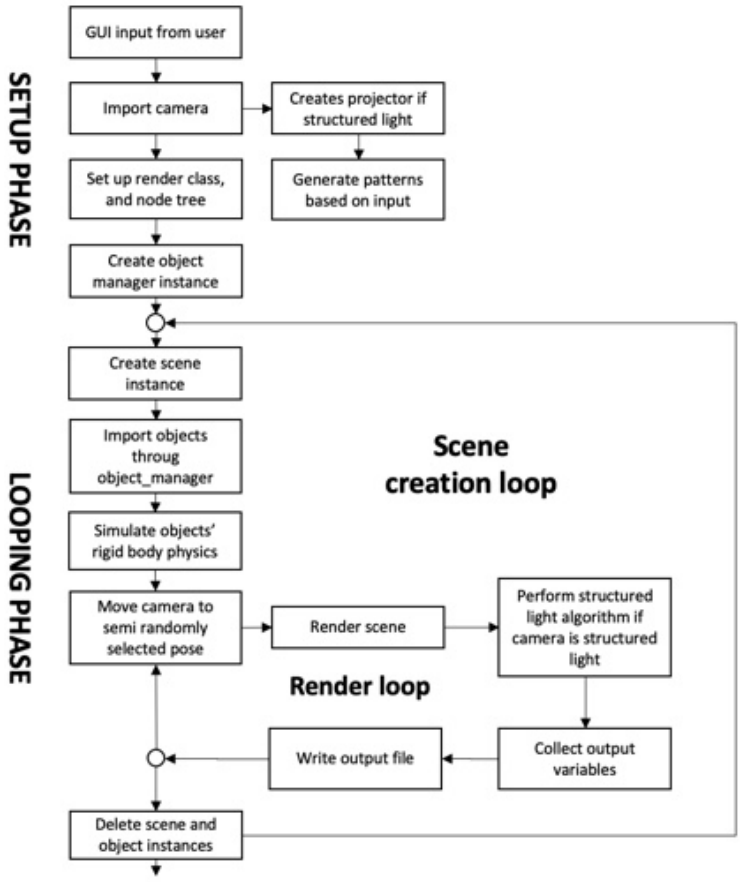


Figure 4.9.: Pipeline process overview.

4.3.1. Pipeline outputs

The selection of pipeline outputs were based on inputs that are needed for a neural network utilizing 3D data will need for 6D pose estimation. Looking at critical outputs and some "nice to have" outputs, the following were selected:

- Depth image from structured light.
- Depth image from z-pass.
- Ground truth 6D poses of the objects in the rendered images.
- Masked image
- Normals map image

A critical factor for the data is the depth image, which introduces the spatial aspect of the data. From the depth image a point cloud can be produced in a desired format, for example using the Open3D library **Zhou2018** in Python, outside the pipeline. The depth is rendered through the structured light algorithm from Section 4.2, but also through the z-pass which is the perfect depth image gathered directly from the Blender software. The RGB image is also rendered along with the z-pass, and adds three dimensions, in the form of pixel colors, if used in the neural network input at training time.

To use this point cloud data to train a neural network, the ground truth poses of the objects in each point cloud are needed. Ground truth poses are used to compute the error in the estimations from the neural network. The error is then back-propagated in the network, altering the weights. This altering of the weights is making the model into a better approximation next time it is run, as discussed in Section 2.1. Therefore, the pipeline needed to output a rotation and a translation measure for the objects relative to some coordinate system. Representing the translation part of the pose is most naturally done by the x, y and z coordinates. However, for rotation this was not as clear, but the representation that was chosen was the rotation matrix. This is due to it being an intuitive and familiar representation of rotations, as well as its matrix multiplication properties described in Section 2.2.1.

The masked image is an image containing the object indexes, whic are uniquely defined for every object in the image. The image can be useda to link the object output data from the output file to the object instances viewed in the image. The masks and indexes are integer values which is written to an OpenEXR file [38] which can contain the object indexes instead of RGB values as the usual .jpg or .png images. An example of such an image can be viewed in Figure 4.10, where the OpenEXR file is converted to a human readable format, with the different indexes having different intensities. In addition to describing an objects location in the image frame, it can be used for instance segmentation and object recognition tasks.

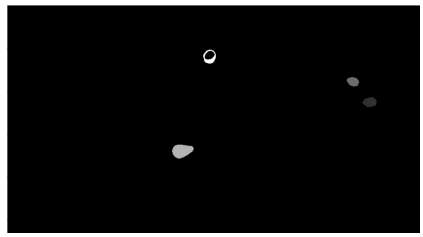


Figure 4.10.: Example of masked image, transfered to human readble form

The normals image illustrates the estimated normals of the 3D environment in the image. The normal vectors are perpendicular to the surfaces in each pixel in the image. They are represented as RGB channels describing the the entries in the normal vectors, red being the x value, y is green and z is blue. The normals image is not a critical output for the pipeline, but it is rendered simultaneously to the z-pass. Thus, it being rendered does not affect the speed of the rendering process. Also, many deep learning algorithms improve when the normals are introduced as input in addition to other inputs, as stated in the article by M. Denninger et al. [5].

4.3.2. Blender add-on

The main goal of the pipeline was to create a tool for producing large amounts of realistic data for 3D applications, in a way that was user friendly and easy to learn. The initial approach for gathering user input was having a .json file created by the user, containing a dictionary from where the variables utilized by the pipeline were collected. This was an easy method to implement and is used in the pipeline BlenderProc [5]. However, the downside is the lack of opportunity to guide the user to correctly define the inputs. The user is required to read and understand a lot of documentation, to be able to make use of the features of the pipeline.

Blender has its own python distribution, and incorporates a scripting environment, from where the user can work behind the graphical user interface, hereafter referred to as GUI, of Blender itself, and automate processes and create scripts that extend it's functionalities. Along with this, developers can utilize existing components and properties from Blenders GUI, to create additional elements to the existing GUI where user inputs can be collected for python scripts. This is called "add-ons", and can be created by everyone using Blender.

The GUI components used are called "properties" and "operators" in the blender environment. A property is an input element that can be defined as vectors, integers, floats, enumerators and more. Operators are added as buttons that launch other python scripts that are called when the user launches these operators. These properties and operators are added to the GUI in what is called "panels" in Blender, which is GUI element "containers" displaying the GUI elements that are added.

An example of such a panel with properties and operators can be viewed in Figure 4.11. The panel contains the projector and camera intrinsic parameters in the form of integer properties, the projector extrinsic parameters relative to the camera are in form of two float vector properties and the structured light parameter is a Boolean property that determines if the projector is included, and thus structured

light rendering is activated. At the bottom of the panel, three operators that manage with the camera poses, can be seen; camera preview, append and remove camera pose.

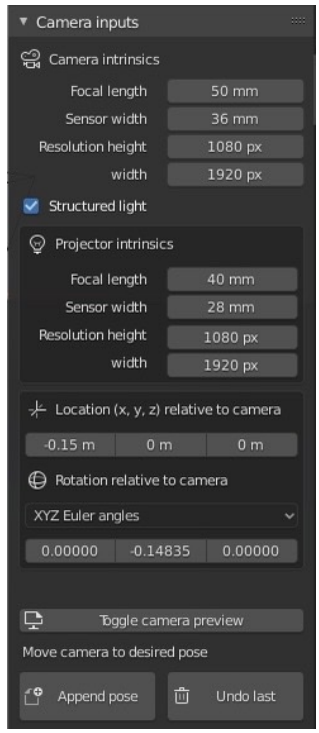


Figure 4.11.: GUI panel for camera inputs

The decision of making DataPipe into an add-on was taken to improve the user interaction with the pipeline, and allow for a lower threshold for using the pipeline. Blender has a large user base that are familiar with its GUI elements, which makes DataPipe easy to use, and also easy to access. In addition, even as the building of the pipeline was far along in the development, conversion from using a .json file to an add-on GUI, was easily done, by making the GUI inputs write to the same type of file. Thus, the pipeline could be left unaffected, and still collect the input from the file.

Another problem that was solved when changing to add-on, was the working directory problem that was faced before conversion. The pipeline classes was created as separate python files launched by the pipeline `main.py`. To use the pipeline in blender, the function had to be imported to Blender, and being launched by Blender's own bundled python distribution. Thus, losing the relative paths to the rest of the pipeline classes, as well as any link to the pipeline folder structure.

Creating the pipeline as an add-on, enabled gathering all pipeline related directories and files inside Blenders add-on folder. This also meant that file locations and paths were relative to Blender, which meant absolute paths could be input through the use of the directory property. The property enabled the use of a file explorer in the pipeline GUI which outputs the absolute paths, and makes for better user interaction for inputting object files and defining output directories.

Add-ons are relatively easy to install for the user as well. The DataPipe zip file can be downloaded from the github repository, and installed through Blender's preferences tab inside the GUI, then the add-on can be turned on and off as needed.

4.3.3. BlendScene class

The concept of a scene in Blender is the environment that is created in the 3D view of the program. The `BlendScene` class of the pipeline are responsible for all operations carried out on the Blender scene, as well as outputting the information about its associated objects and camera for every scene instance that is rendered through the pipeline.

It was important that the scene layout can be varied by the user, as this helps create data sets that are specific to the task at hand. It was therefore decided that the user creates and models the scene layout outside the pipeline, then run the pipeline from the .blend file containing the user created scene. This allows the user to preview different elements of the pipeline, such as the camera and its poses, and how they integrate with the scene layout.

The `BlendScene` class of DataPipe encapsulates all DataPipe objects that are linked to the pipeline. As input, the scene class was set to control the amount of total renders to be executed for the pipeline run, and how many different camera poses that were to be rendered for each scene. Another key element to the scene is the location of the drop zone. It determines the location of the initial positions of the physics objects that is to be simulated. The positional and dimensional agruments for this drop zone is collected from an object that is imported through the use of a Blender operator. The drop zone object is then moved and scaled by the user to fit the scene enviroment. When the pipeline is launched, this information is gathered, and passed to the `ObjectManager` class, in charge of the objects' initial placements.

Output information about all blender objects contained in each scene is stored in a dictionary that the `BlendScene` objects generates. Information about these objects is collected from the `ObjectManager` object, explained in Section 4.3.7 and the `BlendCamera` object. The information that is stored is the camera pose in the

world coordinate frame, and for the objects it is a dict containing information about the objects currently in the scene. This dict is elaborated on in Section 4.3.6. When each scene is finished rendering, the output dict is appended to the `DataPipe_output.pickle file`, such that if the pipeline crashed, all information so far, will be available from the output file.

4.3.4. BlendCamera class

The ambitions of creating a digital version structured light camera for the pipeline, made the `BlendCamera` and `Projector` classes some of the more critical components of DataPipe. Because the structured light sensor addition to the pipeline would produce data of unknown quality, it was early determined that it should be possible to toggle the feature on and off. In addition, the substantial rendering time of such data due to the addition of several more images per scene, the need for an option to select a simple camera outputting noise free "z-buffer" data was added at an early stage of the development.

`BlendCamera` takes input from the user for the intrinsic parameters in the form of focal length, resolution and sensor width, relating to pixel size. The camera position is defined by toggling camera preview on and moving the preview camera object to the desired position. The interactive camera positioning was implemented for convenience and such that the user could get visual aid when selecting camera positions. The user can still choose to position the camera with angle inputs, but additionally, it was now possible to drag an rotate the camera object without knowing the exact coordinates. The input from the camera is collected by extracting the position vector and the quaternions from the preview camera object, and appended to a list in the input dictionary containing camera positions so that the user could choose to have multiple camera views of one scene. The quaternion representation was chosen due to it's robustness in 3D transforms against gimbal lock, and due to inaccuracy when converting between Blenders representation of Euler angles and rotational matrices, which is needed for the output format of the objects positions relative to the camera frame.

Randomizing parameters to create diverse data was important for the pipeline to generate good quality datasets, where the neural network could train on as many different scenarios as possible. Building on this it was wanted that the camera positions should be randomized to a certain degree. However, if the data were to model a specific situation, the user could want to define specific camera poses to better model the real environment. As a compromize to completely randomized camera poses, it was determined that a semi-random camera pose selection should be implemented. Poses are semi-random in that they are randomly chosen for each scene, from a pool of poses defined by the user at input. Randomization is then

achieved in that not all scenes are rendered from the same view point, but the user still has full control over the poses that are in the pool.

The `BlendCamera` object has a boolean variable stating if it is a structured light camera. Setting the input for structured light to true, a nested `Projector` object instance will be initialized. This projector is coupled to the `BlendCamera` object by setting the Blender light objects, explained in the following section, as child objects of the camera, such that they maintain their relative pose to the camera as it moves.

4.3.5. Projector class

The `Projector` class was the main element enabling the digital representation of the structured light camera. Creating a digital representation of a projector in blender was done through the use of light source objects [39] in Blender, with a node tree applying an image texture to the emitted light rays. At the start of the development an add-on for creating projectors in Blender was found [40]. It enabled creation of projector through the GUI intuitive, but it was hard to control through Python. In addition, the use of the add-on would create dependencies for potential pipeline users and the functionality of the pipeline itself, and it would rely on compatibility if the projector add-on was further developed. Hence, it was decided to create a projector object from scratch.

The `Projector` class would have to project multiple different projected patterns, and the solution was to add multiple light sources that project different patterns on the scene. By doing that, a method for rendering one pattern at a time was required. This was solved by using the "view layers" in Blender, which is layers added to the scene in scene in Blender where specific object can be hidden and showed based on what layer they are placed on. All layers viewed the same scene, but the different light sources could now be hidden in the view layers they did not belong to.

Making the light source objects function like a projector, a node tree had to be created for each of the lights. The node tree can be viewed in Figure 4.12, and consist of seven nodes added and modified through the `Projector` class. The first node is a texture coordinate node mapping the textures of the unit sphere that is the light object emitting rays in the normal direction to the light object. From the node, the normal vectors of the unit sphere are taken as output, and divided by the z-component through the use of a separate XYZ and a vector divide node. The division is performed to remove the pincushion distortion of the projected image, essentially projecting the image on a plane, instead of a unit sphere. These vectors are then mapped to the image, giving each vector a pixel value through the mapping node and image texture node, applying the structured

light pattern image. This mapped image is then sent to an emission node which sets the light strength, and then this mapped image is sent to the light output node projecting the image onto the scene.

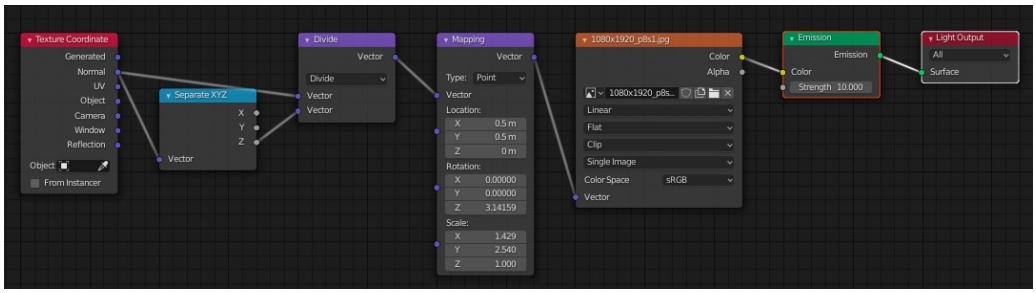


Figure 4.12.: Node tree for projector creation from Blender light source object

To set the projector's field of view the mapping node scale in Figure 4.12 has to be set appropriately. This was done by modeling the light source as an inverse camera. Input from the user was taken as focal length and sensor size, and the scaling in x and y direction was computed similar to the camera matrix in Equation (2.24) from Section 2.5 from the relations

$$scale_x = \frac{f}{s_x} \; and \; \; scale_y = \frac{f}{s_y}, \tag{4.1}$$

where f is the focal length from input and s_x and s_y is the sensor size in horizontal and vertical direction respectively. Modeling the projector as an inverse camera, means that the projector intrinsic matrix can be calculated in the same manner as the camera matrix. Due to the limitations of the report and the modeled in Blender's software being perfectly free from noise, the projector was decided not to be calibrated, since that is a whole field of study in itself.

4.3.6. BlendObject class

The objects in scene is a key element for the pipeline, therefore it was important that the input process was easy, such that the user would take the time to input as many different objects as possible to get diverse data sets. The file explorer property made the process much less cumbersome, by not having to find the path to every single object before pasting it into the input file.

Blender has meters as default unit in it's modeling environment, which can be changed through python code. However, objects that were modeled in millimeters tended to be imported with the default units in meters, making the objects scaled

by 1000 and therefore not fitting the dimensions of the scene, which would impede the physics simulations described in Section 4.3.8. With the pipeline created as an add-on, these objects could be previewed before they were added to the scene, and scaled accordingly by the user, before running the full pipeline.

`BlendObject` also takes collision shape as user input, which can be chosen to be either the mesh shape or a convex hull surrounding the mesh of the object. Selecting the latter drastically improves the physics simulation time, because the collision shape has a lower resolution and is less detailed that using the mesh shape. This enables the user to cut time for objects that does not have intricate and concave shapes.

Each instance of `BlendObject` has a set of instance variables that are passed as output for the pipeline. These variables are name, filename, pass index and the transformation matrix. The variable name, is to uniquely identify the object in each scene instance, the name is given at import time, and is on the form "DataPipe_object.xxxx", where "xxxx" correspond to the pass index padded with zeros. The filename variable is collected from the object import path. This variable is sent to output to be an identifier for the object type, if the pipeline data is to be used for segmentation tasks. The pass index is tied to the masked image output, where each of the objects can be identified by their pass index. The file format of the masked image is ".exr", which is not limited by the standard format for .png and .jpg where there are three channels for the RGB values, but rather it have arbitrary channel format, which allows for object indexes to be passed as integers instead of a randomly generated unique RGB value. The transformation matrix given as output is the transformation between the camera frame and the object. By default Blender's world coordinate system is situated at the center of each scene, and all translations and rotations are described in that basis. Thus, poses of the objects in the scene are only described in the world coordinate system, and the objects poses in the camera frame therefore have to be computed using transform matrices from Section 2.2.4. Multiplying the inverse transform matrix $T_{sc}^{-1} = T_{cs}$, between the world coordinate frame s to the camera frame c by the transformation T_{so} between the world coordinate s frame and the object frame o results in the transformation matrix describing the 6D pose of the object in the camera frame T_{co}, which is passed as output for each object.

4.3.7. ObjectManager class

The `ObjectManager` class is a helper class for `BlendObject`, which administrates the importing, placing and deletion of object in the scene in Blender. Input from the user is appended to a list of objects to include in the pipeline, along with their maximum and minimum occurrence in each scene creation, as well as their

scale and collision shape discussed in Section 4.3.6. The `ObjectManager` imports a random number of each object within these minimum and maximum for every 3D object file, ensuring a randomized composition of objects in every scene. The user input includes the file path to the 3D object, with the only accepted file type being ".obj". The .obj files allow the objects to be imported with textures, unlike several other 3D file types such as ".stl" and ".ply". Importing object with texture adds to the realism of the outputted dataset images, and creates more variation in the scene.

As objects are imported to the scene, the `ObjectManager` generates random initial positions for the objects above the drop zone in the scene, shown in Figure 4.13. Objects are placed in layers in the vertical direction, based on their size and the size of their neighbouring objects such that no collisions or overlapping occurs. In these vertical layers, each object is given a random position in the xy-plane, to add to the randomization aspect of the simulation, and thus creating diverse data with unique poses mimicking a bin picking scenario.

Figure 4.13.: Initial object poses generated by the ObjectManager class.

The `BlendObject` instances are stored in a list in the `ObjectManager` class, such that their instance variables are accessible from the other classes of the pipeline such as the `BlendScene`, which collects information about the objects at output

time.

4.3.8. Simulation class

- Write about the box generated under the dropzone, to avoid objects clipping through the bottom plane, because of the collision tolerance being so low, that the speed of the objects allow them to slip through the tolerance between frames.

The physics simulation is one of the main parts of DataPipe contributing to the randomization of the generated datasets. The `Simulation` class is in charge of simulating the objects physics properties through the use of Blender's rigid body physics simulation. The 3D objects in the scene environment is simulated from their initial poses generated by the `ObjectManager` class, and interacts with the environment when colliding throughout the simulation time, to end up in disordered poses. This simulation is supposed to mimic the scenario of bin picking where parts are thrown inside a bin, and left randomly distributed, which adds an element of realism to the datasets generated.

While testing the simulations on object models of small scale, problems arose with these objects clipping through the plane they collided with. This was found to caused by the collision margins of the objects being low. This allowed the objects to pass through the thin plane in between frames when the object reached large velocities in the simulated drop. However, the collision margins could not be changed because it caused the objects to hover above the surface and thus, giving the objects unreal poses. This was solved by instead adding an occlusion box, just below the drop zone, flush with the plane, and setting it to be invisible in renders. This meant that the objects would have to pass through a much greater volume in between frames, thus eliminating the clipping problem.

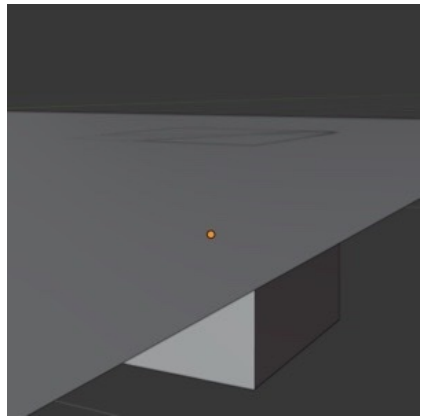


Figure 4.14.: Occlusion box added to prevent objects clipping through plane.

4.3.9. Renderer class

- Rendering in BW mode does not affect the render time, the gray scale filter seems to be applied after rendering
- Maybe write about other rendering engines than cycles, and say that they are possibly better
- Write about the use of a better computer to try rendering on

The render output of the pipeline can be altered using Blender's node system. For the render output, Blender has a "compositor" editor built in, which uses the Blender node system elaborated on in Section 2.3. This allows to collect different outputs from the scene view layers, including the z-buffer, which is the scene depth relative to the camera object, represented by a depth value in each pixel of the image. The compositor allows modification of this information using Blenders node based programming, explained in Section 2.3.

The `Renderer` class is in charge of the whole rendering process from the creation of the compositor node tree to the rendering and creation of the render output directories. The rendering node tree can be viewed in Figure 4.15, where the view layers containing each `Projector` light source is rendered separately. For the six view layers containing the projected sine patterns, to the left in the figure, only the RGB image is rendered, as a .png file. These images are passed through the structured light algorithm in the `Algorithm` class at a later stage of the pipeline. However, to the right in Figure 4.15, the native view layer is rendered. This is the view layer that is created as default, and where the user defined scene lighting still exists. From this view layer, the z-pass depth image of the scene is rendered, along with the normals image, masked image and the RGB image is rendered.

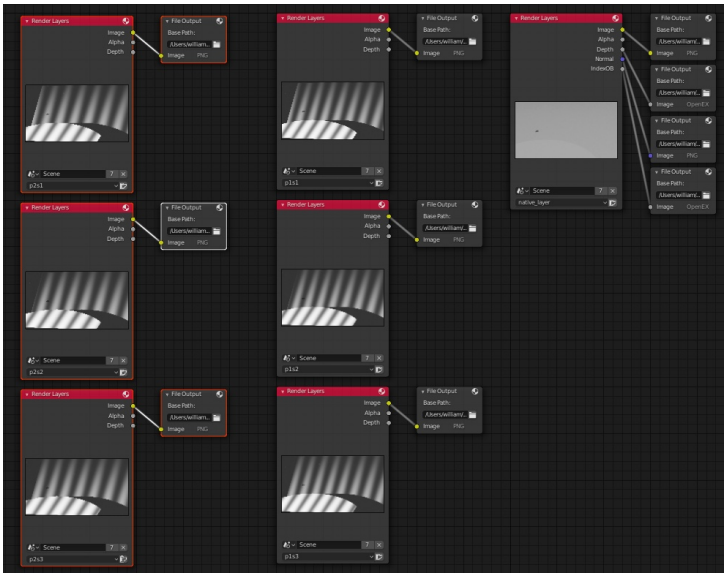


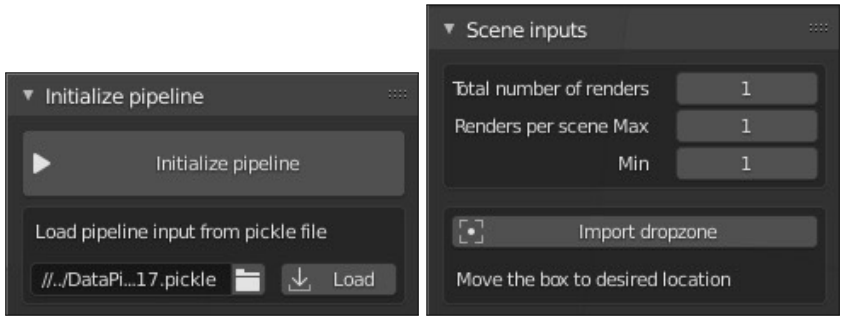
Figure 4.15.: Compositor node tree for two wavelengths with three phase shifts.

Chapter 5.

Results

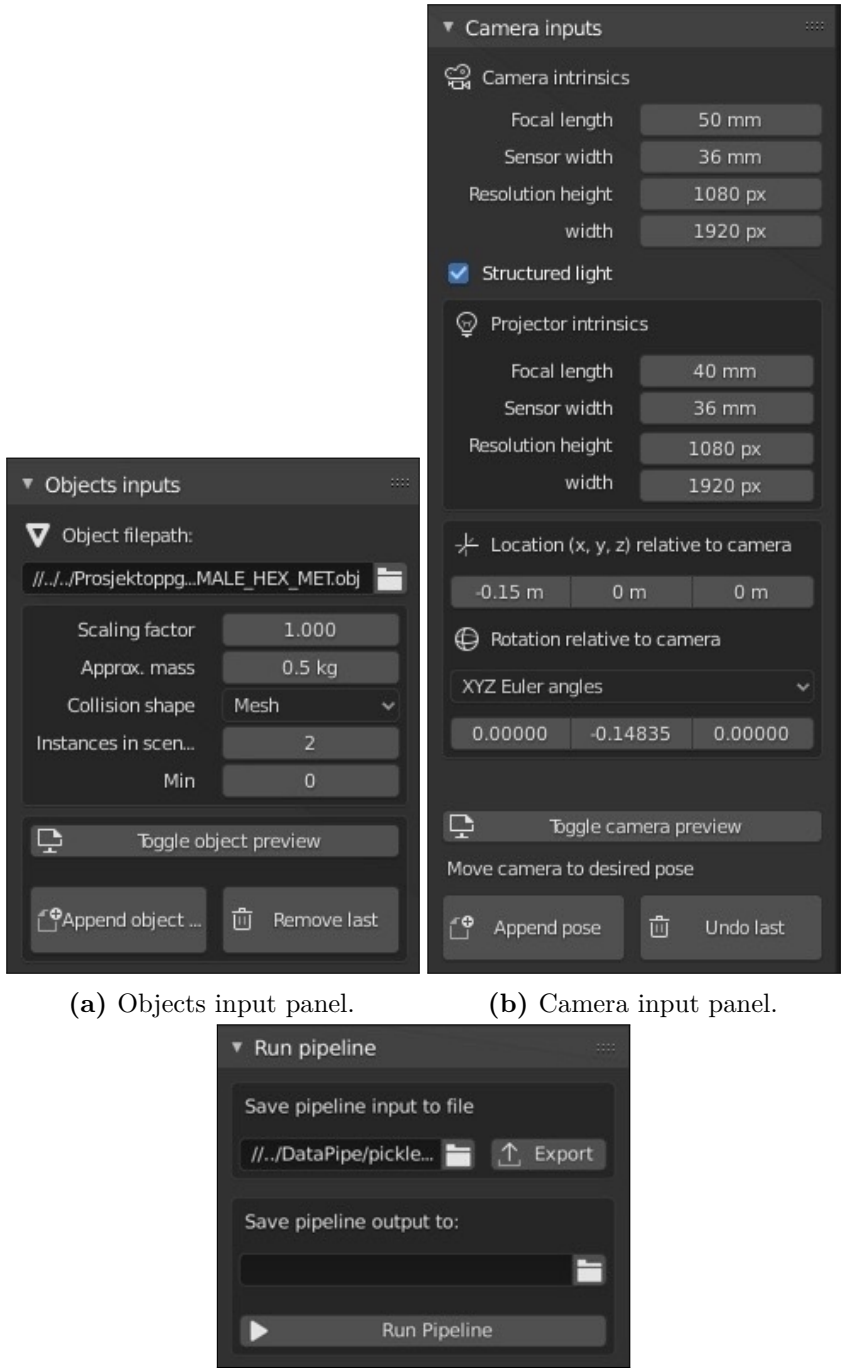
5.1. DataPipe GUI

The DataPipe Blender add-on input panels are created using Blender's own GUI class types, included with the bundled Python API of Blender. Figure 5.1 to 5.2 shows the different GUI panels for collecting user input to the different DataPipe classes making up the pipeline process. In addition, a video showing a short demonstration of the DataPipe GUI elements is provided in the digital appendix.



(a) Pipeline initialization panel. **(b)** Scene input panel.

Figure 5.1.: Scene input panel and initialize pipeline panel



(a) Objects input panel. **(b)** Camera input panel.

(c) Run pipeline panel.

Figure 5.2.: Camera input panel and objects input panel

The DataPipe GUI elements are integrated in the 3D view area of Blender's native GUI, as shown in Figure 5.3, where it is located at the right hand side as a tab. The scripts written to create the GUI elements can be viewed in Appendix A.1.

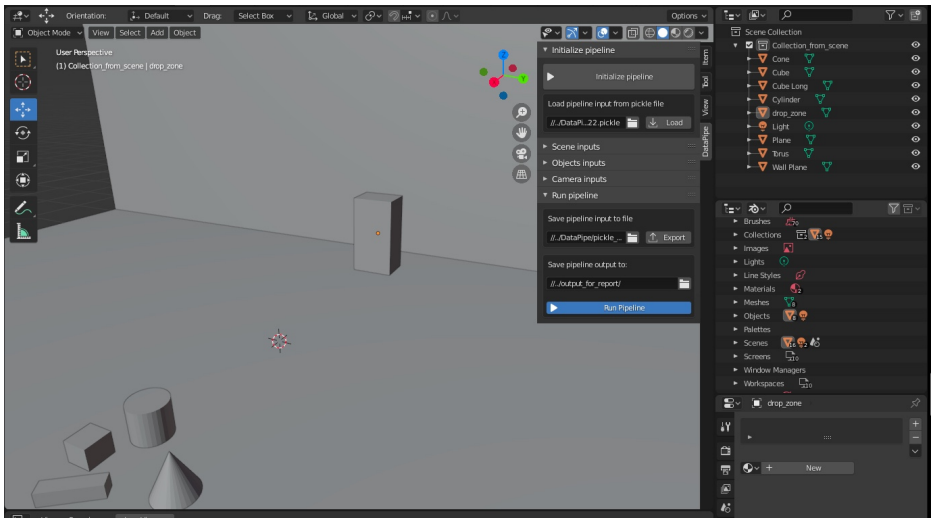
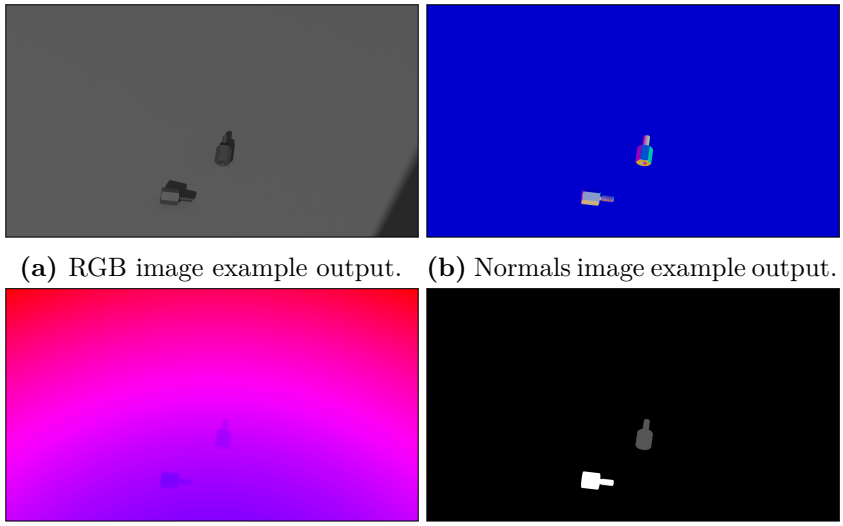


Figure 5.3.: DataPipe input panel placement in Blender's 3D view port

5.2. Pipeline output

DataPipe output for each rendered camera angle includes a z-pass depth image 5.4c, normals image 5.4b, masked image 5.4d, RGB image 5.4a and the structured light depth image included in Figure 5.6d along with the images of the captured fringe patterns projected on the scene in Figure 5.5.



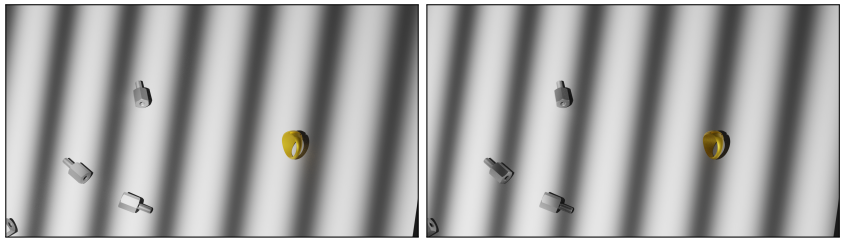
(a) RGB image example output. **(b)** Normals image example output.

(c) depth image example output, color mapped. **(d)** Masked image example output, color mapped.

Figure 5.4.: Example of output images from pipeline run.

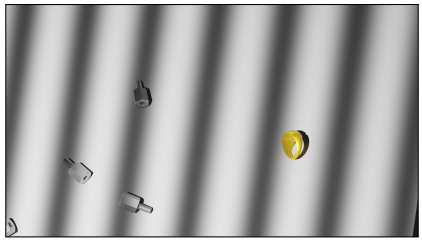
5.3. Structured light

The structured light algorithm consist of four main phases, starting with the rendering of the fringe images. After the rendering is done, the next step is the phase detection on these phase shifted images, before the absolute phase is computed. Lastly, the depth image is obtained by triangulation between the projector and camera frames. In Figures 5.5a to 5.5f, the rendered fringe images can be viewed.

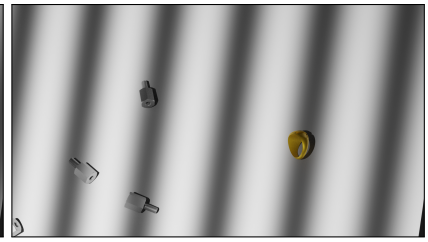


(a) Shift number 1 with 8 period pattern.

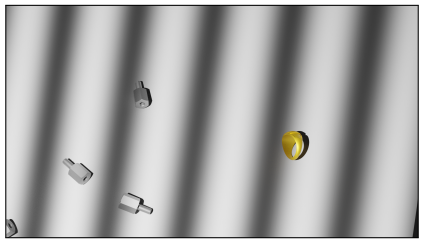
(b) Shift number 2 with 8 period pattern.



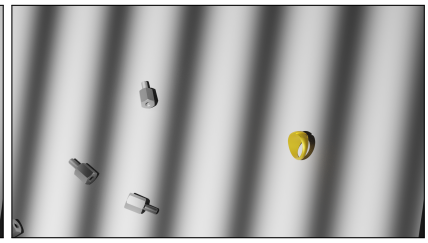
(c) Shift number 3 with 8 period pattern.



(d) Shift number 1 with 7 period pattern.



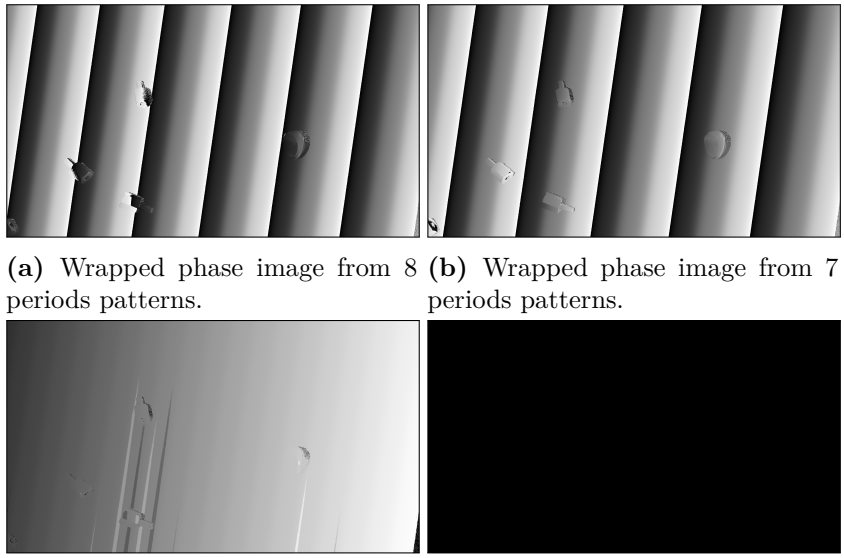
(e) Shift number 2 with 7 period pattern.



(f) Shift number 3 with 7 period pattern.

Figure 5.5.: Captured phase shifted patterns with period

The structured light algorithm is performed in steps, where each step builds on the obtained images from the previous. The intermediate results throughout the algorithm and the resulting depth image, can be viewed in Figures 5.6b to 5.6d, and are collected from a DataPipe run, as output taken while debugging. These images is not included in the regular output of the pipeline, except for the final depth image.



(a) Wrapped phase image from 8 periods patterns.

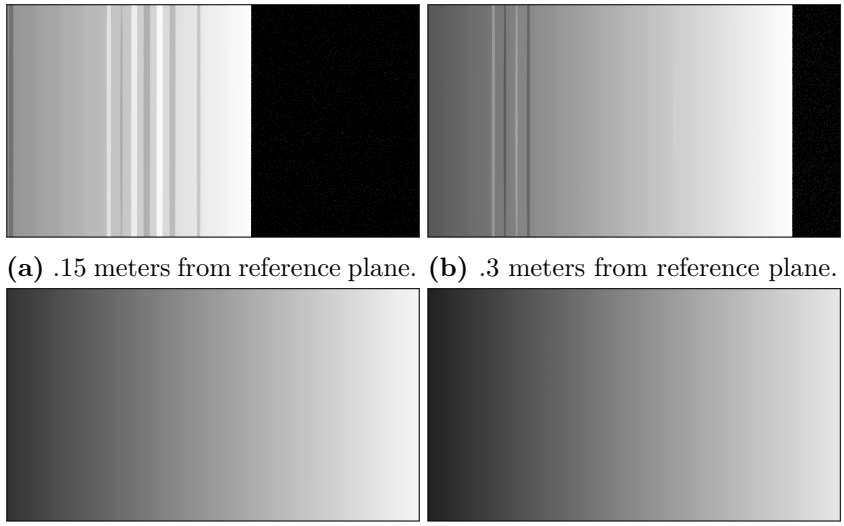
(b) Wrapped phase image from 7 periods patterns.

(c) Absolute phase image from 8 and 7 periods wrapped phases.

(d) Depth image from triangulation with projector.

Figure 5.6.: Images from the steps of the structured light algorithm.

Testing the effective range of the structured light setup used, where the projector is offset by 15cm in the negative y direction relative to the camera, and rotated 8.5 degrees about the camera x axis, is shown in Figure 5.7. A plane were scanned by the same structured light sensor at different distances to check the effective range of the setup.



(a) .15 meters from reference plane. **(b)** .3 meters from reference plane.

(c) .5 meters from reference plane. **(d)** .7 meters from reference plane.

Figure 5.7.: Absolute phase images obtained from projecting on reference plane at different distances.

Chapter 6.

Discussion

6.1. DataPipe add-on GUI

The choice to develop the pipeline as an add-on was taken to make the pipeline more user friendly and available. Add-ons are a large part of Blender, which has a large user base continuously developing new features that can be downloaded and extend Blenders basic functionalities. This same user base is already familiar with the GUI elements used in the pipeline add-on, and thus the threshold is lower to download and start generating data sets, that in turn can be shared with others, and thus making the selection of available data sets for computer vision tasks grow.

The GUI offers a lot of inputs for the pipeline process, which lets the user set the basis for the datasets that is to be generated. For the objects imported to the scene, the input is collected through the objects input panel in Figure 5.2a. The objects can be imported through the file explorer in the panel and previewed in the scene environment, and then scaled accordingly. In addition, the number of instances of each object can be controlled by setting the maximum and minimum number of instances of the object in each scene. This allows the datasets to display a large degree of randomization in object compositions in the rendered images.

The inputs for the scene in entered though the scene panel showed in Figure 5.1b, coltrolling the total number of images to render, as well as the maximum and minimum number of camera poses to render from for each scene. Other than this, the scene panel includes the option to import drop zone, which spawns an object that

6.2. Pipeline outputs

The quality of the pipeline outputs has not been quantitatively evaluated based on performance of a network trained on the data, but visual inspections of Figure 5.2 shows that the rendered data is of high quality, in resolution and in randomization of poses. These figures does not however display a large variety in object mix, but that is due to the lack of suitable 3D models used in that specific run of the pipeline.

from Figure 5.4b the color encoded normal vectors values are shown represented in RGB values. Many deep learning algorithms improve when the normals are introduced as input in addition to other inputs, as stated in the article by M. Denninger et al. [5], in addition the normals are rendered along with the z-pass depth image, so rendering it does not affect run time.

Figure 5.4d and 5.4c shows the masked image and z-pass depth image respectively, converted from OpenEXR format to human readable RGB images. The masked images originally consist of integer values corresponding to identifier indexes for each object. The pixels that contain the object are populated with the object index value. This type of image can be used in instance segmentation tasks.

The RGB output from Figure 5.4a is taken as output to further extend the usability of the datasets. The RGB images can be used in several different computer vision task, and will also add three data points per 3D point when used along with the 3D point cloud in pose estimation tasks.

6.3. Depth from structured light camera

The structured light camera implemented in the pipeline, shows promising results from the absolute phase and the wrapped phase images from the algorithm. In Figure 5.6a to 5.6c the intermediate steps for the structured light depth image shows the fringes being decoded and the absolute phase in Figure 5.6c being extracted though the phase unwrapping algorithm explained in Section 3.2.3.

The obtained absolute phase image shows what is mostly a successful absolute phase recovery, except for some misinterpreted phase values in the middle and towards the bottom of the absolute phase image. These misinterpretations are suspected to stem from the camera being too close or too far to the planar surface in the scene, and thus making the projected fringes “flatten” out, thus losing their variation in the horizontal direction.

This theory is further strengthened by viewing Figure 5.7 where the images captured are of the same plane, at different distances. The observed effect is the same

as in Figure 5.6c, where the projected patterns that are closest to the surface i.e. Figures 5.7a and 5.7b misinterprets the phase values because of the intensity of the projector light is stronger when it is close to the surface. This is not a problem however, since real scanners also have an optimum working distances, related to its intrinsic and extrinsic parameters, as for instance with the Zivid One+ Medium camera [41], which has a working distance between 0.6 and 2.0 meters.

As can be seen from what is supposed to be a structured light depth image in Figure 5.6d, is instead completely black. This is due to the triangulation algorithm only outputting zero values because of some, at this point unknown problem relating to the matrix multiplication of the triangulation algorithm from Section 3.3.2. It is possibly a small error in the code for the algorithm, but due to time limitations, it will not be investigated further in this report.

In hindsight, some of the inputs controlling the structured light system specifications should have been removed. The lack of predictability in the resolution, focal length, sensor size and extrinsic relation of the projector and camera objects, makes the creation of a generalized structured light system applicable for these inputs, almost impossible.

Chapter 7.

Conclusion

The main research objective for this report was to explore the possibilities of creating a working structured light camera inside Blender and incorporate it in the pipeline. This was done by utilizing the wide variety of Blender functionalities for light source objects and camera objects. The projector was modeled by a modified light source emitting pattern images onto a scene. The distortion in the patterns were then captured by the camera and passed through the phase detection and phase unwrapping algorithms developed through theoretical research. However, the triangulation of the obtained absolute phase was, as mentioned, not completed because of the time limitations of the project.

Despite the triangulation part of the algorithm not yielding any results, the intermediate steps was monitored and, and these resulting images was in line with the theory behind the algorithm. These intermediate results give a good indication that the use of a digital structured light scanner for creating reality-like noisy data that could potentially be a contributor to bridging the reality gap.

The two other research objectives was tied to the development of the pipeline process itself. The first was to create a pipeline process for creating textured data that mimic the challenges that are faced in bin picking. The pipeline process is fully functional at this stage, excluding the structured light camera functionality. The objects being dropped onto the scene creates a similar disorderly composition of poses and different 3D models.

The last research objective was to make the pipeline user friendly, so that there is a lower threshold for using the feature. As it happens, Blender incorporates a variety of GUI elements that can be used through Blenders bundled python distribution. This allowed for the creation of an add-on with a GUI tying the inputs to the corresponding operations carried out by the pipeline.

The creation of the GUI made it easier to guide the user to utilize the inputs

correctly by adding text boxes explaining the variables and making invalid inputs unavailable. In addition the whole input phase of the pipeline happens in the 3D view port of Blender, and previewing the object files that are taken as input, as well as camera poses is possible without running the pipeline. This makes the input process more dynamic and understandable. Lowering the threshold for using the pipeline could potentially drive people to try using it and thus contribute to the field of research by generating and sharing 3D data.

7.1. Further work

Naturally further work should be aimed at solving the problem with the triangulation algorithm, and thus obtaining a depth image that can be compared to the perfectly rendered depth from the z-pass of Blender. This can be done by obtaining the depth difference in each valid pixel in the structured light depth image and the corresponding pixel in the perfectly rendered z-pass depth.

As an improvement to the structured light camera, it would be interesting to find out how good the achieved quality of the produced data could be, by adopting a well developed structured light algorithm. In addition the large degree of user controlled input on the camera and projector intrinsic and extrinsic parameters, should be limited. The result of having user input for all these parameters is an unpredictable structured light algorithm, since these parameters are so tightly related to the output quality of the algorithm.

An interesting test of the synthetic structured light data would be to compare the imperfect structured light dataset with the z-buffer data and real data, applied in a 6D object pose estimation neural network algorithm. From these tests it would be interesting to see how the structured light data would perform against the widely used z-buffer data, and more importantly how the two would compare to real scanned datasets.

The pipeline as it is performs no domain randomization on the object textures, only in the composition and randomized object instances in the scene, as well as the physics based randomization of object poses. As each object is imported with textures to the pipeline, it would be beneficial to add randomizing parameters to these textures. For instance altering the object's surface shine, roughness, transparency and other parameters, and then testing the impact these have on the generalization of the model.

References

[1] R. Brégier, F. Devernay, L. Leyrit, and J. L. Crowley, "Symmetry aware evaluation of 3d object detection and pose estimation in scenes of many parts in bulk," in *2017 IEEE International Conference on Computer Vision Workshops (ICCVW)*, 2017, pp. 2209–2218. DOI: 10.1109/ICCVW.2017.258.

[2] T. Hodaň, V. Vineet, R. Gal, E. Shalev, J. Hanzelka, T. Connell, P. Urbina, S. N. Sinha, and B. Guenter, "Photorealistic image synthesis for object instance detection," in *2019 IEEE International Conference on Image Processing (ICIP)*, 2019, pp. 66–70. DOI: 10.1109/ICIP.2019.8803821.

[3] M. Jalal, J. Spjut, B. Boudaoud, and M. Betke, "Sidod: A synthetic image dataset for 3d object pose recognition with distractors," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2019, pp. 475–477. DOI: 10.1109/CVPRW.2019.00063.

[4] K. Group. (). "The industry's foundation for high performance graphics," [Online]. Available: https://www.opengl.org. (accessed: 14.12.2020).

[5] M. Denninger, M. Sundermeyer, D. Winkelbauer, Y. Zidan, D. Olefir, M. Elbadrawy, A. Lodhi, and H. Katam, *Blenderproc*, 2019. arXiv: 1911.01911 [cs.CV].

[6] J. Schell. (). "Cycles introduction," [Online]. Available: https://docs.blender.org/manual/en/latest/render/cycles/introduction.html. (accessed: 09.05.2021).

[7] T. Hodan, M. Sundermeyer, B. Drost, Y. Labbe, E. Brachmann, F. Michel, C. Rother, and J. Matas, *Bop challenge 2020 on 6d object localization*, 2020. arXiv: 2009.07378 [cs.CV].

[8] T. Hodaň, P. Haluza, Š. Obdržálek, J. Matas, M. Lourakis, and X. Zabulis, "T-LESS: An RGB-D dataset for 6D pose estimation of texture-less objects," *IEEE Winter Conference on Applications of Computer Vision (WACV)*, 2017.

[9] W. Eikrem, "Pipeline for generating synthetic 6d pose estimation datasets," Project Thesis, Dec. 2020.

[10] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning*. New York, NY: Springer, Jun. 2013, ISBN: 9781461471370.

[11] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[12] C. Choy, J. Gwak, and S. Savarese, *4d spatio-temporal convnets: Minkowski convolutional neural networks*, 2019. arXiv: 1904.08755 [cs.CV].

[13] A. Cariow, G. Cariowa, and D. Majorkowska-Mech, "An algorithm for quaternion–based 3d rotation," *International Journal of Applied Mathematics and Computer Science*, vol. 30, no. 1, pp. 149–160, 1Mar. 2020. DOI: https://doi.org/10.34768/amcs-2020-0012. [Online]. Available: https://content.sciendo.com/view/journals/amcs/30/1/article-p149.xml.

[14] K. Lynch, *Modern robotics : Mechanics, planning and control*, eng, Cambridge, 2017.

[15] A. Sveier, A. M. Sjøberg, and O. Egeland, "Applied runge–kutta–munthe-kaas integration for the quaternion kinematics," eng, *Journal of guidance, control, and dynamics*, vol. 42, no. 12, pp. 2747–2754, 2019, ISSN: 1533-3884.

[16] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017, pp. 23–30. DOI: 10.1109/IROS.2017.8202133.

[17] T. Bell, B. Li, and S. Zhang, "Structured light techniques and applications," in *Wiley Encyclopedia of Electrical and Electronics Engineering*. American Cancer Society, 2016, pp. 1–24, ISBN: 9780471346081. DOI: https://doi.org/10.1002/047134608X.W8298. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/047134608X.W8298. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/047134608X.W8298.

[18] S. Zhang, *Handbook of 3D machine vision: Optical metrology and imaging*. CRC press, 2013.

[19] C. Zuo, L. Huang, M. Zhang, Q. Chen, and A. Asundi, "Temporal phase unwrapping algorithms for fringe projection profilometry: A comparative review," *Optics and Lasers in Engineering*, vol. 85, pp. 84–103, 2016, ISSN: 0143-8166. DOI: https://doi.org/10.1016/j.optlaseng.2016.04.022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0143816616300653.

[20] Q. Tian, Y. Yang, X. Zhang, and B. Ge, "An experimental evaluation method for the performance of a laser line scanning system with multiple sensors," *Optics and lasers in engineering*, vol. 52, pp. 241–249, 2014, ISSN: 0143-8166.

[21] H. Schreiber and J. H. Bruning, "Phase shifting interferometry," in *Optical Shop Testing*. John Wiley & Sons, Ltd, 2007, ch. 14, pp. 547–666, ISBN: 9780470135976. DOI: https://doi.org/10.1002/9780470135976.ch14. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9780470135976.ch14. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470135976.ch14.

[22] K. Chen, J. Xi, and Y. Yu, "Quality-guided spatial phase unwrapping algorithm for fast three-dimensional measurement," *Optics Communications*, vol. 294, pp. 139–147, 2013, ISSN: 0030-4018. DOI: https://doi.org/10.1016/j.optcom.2013.01.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0030401813000254.

[23] C. Zuo, L. Huang, M. Zhang, Q. Chen, and A. Asundi, "Temporal phase unwrapping algorithms for fringe projection profilometry: A comparative review," *Optics and Lasers in Engineering*, vol. 85, pp. 84–103, 2016, ISSN: 0143-8166. DOI: https://doi.org/10.1016/j.optlaseng.2016.04.022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0143816616300653.

[24] ——, "Temporal phase unwrapping algorithms for fringe projection profilometry: A comparative review," *Optics and lasers in engineering*, vol. 85, pp. 84–103, 2016, ISSN: 0143-8166.

[25] A. Dipanda, S. Woo, F. Marzani, and J.-M. Bilbault, "3-d shape reconstruction in an active stereo vision system using genetic algorithms," *Pattern recognition*, vol. 36, no. 9, pp. 2143–2159, 2003.

[26] W. N. Klarquist and A. C. Bovik, "Fovea: A foveated vergent active stereo vision system for dynamic three-dimensional scene recovery," *IEEE Transactions on robotics and Automation*, vol. 14, no. 5, pp. 755–770, 1998.

[27] R. I. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521623049, 2000.

[28] V. Klema and A. Laub, "The singular value decomposition: Its computation and some applications," *IEEE transactions on automatic control*, vol. 25, no. 2, pp. 164–176, 1980, ISSN: 0018-9286.

[29] K. Liu, Y. Wang, D. L. Lau, Q. Hao, and L. G. Hassebrook, "Dual-frequency pattern scheme for high-speed 3-d shape measurement," *Optics express*, vol. 18, no. 5, pp. 5229–5244, 2010.

[30] M. Sundermeyer, Z.-C. Marton, M. Durner, M. Brucker, and R. Triebel, "Implicit 3d orientation learning for 6d object detection from rgb images," in *Proceedings of the European Conference on Computer Vision (ECCV)*, Sep. 2018.

[31] U. engine. (). "Real-time development platform | 3d, 2d, vr ar engine," [Online]. Available: https://unity.com. (accessed: 10.12.2020).

[32] B. community. (). "Blender - free and open 3d creation software," [Online]. Available: https://www.blender.org. (accessed: 15.10.2020).

[33] ——, (). "Blender - python api overview," [Online]. Available: https://docs.blender.org/api/current/info_overview.html. (accessed: 05.12.2020).

[34] S.-A. Dragly. (). "How structured light works - part 1," [Online]. Available: https://blog.zivid.com/how-structured-light-works-part-1. (accessed: 15.12.2020).

[35] S. community. (). "Numpy documentation discrete fourier transform," [Online]. Available: https://numpy.org/doc/stable/reference/generated/numpy.fft.fft.html. (accessed: 12.03.2021).

[36] A. S. Glassner, *An introduction to ray tracing*. Morgan Kaufmann, 1989.

[37] M. Carletti. (). "Extracting the intrinsic parameters in blender," [Online]. Available: https://mcarletti.github.io/articles/blenderintrinsicparams/. (accessed: 26.05.2021).

[38] O. Community. (). "Openexr homepage," [Online]. Available: https://www.openexr.com. (accessed: 09.06.2021).

[39] B. community. (). "Blender light objects," [Online]. Available: https://docs.blender.org/manual/en/2.83/render/lights/light_object.html. (accessed: 24.05.2021).

[40] J. S. Ocupe. (). "Projector add-on for blender," [Online]. Available: https://github.com/Ocupe/Projectors. (accessed: 10.03.2021).

[41] Zivid. (). "Zivid one+ medium," [Online]. Available: https://www.zivid.com/zivid-one-plus-medium-3d-camera. (accessed: 28.05.2021).

Appendix A.

Name of Appendix

A.1. Add-on ___init___ file

```
'''
This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTIBILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program. If not, see <http://www.gnu.org/licenses/>.
'''
print("########### __init__ Start ###########")
bl_info = {
    "name" : "DataPipe",
    "author" : "William Eikrem",
    "description" : "Pipeline addon for generating synthetic 3D
    datasets for use in training neural networks",
    "blender" : (2, 80, 0),
    "version" : (0, 0, 1),
    "location" : "View3D",
    "warning" : "",
    "category" : "Generic"
}

import bpy
from . import pipeline_panel
from . import pipeline_op



```

```
## This is needed for multi file addons. Otherwise only the __init__.py file will be reloaded.
# [Blender Logo] -> System -> Reload Scripts
if locals().get('loaded'):
    print("Addon was previously loaded. Force-reloading submodules.")
    loaded = False
    from importlib import reload
    from sys import modules
    modules[__name__] = reload(modules[__name__])
    for name, module in modules.items():
        if name.startswith(f"{__package__}."):
            globals()[name] = reload(module)
    del reload, modules

loaded = True

def register():
    try:
        pipeline_op.register()
        pipeline_panel.register()
    except RuntimeError as e:
        print(e)
        # This prevents having to restart blender if a registration fails midway.
        unregister()

# This is run when the addon is disabled.
def unregister():
    try:
        pipeline_op.unregister()
        pipeline_panel.unregister()
    except RuntimeError:
        pass

print("############ __init__ End ############")
```

Listing A.1: *init.py*

A.2. Add-on GUI panels script

```
from os import name
import bpy
import numpy as np

print("############ pipeline_panel Start ############")

class DATAPIPE_PT_Start_panel(bpy.types.Panel):
    bl_idname = 'Start_PT_Panel'
    bl_space_type = 'VIEW_3D'
```

```
    bl_region_type = 'UI'
    bl_category = 'DataPipe'
    bl_label = "Initialize pipeline"

    def draw(self, context):
        layout = self.layout

        row = layout.row()
        row.scale_y = 2
        row.operator('datapipe.set_scene_parameters', icon='PLAY')

        box = layout.box()
        box.label(text='Load pipeline input from pickle file')

        row = box.row()
        col = row.column()
        col.prop(context.scene, 'load_pipeline_input_path')
        col = row.column()
        col.scale_x = 0.7
        col.operator('datapipe.load_input_file', icon='IMPORT')


class DATAPIPE_PT_scenes_panel(bpy.types.Panel):
    bl_idname = 'Scenes_PT_Panel'
    bl_space_type = 'VIEW_3D'
    bl_region_type = 'UI'
    bl_category = 'DataPipe'
    bl_label = "Scene inputs"

    def draw(self, context):

        layout = self.layout
        box = layout.box()
        row = box.split(factor=0.6)
        left_col = row.column()
        right_col = row.column()

        #Total number of renders input
        row = left_col.row()
        row.alignment = 'RIGHT'
        row.label(text='Total number of renders')
        row = right_col.row()
        row.prop(context.scene, 'num_renders')

        #Max renders input
        row = left_col.row()
        row.alignment = 'RIGHT'
        row.label(text='Renders per scene Max')
        row = right_col.row()
        row.prop(context.scene, 'max_renders_per_scene')

```

```
61          #Min renders input
62          row = left_col.row()
63          row.alignment = 'RIGHT'
64          row.label(text='Min')
65          row = right_col.row()
66          row.prop(context.scene, 'min_renders_per_scene')
67
68          row = layout.row()
69
70          #Dropzone inputs
71          box = layout.box()
72          row = box.row()
73          row.operator('datapipe.import_dropzone_object', icon='
    PIVOT_BOUNDBOX')
74          row = box.row()
75          row.label(text='Move the box to desired location')
76
77
78
79
80  class DATAPIPE_PT_objects_panel(bpy.types.Panel):
81      bl_idname = 'Objects_PT_Panel'
82      bl_space_type = 'VIEW_3D'
83      bl_region_type = 'UI'
84      bl_category = 'DataPipe'
85      bl_label = "Objects inputs"
86
87      def draw(self, context):
88
89          layout = self.layout
90
91          row = layout.row()
92          row.label(text="Object filepath:", icon='OUTLINER_OB_MESH')
93          row = layout.row()
94          row.prop(context.scene, 'object_path')
95
96          box = layout.box()
97
98          row = box.split(factor=0.5)
99
100         left_col = row.column()
101         right_col = row.column()
102
103         row = left_col.row()
104         row.alignment = 'RIGHT'
105         row.label(text='Scaling factor')
106
107         row = left_col.row()
108         row.alignment = 'RIGHT'
109         row.label(text='Approx. mass')
110
```

```
111         row = left_col.row()
112         row.alignment = 'RIGHT'
113         row.label(text='Collision shape')
114 
115         row = left_col.row()
116         row.alignment = 'RIGHT'
117         row.label(text='Instances in scene Max')
118 
119         row = left_col.row()
120         row.alignment = 'RIGHT'
121         row.label(text='Min')
122 
123         row = right_col.row()
124         row.prop(context.scene, 'object_scale')
125 
126         row = right_col.row()
127         row.prop(context.scene, 'object_mass')
128 
129         row = right_col.row()
130         row.prop(context.scene, 'object_collision_shape')
131 
132         row = right_col.row()
133         row.prop(context.scene, 'object_instances_max')
134 
135         row = right_col.row()
136         row.prop(context.scene, 'object_instances_min')
137 
138         box = layout.box()
139         row = box.row()
140         row.operator('datapipe.preview_object', icon='WORKSPACE')
141 
142         row = box.row()
143 
144         row = box.row()
145         left_col = row.column()
146         right_col = row.column()
147 
148         row = left_col.row()
149         row.scale_y = 2
150         row.operator('datapipe.append_object_data', icon='FILE_NEW')
151 
152         row = right_col.row()
153         row.scale_y = 2
154         row.operator('datapipe.remove_last_object_data', icon='TRASH
    ')
155 
156 
157 
158 
159 class DATAPIPE_PT_camera_panel(bpy.types.Panel):
160     bl_idname = 'Camera_PT_Panel'
```

```
    bl_space_type = 'VIEW_3D'
    bl_region_type = 'UI'
    bl_category = 'DataPipe'
    bl_label = "Camera inputs"


    def draw(self, context):

        layout = self.layout

        #Camera intrinsics
        row = layout.row()
        row.label(text='Camera intrinsics', icon='CAMERA_DATA')
        row = layout.split(factor=0.5)

        left_col = row.column()
        right_col = row.column()

        #Focal length input
        row = left_col.row()
        row.alignment = 'RIGHT'
        row.label(text='Focal length')
        row = right_col.row()
        row.prop(context.scene, 'camera_focal_length')

        #Sensor width input
        row = left_col.row()
        row.alignment = 'RIGHT'
        row.label(text='Sensor width')
        row = right_col.row()
        row.prop(context.scene, 'camera_sensor_width')

        row = left_col.row()
        row.alignment = 'RIGHT'
        row.label(text='Resolution height')
        row = left_col.row()
        row.alignment = 'RIGHT'
        row.label(text='width')

        row = right_col.row()
        row.prop(context.scene, 'camera_resolution_height')

        row = right_col.row()
        row.prop(context.scene, 'camera_resolution_width')

        #Structured light
        row = layout.row()
        row.prop(context.scene, 'is_structured_light')

        if context.scene.is_structured_light:

```

```
212             #Projector intrinsics
213 
214             box = layout.box()
215             row = box.row()
216             row.label(text='Projector intrinsics', icon='LIGHT')
217             row = box.split(factor=0.5)
218 
219             left_col = row.column()
220             right_col = row.column()
221 
222             row = left_col.row()
223             row.alignment = 'RIGHT'
224             row.label(text='Focal length')
225             row = right_col.row()
226             row.prop(context.scene, 'projector_focal_length')
227 
228             row = left_col.row()
229             row.alignment = 'RIGHT'
230             row.label(text='Sensor width')
231             row = right_col.row()
232             row.prop(context.scene, 'projector_sensor_width')
233 
234             row = left_col.row()
235             row.alignment = 'RIGHT'
236             row.label(text='Resolution height')
237             right_col.column(align=True)
238             row = right_col.row()
239             row.prop(context.scene, 'projector_resolution_height')
240 
241             row = left_col.row()
242             row.alignment = 'RIGHT'
243             row.label(text='width')
244             row = right_col.row()
245             row.prop(context.scene, 'projector_resolution_width')
246 
247 
248             #Projector extrinsics
249             box = layout.box()
250             row = box.row()
251             row.label(text="Location (x, y, z) relative to camera",
    icon='EMPTY_AXIS')
252 
253             row = box.row()
254             row.prop(context.scene, 'projector_loc_vec')
255 
256             row = box.row()
257             row.label(text='Rotation relative to camera', icon='
    SPHERE')
258 
259             row = box.row()
260             row.prop(context.scene, 'projector_rot_enum')
```

```
261
262             row = box.row()
263             if context.scene.projector_rot_enum == 'xyz':
264                 row.prop(context.scene, 'projector_rot_xyz')
265             else:
266                 row.prop(context.scene, 'projector_rot_quat')
267
268         row = layout.row()
269         row.label(text='')
270
271         row = layout.row()
272         row.operator('datapipe.preview_camera_pose', icon='WORKSPACE
    ')
273
274         row = layout.row()
275         row.label(text='Move camera to desired pose')
276         row = layout.row()
277         row.scale_y = 2.0
278         row.operator('datapipe.append_camera_pose', icon='FILE_NEW')
279
280         col = row.column()
281         col.operator('datapipe.remove_camera_pose', icon='TRASH')
282
283
284
285
286 class DATAPIPE_PT_run_panel(bpy.types.Panel):
287     bl_idname = 'Run_PT_Panel'
288     bl_space_type = 'VIEW_3D'
289     bl_region_type = 'UI'
290     bl_category = 'DataPipe'
291     bl_label = "Run pipeline"
292
293     def draw(self, context):
294
295         layout = self.layout
296         box = layout.box()
297
298         row = box.row()
299         row.label(text="Save pipeline input to file")
300
301         row = box.row()
302         col = row.column()
303         col.prop(context.scene, 'save_pipeline_input_path')
304
305         col = row.column()
306         col.scale_x = 0.7
307         col.operator('datapipe.save_pipeline_info', icon='EXPORT')
308
309         row = layout.row()
310
```

```
311         box = layout.box()
312         row = box.row()
313         row.label(text='Save pipeline output to:')
314         row = box.row()
315         row.prop(context.scene, 'pipeline_output_path')
316 
317         row = box.row()
318         row = box.row()
319         row.operator('datapipe.run_pipeline', icon='PLAY')
320 
321 
322 
323 classes = [
324             DATAPIPE_PT_Start_panel,
325             DATAPIPE_PT_scenes_panel,
326             DATAPIPE_PT_objects_panel,
327             DATAPIPE_PT_camera_panel,
328             DATAPIPE_PT_run_panel
329           ]
330 
331 def register():
332     ################################################
333     ####### INITIALIZE PIPELINE PROPERTIES #######
334     ################################################
335     bpy.types.Scene.load_pipeline_input_path = bpy.props.
    StringProperty(
336         name='',
337         subtype='FILE_PATH',
338         description='Filepath to pickle file containing pipeline
    inputs')
339 
340     ################################
341     ####### SCENE PROPERTIES #######
342     ################################
343     bpy.types.Scene.num_renders = bpy.props.IntProperty(
344         name='',
345         default=1,
346         description='Number of images to render')
347     bpy.types.Scene.max_renders_per_scene = bpy.props.IntProperty(
348         name='',
349         default=1,
350         description='The max number of camera poses rendered per
    scene configuration')
351     bpy.types.Scene.min_renders_per_scene = bpy.props.IntProperty(
352         name='',
353         default=1,
354         description='The min number of camera poses rendered per
    scene configuration')
355 
356     ################################
357     ####### OBJECT PROPERTIES #######
```

```
358     ###################################
359     bpy.types.Scene.object_path = bpy.props.StringProperty(
360         name='',
361     subtype='FILE_PATH',
362     description='Filepath to object included in pipeline')
363     bpy.types.Scene.object_scale = bpy.props.FloatProperty(
364         name='',
365         soft_min=0,
366         precision=3,
367         description='Scaling of the object. Used to ensure that the
    3D model\'s units are equal to scene units',
368         default=1)
369     bpy.types.Scene.object_mass = bpy.props.FloatProperty(
370         name='',
371         soft_min=0,
372         default=0.5,
373         unit='MASS',
374         precision=3,
375         description='Approximate object mass, used by the simulated
    physics')
376     bpy.types.Scene.object_collision_shape = bpy.props.EnumProperty(
377         name='',
378         items=[('MESH', 'Mesh', ''),
379                ('CONVEX_HULL', 'Convex hull', '')],
380         description='Object\'s collision shape for physics
    simulation')
381     bpy.types.Scene.object_instances_max = bpy.props.IntProperty(
382         name='',
383         soft_min=0,
384         default=2,
385         description='Maximum number of object instance in each scene
    .')
386     bpy.types.Scene.object_instances_min = bpy.props.IntProperty(
387         name='',
388         soft_min=0,
389         default=0,
390         description='Minimum number of object instance in each scene
    .')
391 
392     ###################################
393     ######## CAMERA PROPERTIES ########
394     ###################################
395     bpy.types.Scene.camera_focal_length = bpy.props.FloatProperty(
396         name='',
397         unit='CAMERA')
398     bpy.types.Scene.camera_sensor_width = bpy.props.FloatProperty(
399         name='',
400         unit='CAMERA')
401     bpy.types.Scene.camera_resolution_height = bpy.props.IntProperty
    (
402         name='',
```

```
403         default= 480,
404         subtype='PIXEL')
405     bpy.types.Scene.camera_resolution_width = bpy.props.IntProperty(
406         name='',
407         default= 720,
408         subtype='PIXEL')
409     bpy.types.Scene.is_structured_light = bpy.props.BoolProperty(
410         name='Structured light',
411         description='Toggles on structured light rendering. The
    pipeline process will be slower, but the results will be
    realistic noise on rendered dataset')
412 
413     ####################################
414     ####### PROJECTOR PROPERTIES #######
415     ####################################
416     bpy.types.Scene.projector_focal_length = bpy.props.FloatProperty
    (
417         name='',
418         unit='CAMERA')
419     bpy.types.Scene.projector_sensor_width = bpy.props.FloatProperty
    (
420         name='',
421         unit='CAMERA')
422     bpy.types.Scene.projector_resolution_height = bpy.props.
    IntProperty(
423         name='',
424         default= 480,
425         subtype='PIXEL')
426     bpy.types.Scene.projector_resolution_width = bpy.props.
    IntProperty(
427         name='',
428         default= 720,
429         subtype='PIXEL')
430     bpy.types.Scene.projector_loc_vec = bpy.props.
    FloatVectorProperty(
431         name='',
432         default=(-0.15, 0, 0),
433         unit='LENGTH')
434     bpy.types.Scene.projector_rot_enum = bpy.props.EnumProperty(
435         name='',
436         items=[('xyz', 'XYZ Euler angles', ''),
437                ('quat', 'Quaternions [w, x, y, z]', '')])
438     bpy.types.Scene.projector_rot_xyz = bpy.props.
    FloatVectorProperty(
439         name='',
440         default= (0, -np.pi/180*8.5, 0),
441         unit='ROTATION')
442     bpy.types.Scene.projector_rot_quat = bpy.props.
    FloatVectorProperty(
443         name='',
444         default= (0.9972502, 0, -0.0741085, 0),
```

```
445         size=4)
446 
447     ########################################
448     ####### OUTPUT PATH PROPERTIES #######
449     ########################################
450     bpy.types.Scene.output_path = bpy.props.StringProperty(
451         name='',
452         subtype='DIR_PATH')
453 
454     ########################################
455     ####### RUN PIPELINE PROPERTIES #######
456     ########################################
457     bpy.types.Scene.save_pipeline_input_path = bpy.props.
    StringProperty(
458         name='',
459         subtype='DIR_PATH')
460     bpy.types.Scene.pipeline_output_path = bpy.props.StringProperty(
461         name='',
462         subtype='DIR_PATH',
463         description='Directory to save output from pipeline run.')
464 
465     for cl in classes:
466         bpy.utils.register_class(cl)
467 
468 def unregister():
469     ################################################
470     ####### INITIALIZE PIPELINE PROPERTIES #######
471     ################################################
472     del bpy.types.Scene.load_pipeline_input_path
473 
474     ################################
475     ####### SCENE PROPERTIES #######
476     ################################
477     del bpy.types.Scene.num_renders
478     del bpy.types.Scene.max_renders_per_scene
479     del bpy.types.Scene.min_renders_per_scene
480 
481     ################################
482     ####### OBJECT PROPERTIES #######
483     ################################
484     del bpy.types.Scene.object_path
485     del bpy.types.Scene.object_scale
486     del bpy.types.Scene.object_instances_max
487     del bpy.types.Scene.object_instances_min
488     del bpy.types.Scene.object_mass
489     del bpy.types.Scene.object_collision_shape
490 
491     ################################
492     ####### CAMERA PROPERTIES #######
493     ################################
494     del bpy.types.Scene.camera_focal_length
```

```
    del bpy.types.Scene.camera_sensor_width
    del bpy.types.Scene.camera_resolution_height
    del bpy.types.Scene.camera_resolution_width
    del bpy.types.Scene.is_structured_light

    ####################################
    ####### PROJECTOR PROPERTIES #######
    ####################################
    del bpy.types.Scene.projector_focal_length
    del bpy.types.Scene.projector_sensor_width
    del bpy.types.Scene.projector_resolution_height
    del bpy.types.Scene.projector_resolution_width
    del bpy.types.Scene.projector_loc_vec
    del bpy.types.Scene.projector_rot_enum
    del bpy.types.Scene.projector_rot_xyz
    del bpy.types.Scene.projector_rot_quat

    ########################################
    ####### OUTPUT PATH PROPERTIES #######
    ########################################
    del bpy.types.Scene.output_path

    ########################################
    ####### RUN PIPELINE PROPERTIES #######
    ########################################
    del bpy.types.Scene.save_pipeline_input_path
    del bpy.types.Scene.pipeline_output_path

    for cl in classes:
        bpy.utils.unregister_class(cl)

print("############ pipeline_panel End ############")
```

Listing A.2: pipeline$_{p}$*anel.py*

A.3. Add-on GUI operators script

```
print("############ pipeline_op Start ############")
from pathlib import Path
import bpy
from .src import utility_fuctions
from .src import config_module
from .src.camera_module import BlendCamera
from .src.scene_module import BlendScene
from .src.objects_module import ObjectManager
from .src.render_module import Renderer
from .src.simulation_module import Simulation
from .src.structured_light_module import Algorithm
import numpy as np
import time
```

```
14
15 ############## DO THIS FOR PIPELINE CLASSES ##############
16 #from .src import random_object
17
18
19 ############# SCENE OPERATORS #############
20 class DATAPIPE_OT_Set_up_Scene(bpy.types.Operator):
21
22     bl_idname = 'datapipe.set_scene_parameters'
23     bl_label = 'Initialize pipeline'
24     bl_options = {'REGISTER', 'UNDO'}
25
26     def execute(self, context):
27
28         utility_fuctions.initialize_pipeline_environment()
29
30         return {'FINISHED'}
31
32 class DATAPIPE_OT_Load_input_file(bpy.types.Operator):
33
34     bl_idname = 'datapipe.load_input_file'
35     bl_label = 'Load'
36     bl_options = {'REGISTER', 'UNDO'}
37
38     def execute(self, context):
39
40         #Load all input from file to configmodule.input_storage
41         config_module.input_storage.input_from_file(context.scene.
    load_pipeline_input_path)
42
43         config_module.input_storage.set_input_panel_vars_from_dict(
    context)
44
45         return {'FINISHED'}
46
47 class DATAPIPE_OT_Import_dropzone_object(bpy.types.Operator):
48
49     bl_idname = 'datapipe.import_dropzone_object'
50     bl_label = 'Import dropzone'
51     bl_options = {'REGISTER', 'UNDO'}
52
53     def execute(self, context):
54
55         if 'drop_zone' not in bpy.data.objects:
56             bpy.ops.mesh.primitive_cube_add()
57             drop_zone = bpy.context.active_object
58             mesh_name = drop_zone.name
59             drop_zone.name = 'drop_zone'
60             bpy.data.meshes[mesh_name].name = 'drop_zone_mesh'
61             drop_zone.location = (0, 0, 2)
62             drop_zone.scale = (0.5, 0.5, 0.5)
```

```
63
64              bpy.ops.object.select_all(action='DESELECT')
65
66              if drop_zone.select_get() is False:
67                  drop_zone.select_set(True)
68
69              bpy.context.view_layer.objects.active = drop_zone
70              bpy.ops.object.select_all(action='DESELECT')
71
72          return {'FINISHED'}
73
74 ############## OBJECT OPERATORS ##############
75 class DATAPIPE_OT_Preview_object(bpy.types.Operator):
76
77     bl_idname = 'datapipe.preview_object'
78     bl_label = 'Toggle object preview'
79     bl_options = {'REGISTER', 'UNDO'}
80
81     def execute(self, context):
82
83         filepath = Path(bpy.path.abspath(context.scene.object_path))
    .resolve() #Filepath from input
84
85         scale = context.scene.object_scale
86         ob_scale = [scale, scale, scale] #object scale from input
87
88
89         if filepath.suffix == '.obj': #Filetype has to be .obj
90
91             if 'temp_collection' not in bpy.data.collections.keys():
     #Check for earlier import
92
93                 temp_collection = bpy.data.collections.new(name='
    temp_collection')
94                 bpy.context.scene.collection.children.link(
    temp_collection) #Link to scene collection
95
96             temp_collection = bpy.data.collections['temp_collection'
    ]
97
98             if 'temp_object' not in bpy.data.objects.keys() and '
    temp_material' not in bpy.data.materials.keys():
99                 bpy.ops.import_scene.obj(filepath=str(filepath))
100
101                 if len(bpy.context.selected_objects) != 1: #Only one
     object can be imported at a time
102                     for ob in bpy.context.selected_objects:
103                         #Remove belonging data
104                         bpy.data.materials.remove(ob.active_material
    , do_unlink=True)
105                         bpy.data.objects.remove(ob, do_unlink=True)
```

```
106                         bpy.data.meshes.remove(bpy.data.meshes[ob.
    name], do_unlink=True)
107                     print("Input .obj file can only contain one
    object")
108                 else:
109                     ob = bpy.context.selected_objects[0] #Get object
110 
111                     ob.users_collection[0].objects.unlink(ob) #
    Remove object from default collection
112                     temp_collection.objects.link(ob) #Link to temp
    collection
113 
114                     mesh_name = ob.name #Store current mesh name
115 
116                     ob.name = 'temp_object'
117                     ob.scale = ob_scale #Set scale from input
118 
119                     ob_mat = ob.active_material #ob material data
120                     ob_mat.name = 'temp_material'
121 
122                     bpy.data.meshes[mesh_name].name = 'temp_mesh' #
    ob mesh data
123 
124             else:
125                 if 'temp_object' in bpy.data.objects:
126                     bpy.data.objects.remove(bpy.data.objects['
    temp_object'], do_unlink=True)
127                     bpy.data.meshes.remove(bpy.data.meshes['
    temp_mesh'], do_unlink=True)
128                     if 'temp_material' in bpy.data.materials:
129                         bpy.data.materials.remove(bpy.data.materials
    ['temp_material'], do_unlink=True)
130                 else:
131                     bpy.data.materials.remove(bpy.data.materials['
    temp_material'], do_unlink=True)
132         else:
133             print("Required filetype for object is \".obj\".")
134 
135         return {'FINISHED'}
136 
137 class DATAPIPE_OT_Append_object_data(bpy.types.Operator):
138 
139     bl_idname = 'datapipe.append_object_data'
140     bl_label = 'Append object to pipeline'
141     bl_options = {'REGISTER', 'UNDO'}
142 
143     def execute(self, context):
144         object_config = config_module.input_storage.config_dict['
    objects'] #Get object config
145         objects_list = object_config['objects_list']
146 
```

```
147         #User inputs
148         object_filepath = Path(bpy.path.abspath(context.scene.
    object_path)).resolve()
149         object_scale = context.scene.object_scale
150         object_mass = context.scene.object_mass
151         object_collision_shape = context.scene.
    object_collision_shape
152         object_instances_max = context.scene.object_instances_max
153         object_instances_min = context.scene.object_instances_min
154 
155         objects_list.append({'filepath': object_filepath, 'scale':
    object_scale, 'mass': object_mass, 'collision_shape':
    object_collision_shape, 'max': object_instances_max, 'min':
    object_instances_min}) #Append user input to config dict
156 
157         if 'temp_object' in bpy.data.objects.keys(): #Remove object
    data
158             bpy.data.objects.remove(bpy.data.objects['temp_object'],
     do_unlink=True)
159             bpy.data.materials.remove(bpy.data.materials['
    temp_material'], do_unlink=True)
160             bpy.data.meshes.remove(bpy.data.meshes['temp_mesh'],
    do_unlink=True)
161 
162         return {'FINISHED'}
163 
164 
165 class DATAPIPE_OT_Remove_last_object_data(bpy.types.Operator):
166 
167     bl_idname = 'datapipe.remove_last_object_data'
168     bl_label = 'Remove last'
169     bl_options = {'REGISTER', 'UNDO'}
170 
171     def execute(self, context):
172         object_config = config_module.input_storage.config_dict['
    objects']
173         object_list = object_config['objects_list']
174 
175         if len(object_list) > 0: #Check if any objects in list
176             del object_list[-1] #Remove last object
177 
178         return {'FINISHED'}
179 
180 
181 ############## CAMERA OPERATORS ##############
182 class DATAPIPE_OT_Append_camera_pose(bpy.types.Operator):
183 
184     bl_idname = 'datapipe.append_camera_pose'
185     bl_label = 'Append pose'
186     bl_options = {'REGISTER', 'UNDO'}
187 
```

```
188     def execute(self, context):
189 
190         camera_pose_list = config_module.input_storage.config_dict['
    camera']['wrld2cam_pose_list']
191 
192         if 'temp_cam' in bpy.data.cameras.keys():
193             cam = bpy.data.objects['temp_cam']
194 
195             loc = list(cam.location)
196             cam.rotation_mode = 'QUATERNION'
197             rot = list(cam.rotation_quaternion)
198 
199             transform = {'rotation': rot, 'location': loc}
200 
201             camera_pose_list.append(transform)
202 
203             ###### reseting camera pose props to zero
204             cam.location = (0, 0, 1)
205             cam.rotation_mode = 'XYZ'
206             cam.rotation_euler = (np.pi/2, 0, 0)
207 
208             print("\n### New pose added to camera pose list ###\n->
    Utility camera pose list contains {} poses\n".format(len(
    config_module.input_storage.config_dict['camera']['
    wrld2cam_pose_list'])))
209 
210         return {'FINISHED'}
211 
212 class DATAPIPE_OT_Remove_camera_pose(bpy.types.Operator):
213 
214     bl_idname = 'datapipe.remove_camera_pose'
215     bl_label = 'Undo last'
216     bl_options = {'REGISTER', 'UNDO'}
217 
218     def execute(self, context):
219 
220         camera_pose_list = config_module.input_storage.config_dict['
    camera']['wrld2cam_pose_list']
221 
222         if len(camera_pose_list) > 0:
223             del camera_pose_list[-1]
224             print("Number of poses in list after removal: {}".format
    (len(camera_pose_list)))
225 
226         return {'FINISHED'}
227 
228 
229 class DATAPIPE_OT_Preview_camera_pose(bpy.types.Operator):
230 
231     bl_idname = 'datapipe.preview_camera_pose'
232     bl_label = 'Toggle camera preview'
```

```
233
234     def execute(self, context):
235
236         if 'temp_cam' in bpy.data.cameras.keys(): #Check if camera
    and projector temp object already exists
237             #Remove camera object when toggeling off preview
238             bpy.data.objects.remove(bpy.data.objects['temp_cam'],
    do_unlink=True)
239             bpy.data.cameras.remove(bpy.data.cameras['temp_cam'],
    do_unlink=True)
240
241             if 'temp_projector' in bpy.data.lights.keys():
242             #Remove projector object when toggeling off preview
243                 bpy.data.objects.remove(bpy.data.objects['
    temp_projector'], do_unlink=True)
244                 bpy.data.lights.remove(bpy.data.lights['
    temp_projector'], do_unlink=True)
245         else:
246             #Insert camera object in specified position for preview
247             cam = bpy.data.cameras.new(name='temp_cam') #Create
    camera data
248             cam_obj = bpy.data.objects.new(name='temp_cam',
    object_data=cam) #Create camera object
249             bpy.context.scene.collection.objects.link(cam_obj)
250
251             cam.sensor_width = context.scene.camera_sensor_width
252             cam.lens = context.scene.camera_focal_length
253
254             #Set default pose
255             cam_obj.location = (0,0,1)
256             cam_obj.rotation_euler = (np.pi/2, 0, 0)
257
258             #Check if projector pose should be previewed
259             if context.scene.is_structured_light:
260
261                 projector = bpy.data.lights.new(name='temp_projector
    ', type='SPOT')
262                 projector_obj = bpy.data.objects.new(name='
    temp_projector', object_data=projector)
263                 bpy.context.scene.collection.objects.link(
    projector_obj)
264
265                 projector_obj.parent = cam_obj #Set projector to
    child of camera object
266                 projector_obj.parent_type = 'OBJECT'
267
268                 projector.spot_blend = 0
269                 projector.spot_size = 45*np.pi/180
270                 projector.shadow_soft_size = 0
271
272                 proj_loc = context.scene.projector_loc_vec
```

```
273                 projector_obj.location = proj_loc
274 
275                 #Check rotation mode of projector
276                 if context.scene.projector_rot_enum == 'quat':
277                     projector_obj.rotation_mode = 'QUATERNION'
278                     proj_rot = context.scene.projector_rot_quat
279                     projector_obj.rotation_quaternion = proj_rot
280                 else:
281                     projector_obj.rotation_mode = 'XYZ'
282                     proj_rot = context.scene.projector_rot_xyz
283                     projector_obj.rotation_euler = proj_rot
284 
285         return {'FINISHED'}
286 
287 
288 class DATAPIPE_OT_Save_pipeline_info(bpy.types.Operator):
289 
290     bl_idname = 'datapipe.save_pipeline_info'
291     bl_label = 'Export'
292 
293     def execute(self, context):
294 
295         config_module.input_storage.write_to_config_dict(context)
296 
297         config_module.input_storage.write_to_pickle_file(context.
    scene.save_pipeline_input_path)
298 
299         return {'FINISHED'}
300 
301 
302 ############## RUN PIPELINE OPERATORS ##############
303 class DATAPIPE_OT_Runner(bpy.types.Operator):
304 
305     bl_idname = 'datapipe.run_pipeline'
306     bl_label = 'Run Pipeline'
307     bl_options = {'REGISTER', 'UNDO'}
308 
309     def execute(self, context):
310 
311         # Remove temp objects before running the pipeline, to avoid
    bugs.
312         if 'temp_projector' in bpy.data.lights.keys(): #Remove
    projector object preview before running the pipeline
313             bpy.data.objects.remove(bpy.data.objects['temp_projector
    '], do_unlink=True)
314             bpy.data.lights.remove(bpy.data.lights['temp_projector'
    ], do_unlink=True)
315 
316         if 'temp_cam' in bpy.data.cameras.keys(): #Remove camera
    object preview before running the pipeline
317             bpy.data.objects.remove(bpy.data.objects['temp_cam'],
```

```
    do_unlink=True)
318             bpy.data.cameras.remove(bpy.data.cameras['temp_cam'],
    do_unlink=True)
319 
320         if 'temp_object' in bpy.data.objects.keys(): #Remove object
    preview before running the pipeline
321             bpy.data.materials.remove(bpy.data.materials['
    temp_material'], do_unlink=True)
322             bpy.data.objects.remove(bpy.data.objects['temp_object'],
     do_unlink=True)
323             bpy.data.meshes.remove(bpy.data.meshes['temp_mesh'],
    do_unlink=True)
324 
325         if 'temp_collection' in bpy.data.collections.keys(): #Remove
     collections preview before running the pipeline
326             bpy.data.collections.remove(bpy.data.collections['
    temp_collection'], do_unlink=True)
327 
328         # Load all information from GUI to config dict
329         config_module.input_storage.write_to_config_dict(context)
330 
331         ##################### PIPELINE FROM HERE ON OUT
    ####################
332         print("\n####################\nPIPELINE RUN INITIATED\n
    ####################\n")
333         start_time = time.time()
334 
335         config = config_module.input_storage.config_dict
336 
337         camera = BlendCamera(camera_name='pipe_cam',config=config)
338 
339         renderer = Renderer(camera=camera)
340 
341         object_manager = ObjectManager(config=config)
342 
343         loop_finished = False
344 
345         while not loop_finished:
346 
347             #Prep simulation goes here.
348 
349             scene = BlendScene(config=config, camera=camera,
    object_manager=object_manager)
350 
351             object_manager.import_objects()
352 
353             object_manager.create_initial_positions(scene=scene)
354 
355             #Simulation goes here
356             simulation = Simulation(sim_end=400)
357             simulation.run_loop() #Loop physics simulation
```

```
358             simulation.apply_simulated_transforms(object_manager=
    object_manager)
359 
360             for render in range(1,scene.renders_for_scene+1):
361 
362                 renderer.set_output_paths(scene=scene, render_num=
    render)
363 
364                 camera.move(curr_render=render)
365 
366                 scene.write_output_info_to_scene_dict(render_num=
    render, camera=camera, object_manager=object_manager)
367 
368                 renderer.render_results()
369 
370                 if camera.is_structured_light:
371                     SL_algorithm = Algorithm(renderer=renderer,
372                                              pattern_names=camera.
    pattern_names,
373                                              pattern_generator=
    camera.pattern_generator,camera=camera)
374 
375 
376             scene.write_scene_dict_to_file()
377 
378             loop_finished = scene.last_scene
379 
380             del scene
381 
382         BlendScene.reset_scene_number()
383         config_module.input_storage.reset_config_dict()
384 
385         end_time = time.time()
386 
387         print("###################\n# Timing results #\n# Run time:
    {:2.2f} #\n###################\n".format(end_time-start_time))
388         print("\nPIPELINE RUN FINISHED\n")
389         return {'FINISHED'}
390 
391 
392 classes = [
393             DATAPIPE_OT_Set_up_Scene,
394             DATAPIPE_OT_Load_input_file,
395             DATAPIPE_OT_Import_dropzone_object,
396             DATAPIPE_OT_Preview_object,
397             DATAPIPE_OT_Append_object_data,
398             DATAPIPE_OT_Remove_last_object_data,
399             DATAPIPE_OT_Preview_camera_pose,
400             DATAPIPE_OT_Append_camera_pose,
401             DATAPIPE_OT_Remove_camera_pose,
402             DATAPIPE_OT_Save_pipeline_info,
```

```
403             DATAPIPE_OT_Runner,
404           ]
405
406 def register():
407     for cl in classes:
408         try:
409             bpy.utils.register_class(cl)
410         except RuntimeError as e:
411             print(e)
412
413 def unregister():
414     for cl in classes:
415         try:
416             bpy.utils.unregister_class(cl)
417         except RuntimeError:
418             pass
419
420 print("############ pipeline_op End ############")
```

Listing A.3: pipeline$_op$.*py*

A.4. Scene class

```
1
2  import bpy
3  import random
4  from pathlib import Path
5  import numpy as np
6  import pickle
7
8  from .objects_module import ObjectManager
9  from .camera_module import BlendCamera
10
11 class BlendScene:
12
13     scene_num = 0
14     finished_renders = 0
15
16     run_instance_path = ''
17
18     def __init__(self, config: dict, camera: BlendCamera,
       object_manager: ObjectManager):
19         self.scene_config = config['scene']
20
21         self.object_manager = object_manager
22
23         self.scene_number = self.get_scene_number()
24         self.scene_name = "scene.{:04}".format(self.scene_num)
25
26         print("### SCENE {} CREATED ###".format(self.scene_num))
```

```
27
28          if self.scene_num == 1:
29              self.set_run_instance_path(config=config)
30
31          self.run_instance_path = self.get_run_instance_path()
32
33          self.drop_zone_location, self.drop_zone_dimensions = self.
    get_drop_zone_info(scene_config=self.scene_config)
34
35          self.output_path = Path.joinpath(Path(config['output']['path
    ']), self.scene_name)
36          self.output_path.mkdir()
37
38          self.last_scene = False
39
40          self.total_num_renders = self.scene_config['num_renders']
41
42          self.renders_for_scene = self.get_num_renders(camera=camera)
43
44          self.scene_dict = {}
45
46      @classmethod
47      def get_scene_number(cls):
48          temp = cls.scene_num
49          cls.scene_num += 1
50          return cls.scene_num
51
52      @classmethod
53      def reset_scene_number(cls):
54          cls.scene_num = 0
55
56      @classmethod
57      def set_run_instance_path(cls, config):
58
59          path = Path(config['output']['path'])
60          path.mkdir()
61          cls.run_instance_path = str(path)
62
63      @classmethod
64      def get_run_instance_path(cls):
65          return cls.run_instance_path
66
67      @classmethod
68      def set_finished_renders(cls, renders_for_scene: int):
69          cls.finished_renders += renders_for_scene
70
71      @classmethod
72      def get_num_finished_renders(cls):
73          return cls.finished_renders
74
75      def get_drop_zone_info(self, scene_config: dict):
```

```
76
77          if 'drop_zone' in bpy.data.objects.keys():
78
79              drop_zone_ob = bpy.data.objects['drop_zone']
80              bpy.ops.object.select_all(action='DESELECT')
81              drop_zone_ob.select_get()
82              bpy.ops.object.transform_apply(location=False, scale=
    True, rotation=False)
83              scale = drop_zone_ob.scale
84              drop_zone_loc = drop_zone_ob.location
85              drop_zone_dim = drop_zone_ob.dimensions
86
87              bpy.data.objects.remove(drop_zone_ob, do_unlink=True)
88              bpy.data.meshes.remove(bpy.data.meshes['drop_zone_mesh'
    ], do_unlink=True)
89
90          else:
91              drop_zone_loc = list(scene_config['drop_zone_loc'])
92              scale = scene_config['drop_zone_scale']
93              drop_zone_dim = [2*scale[0], 2*scale[1], 2*scale[2]]
94
95          if self.scene_num == 1:
96              #Add cube under dropzone, to avoid simulation cliping
    through
97              bpy.ops.mesh.primitive_cube_add()
98              cube = bpy.context.active_object
99              old_col = cube.users_collection[0]
100             self.object_manager.objects_collection.objects.link(cube
    )
101             old_col.objects.unlink(cube)
102
103             bpy.data.meshes[cube.name].name = 'occlusion_box'
104             cube.name = 'occlusion_box'
105             bpy.ops.rigidbody.object_add(type='PASSIVE')
106             cube_scale = [scale[0],scale[1], 0.1]
107
108             cube.hide_render = True
109             cube.scale = cube_scale
110
111             cube.location = [drop_zone_loc[0], drop_zone_loc[1],
    drop_zone_loc[2] - drop_zone_dim[2]/2 - 0.1]
112             cube.rigid_body.collision_margin = 0.001
113
114         return list(drop_zone_loc), list(drop_zone_dim)
115
116
117     def get_num_renders(self, camera: object):
118
119         max_renders = self.scene_config['max_renders_per_scene']
120         min_renders = self.scene_config['min_renders_per_scene']
121
```

```
122         if max_renders > len(camera.pose_list):
123             max_renders = len(camera.pose_list)
124 
125         num_renders = random.randint(a=min_renders, b=max_renders)
126         finished_renders = self.get_num_finished_renders()
127 
128         if finished_renders + num_renders >= self.total_num_renders:
129             num_renders = self.total_num_renders - finished_renders
130 
131             self.last_scene = True
132 
133         self.set_finished_renders(num_renders)
134 
135         return num_renders
136 
137     def write_output_info_to_scene_dict(self, render_num: int,
    camera: BlendCamera, object_manager: ObjectManager):
138         print("\nWriting to output dict:\n")
139         object_output_list = object_manager.
    get_objects_information_dict(camera=camera)
140         render_key = 'render.{:04d}'.format(render_num)
141 
142         wrld2cam_pose = np.asarray(camera.blend_cam_obj.matrix_world
    )
143 
144         self.scene_dict[render_key] = {'wrld2cam_pose':
    wrld2cam_pose,
145                                        'objects_in_scene':
    object_output_list}
146 
147     def write_scene_dict_to_file(self):
148 
149         filename = Path('output_dict.pickle')
150         output_path = Path(self.run_instance_path)
151 
152         pickle_path = Path.joinpath(output_path, filename)
153 
154         if pickle_path.exists():
155             path_str = str(pickle_path)
156             with open(path_str, "rb") as pickle_file:
157 
158                 info_dict = pickle.load(pickle_file)
159 
160                 info_dict[self.scene_name] = self.scene_dict
161 
162             with open(path_str, "wb") as pickle_file_out:
163 
164                 pickle.dump(info_dict, pickle_file_out)
165         else:
166             path_str = str(pickle_path)
167             with open(path_str, "wb") as pickle_file_out:
```

```
                info_dict = {self.scene_name: self.scene_dict}

                pickle.dump(info_dict, pickle_file_out)
```

Listing A.4: BlendScene

A.5. Camera class

```
import bpy
import numpy as np
import random
import copy

from .projector_module import Projector


class BlendCamera:

    def __init__(self, camera_name: str, config: dict):
        print("### Camera object created ###")

        self.camera_config = config['camera']

        self.name = camera_name

        self.is_structured_light = self.camera_config["
    is_structured_light"]

        self.pose_list = self.camera_config["wrld2cam_pose_list"]
        self.pose_list_copy = copy.copy(self.pose_list)

        self.blend_cam_obj, self.blend_cam_data = self.import_camera
    ()

        bpy.context.scene.camera = self.blend_cam_obj

        if self.is_structured_light:
            self.projector = Projector(self, config=config)
            self.pattern_names = self.projector.pattern_names_list
            self.pattern_generator = self.projector.
    pattern_generator
            self.projector_matrix = self.projector.
    get_projector_matrix()
            M = np.asarray(bpy.data.objects[self.projector.
    pattern_names_list[0]].matrix_basis)
            self.projector_extrinsic = M[:3,:]

        self.K = self.get_calibration_matrix_K_from_blender(mode='
    complete')
        self.camera_extrinsic = np.array([[1, 0, 0, 0],
```

```
                                            [0, 1, 0, 0],
                                            [0, 0, 1, 0]])


    def import_camera(self):
        """
        Imports camera into blender scene.
        """
        camera = bpy.data.cameras.new(name=self.name)
        camera_obj = bpy.data.objects.new(name=self.name,
    object_data=camera)

        parent_col = bpy.context.scene.collection
        parent_col.objects.link(camera_obj)

        camera_obj.rotation_mode = 'QUATERNION'

        return camera_obj, camera



    def get_random_pose_from_list(self, curr_render: int):
        """
        Selects a random pose from camera pose list.

        Returns random tansformation numpy matrix.
        """
        if curr_render == 1:
            self.pose_list_copy = copy.copy(self.pose_list)

        #Pick random index of pose list
        index = random.randint(a=0, b=len(self.pose_list_copy)-1)

        pose = self.pose_list_copy.pop(index)

        return pose

    def move(self, curr_render: int):
        """
        moves camera object to random position.
        """
        print("\n###### MOVING CAMERA ######")

        #Sample random pose from input list
        rand_pose = self.get_random_pose_from_list(curr_render=
    curr_render)

        self.blend_cam_obj.location = rand_pose['location']
        self.blend_cam_obj.rotation_quaternion = rand_pose['rotation
    ']

```

```
85          bpy.context.view_layer.objects.active = self.blend_cam_obj
86          bpy.ops.object.visual_transform_apply()
87
88
89      def get_calibration_matrix_K_from_blender(self, mode='complete')
    :
90
91          scene = bpy.context.scene
92
93          scale = scene.render.resolution_percentage / 100
94          width = scene.render.resolution_x * scale # px
95          height = scene.render.resolution_y * scale # px
96
97          camdata = scene.camera.data
98
99          if mode == 'simple':
100
101             aspect_ratio = width / height
102             K = np.zeros((3,3), dtype=np.float32)
103             K[0][0] = width / 2 / np.tan(camdata.angle / 2)
104             K[1][1] = height / 2. / np.tan(camdata.angle / 2) *
    aspect_ratio
105             K[0][2] = width / 2.
106             K[1][2] = height / 2.
107             K[2][2] = 1.
108             K.transpose()
109
110         if mode == 'complete':
111
112             focal = camdata.lens # mm
113             sensor_width = camdata.sensor_width # mm
114             sensor_height = camdata.sensor_height # mm
115             pixel_aspect_ratio = scene.render.pixel_aspect_x / scene
    .render.pixel_aspect_y
116
117             if (camdata.sensor_fit == 'VERTICAL'):
118                 # the sensor height is fixed (sensor fit is
    horizontal),
119                 # the sensor width is effectively changed with the
    pixel aspect ratio
120                 s_u = width / sensor_width / pixel_aspect_ratio
121                 s_v = height / sensor_height
122             else: # 'HORIZONTAL' and 'AUTO'
123                 # the sensor width is fixed (sensor fit is
    horizontal),
124                 # the sensor height is effectively changed with the
    pixel aspect ratio
125                 pixel_aspect_ratio = scene.render.pixel_aspect_x /
    scene.render.pixel_aspect_y
126                 s_u = width / sensor_width
127                 s_v = height * pixel_aspect_ratio / sensor_height
```

```
128
129             # parameters of intrinsic calibration matrix K
130             alpha_u = focal * s_u
131             alpha_v = focal * s_v
132             u_0 = width / 2
133             v_0 = height / 2
134             skew = 0 # only use rectangular pixels
135
136             K = np.array([
137                 [alpha_u,    skew, u_0],
138                 [      0, alpha_v, v_0],
139                 [      0,       0,   1]
140             ], dtype=np.float32)
141
142         return K
```

Listing A.5: BlendCamera

A.6. Projector class

```
1  import os
2  import bpy
3  import numpy as np
4  from pathlib import Path
5
6  from . import utility_fuctions
7  from .pattern_generator import PatternGenerator
8
9  print("########## Projector start ##########")
10
11 class Projector:
12     """
13     Projector class for Blender pipeline
14     """
15
16     def __init__(self, blend_camera: object, config: dict):
17         print("### Projector object created ###")
18
19         self.projector_config = config['projector']
20
21         #Collect intrinsics from config
22         self.pattern_shape = self.projector_config['resolution']
23         self.focal_length = self.projector_config['focal_length']
24         self.sensor_size_horizontal = self.projector_config['
    sensor_width']
25
26         #Generate patterns if they dont exist
27         self.pattern_generator = PatternGenerator(resolution=self.
    pattern_shape)
```

```
28          self.patterns = self.pattern_generator.patterns_list #List
    of pattern dicts
29
30          self.cam2proj_rot = self.projector_config['proj2cam_pose']['
    rotation']
31          self.cam2proj_loc = self.projector_config['proj2cam_pose']['
    location']
32
33          print("\n### Projector pose ###\n--> Quaternions: {}\n-->
    Translation: {}\n".format(self.cam2proj_rot, self.cam2proj_loc))
34
35          self.pattern_names_list = self.pattern_generator.
    pattern_names
36          self.pattern_filepath = Path(utility_fuctions.PathUtility.
    get_patterns_path())
37
38          self.camera = blend_camera #BlendCamera object
39
40          self.create_collections_and_viewlayers()
41          self.import_light_sources()
42          self.connect_collections_and_viewlayers()
43
44
45      def create_collections_and_viewlayers(self):
46          """
47          Creates collections and viewlayers, corresponding to the
    generated projected patterns
48          """
49
50          #Set parent collection to be scene master collection
51          parent_collection = bpy.context.scene.collection
52
53          view_layers = bpy.context.scene.view_layers
54
55          if not len(view_layers) == 1:
56              raise Exception('Blender file can not contain more than
    one view layer when running the pipeline\nBlender file currently
     contains {} view layers'.format(len(view_layers)))
57
58          bpy.context.view_layer.name = 'native_layer' #Rename native
    view layer
59
60          #Seperate native lighting from projector lighting
61          lights_in_scene = []
62          for light in bpy.data.lights:
63
64              if bpy.data.lights[light.name].users:
65                  lights_in_scene.append(light.name)
66
67          native_lights_list = list(set(lights_in_scene) - set(self.
    pattern_names_list))
```

```
68
69          #Create a collection to store native lighting
70          native_light_collection = bpy.data.collections.new(name='
    native_lights')
71          parent_collection.children.link(native_light_collection)
72
73          #Add native lighting to collection
74          for native_light_name in native_lights_list:
75
76              native_light = bpy.data.objects[native_light_name]
77
78              current_collection = native_light.users_collection
79
80              #Unlink object from previous collection and link to new
    collection
81              native_light_collection.objects.link(native_light)
82              current_collection[0].objects.unlink(native_light)
83
84          #Loop number of phase shift patterns in structured light
    algorithm
85          for pattern_name in self.pattern_names_list:
86
87              #Create viewlayers for both wave lengths at current
    shift
88              bpy.context.scene.view_layers.new(name=pattern_name)
89
90              #Create collections for both wave lengths at current
    shift
91              collection = bpy.data.collections.new(name="{}".format(
    pattern_name))
92
93              #Make collections children of master collection/scene
    collection
94              parent_collection.children.link(collection)
95
96
97      def connect_collections_and_viewlayers(self):
98          """
99          Connect collections with their associated view layers, to
    enable hiding and unhiding in other view layers.
100         """
101
102         for layer in bpy.context.scene.view_layers: #Loop layers
103             native_lighting_col = bpy.context.scene.view_layers[
    layer.name].layer_collection.children['native_lights']
104
105             if layer.name in self.pattern_names_list: #
106                 native_lighting_col.exclude = True
107
108             for collection in bpy.context.scene.view_layers[layer.
    name].layer_collection.children: #Loop collections in layer
```

```
109
110             if layer.name in self.pattern_names_list: #On a
    projector layer
111
112                 if collection.name == layer.name or collection.
    name not in self.pattern_names_list and collection.name != '
    native_lights': #Collection to show in layer
113                     collection.exclude = False
114
115                 else: #Collection not to show in layer
116                     collection.exclude = True
117
118             else: #Not a projector layer
119
120                 if collection.name in self.pattern_names_list: #
    Hide the projectors in the native layers
121                     collection.exclude = True
122
123
124
125     def import_light_sources(self):
126         """
127         Creating projector light sources in blender, and creating
    node tree for each pattern
128         """
129
130         print('################## importing light sources
    ##################')
131         for pattern_name in self.pattern_names_list:
132
133             coll = bpy.data.collections[pattern_name] #Store the
    associated blender collection as variable
134
135             bpy.data.lights.new(name=pattern_name, type='SPOT') #
    Create new light source
136
137             light = bpy.data.lights[pattern_name] #Store blender
    light as variable
138             light_obj = bpy.data.objects.new(name=pattern_name,
    object_data=light) #Store blender light object as variable
139             coll.objects.link(light_obj) #Link light object to
    assiciated collection
140
141             light.spot_blend = 0 #Edge blending of spotlight turned
    off
142             light.spot_size = np.pi #Set spot field of view to 180 (
    Larger than the image field of view)
143             light.shadow_soft_size = 0 #Makes edges of projected
    image sharp
144
145             light_obj.parent = bpy.data.objects[self.camera.name] #
```

```
    Set light to child of camera
146             light_obj.parent_type = 'OBJECT'
147
148             light_obj.location = self.cam2proj_loc #Set light
    location relative to camera
149
150             if len(self.cam2proj_rot) == 4:
151                 light_obj.rotation_mode = 'QUATERNION' #Set rotation
     mode to quaternion
152                 light_obj.rotation_quaternion = self.cam2proj_rot
153             else:
154                 light_obj.rotation_mode = 'XYZ' #Set rotation mode
    to euler xyz
155                 light_obj.rotation_euler = self.cam2proj_rot
156
157             light_obj.name = pattern_name #Rename light to be same
    as associated view layer and collection
158
159             #Set up the node tree for the projector
160             self.create_projector_node_tree(light=light)
161
162
163     def create_projector_node_tree(self, light):
164         '''
165         Creates the node tree for the light source, such that it
    projects the pattern as a projector
166         '''
167
168         light.use_nodes = True
169         node_tree = light.node_tree
170         nodes = node_tree.nodes
171         links = node_tree.links
172
173         #Store auto generated nodes for later use
174         emission_node = nodes[0]
175         light_out_node = nodes[1]
176
177         #Create and place texture coordinate node in node tree
178         texture_coord_node = nodes.new(type='ShaderNodeTexCoord')
179         texture_coord_node.location = (0, 0)
180         texture_coord_node.name = "text_coord_{}".format(light.name)
181
182         #Create and place separate XYZ node in node tree
183         separate_xyz_node = nodes.new(type='ShaderNodeSeparateXYZ')
184         separate_xyz_node.location = (200, -80)
185         separate_xyz_node.name = "separateXYZ_{}".format(light.name)
186
187         links.new(input=separate_xyz_node.inputs[0], output=
    texture_coord_node.outputs[1]) #Link texture node to seperate
    xyz node
188
```

```
189         #Create and place vector math divide node in node tree
190         divide_node = nodes.new(type='ShaderNodeVectorMath')
191         divide_node.location = (400, 0)
192         divide_node.operation = 'DIVIDE'
193         divide_node.name = "divide_{}".format(light.name)
194 
195         links.new(input=divide_node.inputs[0], output=
    texture_coord_node.outputs[1]) #Link texture node divide node
196         links.new(input=divide_node.inputs[1], output=
    separate_xyz_node.outputs[2]) #Link seperate xyz node to texture
     node
197 
198         #Mapping node
199         mapping_node = nodes.new(type='ShaderNodeMapping')
200         mapping_node.location = (600, 0)
201         mapping_node.name = "mapping_{}".format(light.name)
202 
203         x_scale = (self.focal_length/(self.sensor_size_horizontal))
    #Scale mapping node to focal length and vertical resolution
204         y_scale = (self.focal_length/(self.sensor_size_horizontal))
    *(self.pattern_shape[1]/self.pattern_shape[0]) #Scale mapping
    node to focal length and horizontal resolution
205 
206         links.new(input=mapping_node.inputs[0], output=divide_node.
    outputs[0])
207         mapping_node.inputs[1].default_value = [0.5,0.5,0] #Center
    image in sportlight
208         mapping_node.inputs[2].default_value = [0,0,np.pi] #Rotate
    image about z-axis of camera
209         mapping_node.inputs[3].default_value = [x_scale, y_scale, 1]
     #SCaling image mapping to fit focal length
210 
211         #Texture image node
212         texture_img_node = nodes.new(type='ShaderNodeTexImage')
213         texture_img_node.location = (800, 0)
214         texture_img_node.name = "text_img_{}".format(light.name)
215 
216         links.new(input=texture_img_node.inputs[0], output=
    mapping_node.outputs[0]) #Link mapping node to texture image
    node
217         texture_img_node.extension = 'CLIP' #Clip image, so that it
    doesnt repeat
218 
219         #Collect image from /utility/SL_patterns folder
220         pattern_filename = '{}x{}_{}.jpg'.format(self.pattern_shape
    [0],self.pattern_shape[1], light.name) #Filename of pattern
    stored in
221         pattern_img = bpy.data.images.load(filepath=str(Path.
    joinpath(self.pattern_filepath,Path(pattern_filename))))
222         texture_img_node.image = pattern_img #Set image to be
    projected from node
```

```
223
224         #Emission node
225         emission_node.location = (1200, 0)
226         emission_node.name = 'emission_{}'.format(light.name)
227
228         links.new(input=emission_node.inputs[0], output=
    texture_img_node.outputs[0]) #Link texture image node to
    emission node
229         emission_node.inputs[1].default_value = 5
230
231         #Light output node
232         light_out_node.location = (1400, 0)
233         light_out_node.name = "light_output_{}".format(light.name)
234
235     def get_projector_matrix(self):
236
237         focal_length = self.focal_length
238         sensor_width = self.sensor_size_horizontal
239
240         height_px = self.pattern_shape[0]
241         width_px = self.pattern_shape[1]
242
243         pixel_width = sensor_width/width_px
244
245         aspect_ratio = height_px/width_px
246
247         u_0 = height_px/2
248         v_0 = width_px/2
249
250         alpha_u = focal_length*(width_px/sensor_width)
251         alpha_v = focal_length*(height_px*aspect_ratio/(pixel_width*
    height_px))
252
253
254         M = np.array([[alpha_u,       0, u_0],
255                       [      0, alpha_v, v_0],
256                       [      0,       0,   1]])
257         return M
```

Listing A.6: Projector

A.7. Objects class

```
1 import types
2 import bpy
3 import os
4 import numpy as np
5 import random
6 import copy
7
```

```
from .camera_module import BlendCamera


class BlendObject:

    def __init__(self, object_info: dict, index: int, collection):

        #Collect input data
        self.filepath = object_info['filepath']
        self.scale = object_info['scale']
        self.mass = object_info['mass']
        self.collision_shape = object_info['collision_shape']
        self.index = index

        #Set unique pbject name and add to collection
        self.name = 'DataPipe_object.{:04d}'.format(self.index) #Set object name
        self.objects_collection = collection

        print("### OBJECT {} CREATED".format(self.name))

        #Import object to blender
        filename, blend_ob, blend_mat, blend_mesh = self.import_ob(filepath=self.filepath, index=self.index, scale=self.scale)
        self.filename = filename
        self.blend_ob = blend_ob
        self.blend_mat = blend_mat
        self.blend_mesh = blend_mesh

        self.dimensions = self.get_object_dimensions()


    def import_ob(self, filepath: str, index: int, scale: float):

        head, tail = os.path.split(filepath)
        filename = tail.replace('.obj', '') #Extract filename

        bpy.ops.object.select_all(action='DESELECT')
        bpy.ops.import_scene.obj(filepath=str(filepath))

        obj_in_file = len(bpy.context.selected_objects)
        if obj_in_file != 1: #Can only contain one object.
            raise Exception(".obj file can not contain more than one object, there are {} objects in file:\n{}".format(obj_in_file, filepath))

        if bpy.context.selected_objects[0].name != filename:
            filename = bpy.context.selected_objects[0].name

        obj = bpy.data.objects[filename]
        mat = obj.active_material
```

```
55          mesh = bpy.data.meshes[filename]
56
57          #Set names to current object name
58          obj.name = self.name
59          mat.name = self.name
60          mesh.name = self.name
61
62          #Remove from default collection and add to datapipe object
    collection
63          obj.users_collection[0].objects.unlink(obj)
64          self.objects_collection.objects.link(obj)
65
66          #Apply object scaling
67          obj.scale = (scale, scale, scale) #Set scale from user input
68          bpy.ops.object.transform_apply(location=False, scale=True,
    rotation=False) #Apply scale to object
69
70          #Set object physics properties
71          bpy.context.view_layer.objects.active = obj
72          bpy.ops.rigidbody.object_add(type='ACTIVE')
73          bpy.context.object.rigid_body.collision_shape = self.
    collision_shape
74          obj.rigid_body.mass = self.mass
75          obj.rigid_body.collision_margin = 0.001
76
77          obj.pass_index = index #Set pass index for masked image
78
79          return filename, obj, mat, mesh
80
81      def get_object_dimensions(self):
82
83          return self.blend_ob.dimensions
84
85      def delete_ob(self):
86          print("Blender object {} deleted".format(self.name))
87
88          bpy.data.objects.remove(self.blend_ob, do_unlink=True)
89          bpy.data.materials.remove(self.blend_mat, do_unlink=True)
90          bpy.data.meshes.remove(self.blend_mesh, do_unlink=True)
91
92      def place_ob(self, x, y, z):
93
94          self.blend_ob.location = x, y, z
95          self.blend_ob.rotation_mode = 'XYZ'
96          self.blend_ob.rotation_euler = (random.random()*2*np.pi,
    random.random()*2*np.pi, random.random()*2*np.pi)
```

Listing A.7: BlendObject

A.8. Objects manager class

```
1
2
3  class ObjectManager:
4
5      def __init__(self, config: dict):
6
7          self.objects_config = config['objects'] #Collect input dict
8
9          self.objects_info_list = self.objects_config['objects_list']
       #Object info list
10         self.objects_in_scene = []
11
12         self.objects_collection = self.create_objects_collection() #
       Create collection to store pipeline objects
13
14
15     def create_objects_collection(self):
16
17         objects_collection = bpy.data.collections.new('
       DataPipe_objects')
18         bpy.context.scene.collection.children.link(
       objects_collection)
19
20         return objects_collection
21
22     def import_objects(self):
23
24         self.delete_all_objects()
25
26         index = 1
27         print("\n### Importing objects ###\n")
28         for object_input in self.objects_info_list:
29
30             max_instances = object_input['max']
31             min_instances = object_input['min']
32
33             instances_in_scene = random.randint(a=min_instances, b=
       max_instances)
34
35             for instance in range(instances_in_scene):
36
37                 obj = BlendObject(object_info=object_input, index=
       index, collection=self.objects_collection)
38
39                 self.objects_in_scene.append(obj)
40
41                 index += 1
42         random.shuffle(self.objects_in_scene) #Randomizing the order
       of the objects
43
```

```
44      def delete_all_objects(self):
45          if len(self.objects_in_scene) != 0:
46
47              for obj in self.objects_in_scene:
48                  obj.delete_ob()
49
50                  del obj
51
52              self.objects_in_scene = []
53
54      def create_initial_positions(self, scene):
55
56          drop_zone_loc = scene.drop_zone_location
57          drop_zone_dim = scene.drop_zone_dimensions
58
59          z = drop_zone_loc[2] #Set initial z-coordinate to be at the
    midpoint of the dropzone height
60          delta_z = 0
61
62          max_x_coord = drop_zone_loc[0] + drop_zone_dim[0]/2 #Max x-
    value to place objects
63          min_x_coord = drop_zone_loc[0] - drop_zone_dim[0]/2 #Min x-
    value to place objects
64
65          max_y_coord = drop_zone_loc[1] + drop_zone_dim[1]/2 #Max y-
    value to place objects
66          min_y_coord = drop_zone_loc[1] - drop_zone_dim[1]/2 #Min y-
    value to place objects
67
68          for obj in self.objects_in_scene: #place objects random
69
70              max_dim = max(obj.dimensions) #The object's maximal
    dimension (either x, y, or z direction)
71
72              max_x_obj = max_x_coord - max_dim/2
73              min_x_obj = min_x_coord + max_dim/2
74
75              max_y_obj = max_y_coord - max_dim/2
76              min_y_obj = min_y_coord + max_dim/2
77
78              x = random.random()*(max_x_obj-min_x_obj) + min_x_obj
79              y = random.random()*(max_y_obj-min_y_obj) + min_y_obj
80
81              z += delta_z + max_dim/2
82
83              obj.place_ob(x=x, y=y, z=z)
84
85              delta_z = max_dim/2
86
87      def get_objects_information_dict(self, camera: BlendCamera):
88
```

```
89          obj_output_list = []
90
91          for obj in self.objects_in_scene:
92
93              dict = {}
94
95              wrld2cam_transform = np.asarray(camera.blend_cam_obj.
    matrix_world)
96
97              wrld2obj_transform = np.asarray(obj.blend_ob.
    matrix_world)
98
99              cam2obj_pose = np.matmul(np.linalg.inv(
    wrld2cam_transform), wrld2obj_transform)
100
101             dict = {'name': obj.name,
102                     'filename': obj.filename,
103                     'mask_index': obj.index,
104                     'cam2obj_pose': cam2obj_pose}
105
106             obj_output_list.append(dict)
107
108
109         return obj_output_list
```

Listing A.8: ObjectManager

A.9. Simulation class

```
1  import bpy
2  import time
3
4  from.objects_module import ObjectManager
5
6
7  class Simulation:
8
9      def __init__(self, sim_end: int):
10
11         self.sim_end = sim_end
12         self.sim_start = 1
13
14         bpy.context.scene.rigidbody_world.point_cache.frame_start =
    self.sim_start
15         bpy.context.scene.frame_set(frame=self.sim_start)
16
17         bpy.context.scene.rigidbody_world.point_cache.frame_end =
    self.sim_end
18         bpy.context.scene.frame_end = self.sim_end
19
```

```
20
21      def run_loop(self):
22          sim_end = self.sim_end
23          print("Simulation initiated")
24          start = time.time()
25          temp_prev = time.time()
26          for frame_num in range(sim_end-2):
27              bpy.context.scene.frame_set(bpy.context.scene.
    frame_current + 1) #Loop frames to 400
28              temp = time.time()
29              print("\n--- FRAME {} at time {}s".format(frame_num+1,
    temp-temp_prev))
30              temp_prev = temp
31          end = time.time()
32          print("Simulation done in {} seconds".format(end-start))
33
34
35      def apply_simulated_transforms(self, object_manager:
    ObjectManager):
36
37          for obj in object_manager.objects_in_scene:
38
39              bpy.context.view_layer.objects.active = obj.blend_ob #
    Set obj to active
40              bpy.ops.object.visual_transform_apply() #Apply transform
```

Listing A.9: Simulation

A.10. Render class

```
1  import bpy
2  from pathlib import Path
3  import time
4
5  from .scene_module import BlendScene
6
7
8  class Renderer:
9
10     def __init__(self, camera: object):
11
12         self.camera = camera
13
14         self.view_layers_list = camera.projector.pattern_names_list
15
16         self.output_nodes_list = self.create_render_node_tree()
17
18         self.render_path = ''
19
20
```

```
21      def create_render_node_tree(self):
22
23          print("\n\n#####################\nCREATING NODE TREE\n
    ####################\n\n")
24          x_coord = 0
25          y_coord = 0
26
27          delta_y = -110
28          delta_x = 300
29
30          bpy.context.scene.use_nodes = True
31          node_tree = bpy.context.scene.node_tree
32          node_tree.nodes.clear() #Clear auto created nodes
33          links = node_tree.links
34
35
36          out_node_list = []
37
38          native_layer = bpy.context.scene.view_layers['native_layer']
39
40          #Set viewlayer render options
41          native_layer.use_pass_z = True
42          native_layer.use_pass_normal = True
43          native_layer.use_pass_object_index = True
44
45          render_node = node_tree.nodes.new(type="
    CompositorNodeRLayers") #Add a render layer node
46          render_node.layer = native_layer.name
47          render_node.name = "render_node_native_layer"
48          render_node.location = x_coord, y_coord
49
50          x_coord += delta_x
51
52          #RGB image node
53          rgb_out_node = out_node = node_tree.nodes.new(type="
    CompositorNodeOutputFile")
54          rgb_out_node.name = 'rgb_output_node'
55          rgb_out_node.format.file_format = 'PNG'
56          rgb_out_node.location = x_coord, y_coord
57          rgb_out_node.inputs[0].name = 'rgb_image'
58
59          out_node_list.append(rgb_out_node) #Add rgb output node to
    out_node_list
60
61          rgb_link = links.new(input=rgb_out_node.inputs[0], output=
    render_node.outputs[0]) #Link to render layer node
62
63          y_coord += delta_y
64
65          #Depth image node
66          depth_out_node = node_tree.nodes.new(type="
```

```
    CompositorNodeOutputFile")
67          depth_out_node.name = 'depth_output_node'
68          depth_out_node.format.file_format = 'OPEN_EXR'
69          depth_out_node.location = x_coord, y_coord
70          depth_out_node.inputs[0].name = 'depth_image'
71
72          out_node_list.append(depth_out_node) #Add depth output node
    to out_node_list
73
74          depth_link = links.new(input=depth_out_node.inputs[0],
    output=render_node.outputs[2]) #Link to render layer node
75
76          y_coord += delta_y
77
78          #Normals image node
79          normals_out_node = node_tree.nodes.new(type="
    CompositorNodeOutputFile")
80          normals_out_node.name = 'normals_output_node'
81          normals_out_node.format.file_format = 'PNG'
82          normals_out_node.location = x_coord, y_coord
83          normals_out_node.inputs[0].name = 'normals_image'
84
85          out_node_list.append(normals_out_node) #Add normals output
    node to out_node_list
86
87          normals_link = links.new(input=normals_out_node.inputs[0],
    output=render_node.outputs[3]) #Link to render layer node
88
89          y_coord += delta_y
90
91          #Masked image node
92          mask_out_node = node_tree.nodes.new(type="
    CompositorNodeOutputFile")
93          mask_out_node.name = 'mask_output_node'
94          mask_out_node.format.file_format = 'OPEN_EXR'
95          mask_out_node.location = x_coord, y_coord
96          mask_out_node.inputs[0].name = 'masked_image'
97
98          out_node_list.append(mask_out_node) #Add mask output node to
     out_node_list
99
100         mask_link = links.new(input=mask_out_node.inputs[0], output=
    render_node.outputs[4]) #Link to render layer node
101
102         x_coord = -500
103         y_coord = 0
104         delta_y = -400
105
106         if self.camera.is_structured_light: #Create render nodes for
     structured light view layers
107
```

```
108             for layer_name in self.view_layers_list:
109 
110                 render_node = node_tree.nodes.new(type="
    CompositorNodeRLayers") #Add a render layer node
111                 render_node.layer = layer_name
112                 render_node.name = "render_node_{}".format(
    layer_name)
113                 render_node.location = x_coord, y_coord
114 
115                 out_node = node_tree.nodes.new(type="
    CompositorNodeOutputFile")
116                 out_node.name = "output_node_{}".format(layer_name)
117                 out_node.location = x_coord + delta_x, y_coord
118                 out_node.format.file_format = 'PNG'
119                 out_node.inputs[0].name = layer_name
120 
121                 out_node_list.append(out_node)
122 
123                 out_link = links.new(input=out_node.inputs[0],
    output=render_node.outputs[0])
124 
125                 y_coord += delta_y
126 
127         return out_node_list
128 
129     def set_output_paths(self, scene: BlendScene, render_num):
130 
131         scene_path = scene.output_path
132 
133         render_dir_name = 'render.{:04d}'.format(render_num)
134         render_path = Path.joinpath(scene_path, Path(render_dir_name
    ))
135 
136         render_path.mkdir()
137         render_path = str(render_path)
138 
139         self.render_path = render_path
140 
141         for out_node in self.output_nodes_list:
142 
143             path_string = "{}/{}".format(render_path, out_node.
    inputs[0].name)
144             path = Path(path_string)
145             out_node.base_path =  str(path) #Set render path for
    nodes
146 
147 
148     def render_results(self):
149         render_start = time.time()
150         bpy.ops.render.render(use_viewport=True)
151         render_end = time.time()
```

```
152         print("Rendertime: {:.4f}".format(render_end-render_start))
```

Listing A.10: Renderer

A.11. Config class

```
import pickle
from pathlib import Path
import os
from . import utility_fuctions

import bpy

class input_storage:

    config_dict = {
                        'camera': {
                            'wrld2cam_pose_list':[],
                            'focal_length': 50,
                            'sensor_width': 36,
                            'resolution': [480, 720],
                            'is_structured_light': False

                        },
                        'projector': {
                            'proj2cam_pose': {},
                            'focal_length': 50,
                            'sensor_width': 36,
                            'resolution': [480, 720]
                        },
                        'scene': {
                            'num_renders': 1,
                            'max_renders_per_scene': 1,
                            'min_renders_per_scene': 1,
                            'drop_zone_loc': [0, 0, 2],
                            'drop_zone_scale': [0.5, 0.5, 0.5]

                        },
                        'objects': {
                            'objects_list':[]

                        },
                        'output': {
                            'path': ''
                        }

    }

    @classmethod
    def reset_config_dict(cls):
```

```
        """
        Resets configuration dict for the full pipeline.
        """

        cls.config_dict = {
                            'camera': {
                                'wrld2cam_pose_list':[],
                                'focal_length': 50,
                                'sensor_width': 36,
                                'resolution': [480, 720],
                                'is_structured_light': False

                            },
                            'projector': {
                                'proj2cam_pose': {},
                                'focal_length': 50,
                                'sensor_width': 36,
                                'resolution': [480, 720]
                            },
                            'scene': {
                                'num_renders': 1,
                                'max_renders_per_scene': 1,
                                'min_renders_per_scene': 1,
                                'drop_zone_loc': [0, 0, 2],
                                'drop_zone_scale': [0.5, 0.5, 0.5]

                            },
                            'objects': {
                                'objects_list':[]

                            },
                            'output': {
                                'path': ''
                            }

            }

    @classmethod
    def write_to_config_dict(cls, context):

        config = cls.config_dict

        scene_config = config['scene']
        objects_config = config['objects']
        camera_config = config['camera']
        projector_config = config['projector']
        output_config = config['output']

        #Scene inputs
        scene_config['num_renders'] = context.scene.num_renders
        scene_config['max_renders_per_scene'] = context.scene.
```

```
    max_renders_per_scene
96      scene_config['min_renders_per_scene'] = context.scene.
    min_renders_per_scene
97
98      #Drop zone inputs
99      drop_zone = bpy.data.objects['drop_zone'] #Get blender
    object
100
101     bpy.ops.object.select_all(action='DESELECT') #Deselect all
    object in blender scene
102
103     if drop_zone.select_get() is False:
104         drop_zone.select_set(True) #Select dropzone object
105
106     bpy.context.view_layer.objects.active = drop_zone #Set to
    active object
107
108     bpy.ops.object.select_all(action='DESELECT') #Deselect all
    object after
109
110     #Store object data in config
111     scene_config['drop_zone_loc'] = list(bpy.data.objects['
    drop_zone'].location)
112     scene_config['drop_zone_scale'] = list(bpy.data.objects['
    drop_zone'].scale)
113
114     #Camera intrinsics
115     camera_config['focal_length'] = context.scene.
    camera_focal_length
116     camera_config['sensor_width'] = context.scene.
    camera_sensor_width
117     camera_config['resolution'] = [context.scene.
    camera_resolution_height, context.scene.camera_resolution_width]
118     camera_config['is_structured_light'] = context.scene.
    is_structured_light
119
120     #Projector intrinsics
121     projector_config['focal_length'] = context.scene.
    projector_focal_length
122     projector_config['sensor_width'] = context.scene.
    projector_sensor_width
123     projector_config['resolution'] = [context.scene.
    projector_resolution_height, context.scene.
    projector_resolution_width]
124
125     #Projector extrinsics
126     loc = list(context.scene.projector_loc_vec)
127     if context.scene.projector_rot_enum == 'quat':
128         rot = list(context.scene.projector_rot_quat)
129     else:
130         rot = list(context.scene.projector_rot_xyz)
```

```
131
132         projector_config['proj2cam_pose'] = {'rotation': rot, '
    location': loc}
133
134         #Output
135         output_config['path'] = utility_fuctions.PathUtility.
    get_pipeline_run_output_path(context.scene.pipeline_output_path)
136
137
138     @classmethod
139     def set_input_panel_vars_from_dict(cls, context):
140         """
141         Set variables in Blender GUI to be the same as loaded file
142         """
143
144         config = cls.config_dict
145         scene_config = config['scene']
146         objects_config = config['objects']
147         camera_config = config['camera']
148         projector_config = config['projector']
149         output_config = config['output']
150
151         #Set scene variables
152         context.scene.num_renders = scene_config['num_renders']
153         context.scene.max_renders_per_scene = scene_config['
    max_renders_per_scene']
154         context.scene.min_renders_per_scene = scene_config['
    min_renders_per_scene']
155
156         #Import and move dropzone in blender
157         if 'drop_zone' in bpy.data.objects.keys():
158
159             bpy.data.objects['drop_zone'].location = scene_config['
    drop_zone_loc']
160             bpy.data.objects['drop_zone'].scale = scene_config['
    drop_zone_scale']
161
162         else:
163
164             bpy.ops.mesh.primitive_cube_add()
165             drop_zone = bpy.context.active_object
166             mesh_name = drop_zone.name
167             drop_zone.name = 'drop_zone'
168             bpy.data.meshes[mesh_name].name = 'drop_zone_mesh'
169             drop_zone.location = scene_config['drop_zone_loc']
170             drop_zone.scale = scene_config['drop_zone_scale']
171
172             bpy.ops.object.select_all(action='DESELECT')
173             if drop_zone.select_get() is False:
174                 drop_zone.select_set(True)
175             bpy.context.view_layer.objects.active = drop_zone
```

```
176             bpy.ops.object.select_all(action='DESELECT')
177
178
179         #Set camera variables
180         context.scene.camera_focal_length = camera_config['
    focal_length']
181         context.scene.camera_sensor_width = camera_config['
    sensor_width']
182         camera_resolution = camera_config['resolution']
183         context.scene.camera_resolution_height = camera_resolution
    [0]
184         context.scene.camera_resolution_width = camera_resolution[1]
185         context.scene.is_structured_light = camera_config['
    is_structured_light']
186
187         #Set projector variables
188         context.scene.projector_focal_length = projector_config['
    focal_length']
189         context.scene.projector_sensor_width = projector_config['
    sensor_width']
190         projector_resolution = projector_config['resolution']
191         context.scene.camera_resolution_height =
    projector_resolution[0]
192         context.scene.camera_resolution_width = projector_resolution
    [1]
193
194         context.scene.projector_loc_vec = projector_config['
    proj2cam_pose']['location']
195         rot = projector_config['proj2cam_pose']['rotation']
196         if len(rot) == 3:
197             context.scene.projector_rot_enum = 'xyz'
198             context.scene.projector_rot_xyz = rot
199         else:
200             context.scene.projector_rot_enum = 'quat'
201             context.scene.projector_rot_quat = rot
202
203
204
205     @classmethod
206     def write_to_pickle_file(cls, dir_path: str):
207         dir_path = Path(bpy.path.abspath(dir_path)).resolve()
208         filename = Path('DataPipe_input.pickle')
209
210         path = Path.joinpath(dir_path, filename)
211
212         if utility_fuctions.file_exists(path):
213
214             exist = True
215             index = 1
216
217             while exist:
```

```
218
219                 filename = Path('DataPipe_input.{:04d}.pickle'.
    format(index))
220                 path = Path.joinpath(dir_path, filename)
221                 exist = utility_fuctions.file_exists(path)
222                 index += 1
223
224         pickle_file = open(path, "wb")
225         pickle.dump(cls.config_dict, pickle_file)
226         pickle_file.close()
227
228     @classmethod
229     def input_from_file(cls, file_path: str):
230
231         file_path = Path(bpy.path.abspath(file_path)).resolve()
232
233         if file_path.suffix == '.pickle':
234             pickle_file = open(file_path, "rb")
235             cls.config_dict = pickle.load(pickle_file)
236             print("-- Input loaded from file:\n{}".format(cls.
    config_dict))
237         else:
238             print("-- Filetype must be .pickle")
```

Listing A.11: input$_{c}onfig$

A.12. Patterns class

```
1  import os
2  from pathlib import Path
3  import cv2
4  import numpy as np
5  from numpy import pi
6  import copy
7
8  from . import utility_fuctions
9
10
11
12 class PatternGenerator:
13     """
14     Class for generating structured light patterns.
15     """
16
17     shifts = 3
18
19     def __init__(self, resolution: list):
20
21         self.resolution = np.asarray(resolution)
22
```

```
        self.num_phase_shifts = 3
        self.periods = [8, 7]

        self.patterns_list = []
        self.generate_fringe_pattern(resolution=self.resolution, periods=self.periods[0])
        self.generate_fringe_pattern(resolution=self.resolution, periods=self.periods[1])

        self.store_patterns()

        self.pattern_names = self.get_pattern_names_list()


    def get_pattern_names_list(self):
        """
        Generate a list of pattern names to be used in creation of viewlayers
        """

        names = []

        for pattern in self.patterns_list:
            names.append(pattern['name'])
        return names


    def generate_fringe_pattern(self, resolution: list, periods: int):
        """
        Generates three monochrome fringe patterns for structured light scanner

        :param width: image width
        :type width: int
        :param height: image height
        :type height: int
        :param periods: number of full periods along image width
        :type periods: int.
        """
        print("### Generating patterns ###")

        resolution = np.asarray(resolution)
        print("Resolution: {}".format(resolution))
        height, width = resolution[0], resolution[1]

        periods = periods
        shifts = self.num_phase_shifts

        delta_x= 2*pi*periods/(width)
        x = np.arange(0,2*pi*periods,delta_x)
```

```
69
70          phi = 2*pi/shifts
71
72          canvas = np.ones((height,width,shifts))
73          waves = np.transpose(255 * (0.5 + 0.5 * np.cos(np.array([x,
    x + phi, x - phi]))))
74
75          patterns = canvas*waves
76
77          for shift in range(shifts):
78
79              name = 'p{}s{}'.format(periods, shift+1)
80
81              pattern = copy.copy(patterns[:,:, shift])
82
83              pattern_dict = {'name': name,
84                              'pattern':pattern}
85
86              self.patterns_list.append(pattern_dict)
87
88
89      def store_patterns(self):
90          """
91          Stores generated pattern images in utility folder
92          """
93
94          patterns_folder_path = Path(utility_fuctions.PathUtility.
    get_patterns_path())
95
96          for pattern in self.patterns_list:
97
98              height, width = pattern['pattern'].shape
99
100             filename = "{}x{}_{}.jpg".format(height, width, pattern[
    'name'])
101
102             Path(filename)
103             save_path = str(Path.joinpath(patterns_folder_path,
    filename))
104
105             pattern['filename'] = save_path
106
107             if not os.path.exists(save_path): #Check if the pattern
    already exist
108                 cv2.imwrite(save_path, pattern['pattern'])
```

Listing A.12: PatternGenerator

A.13. Algorithm class

import numpy as np from pathlib import Path import copy import cv2

from .render$_{m}oduleimportRendererfrom.pattern_{g}eneratorimportPatternGeneratorfrom.c$

class Algorithm:

def $_{i}nit_{(self,renderer:Renderer,pattern_{n}ames:list,pattern_{g}enerator:PatternGenerator,camera:BlendCamera)}$:

self.render$_{p}ath = renderer.render_{p}ath$

self.file$_{p}aths = self.get_{r}ender_{o}ut_{s}ubdirs(pattern_{n}ames = pattern_{n}ames)$

self.images = self.load$_{i}mages(file_{p}aths = self.file_{p}aths)$

self.periods = pattern$_{g}enerator.periods$

self.phase1$_{i}mg = self.phase_{d}etection(images = self.images[:,:,:3])self.phase2_{i}mg =$
$self.phase_{d}etection(images = self.images[:,:,3:])$

self.abs$_{p}hase_{i}mg = self.absolute_{p}hase(self.phase1_{i}mg, self.phase2_{i}mg)$

self.camera$_{m}atrix = np.matmul(camera.K, camera.camera_{e}xtrinsic)self.projector_{m}atrix =$
$np.matmul(camera.projector_{m}atrix, camera.projector_{e}xtrinsic)$

self.depth$_{i}mg, image = self.triangulate_{d}epth(camera.projector.pattern_{s}hape[1], self.abs_{p}ha$

def get$_{r}ender_{o}ut_{s}ubdirs(self, pattern_{n}ames : list) :''' Collectstherenderedoutputimagepaths_$

render$_{d}ir = Path(self.render_{p}ath)$

file$_{p}aths = []$

for pattern$_{n}ameinpattern_{n}ames : sub_{d}ir = Path(pattern_{n}ame)img_{d}ir = Path.joinpath(ren$

for file$_{p}athinimg_{d}ir.iterdir() :$

if file$_{p}ath.suffix ==' .png' :$

file$_{p}aths.append(str(file_{p}ath))break$

return file$_{p}aths$

def load$_{i}mages(self, file_{p}aths) :''' Loadallrenderedstructuredlightimagesintomatrix'''print$

supported$_{f}iletypes = ['.png','.jpg']$

num$_{i}mages = len(file_{p}aths)$

threshold = (25**2)*num$_{i}mages$

count = 0

for image$_path in file_paths$:

image$_path = Path(image_path)$

if image$_path.suffix not in supported_filetypes : raiseException("Filetype not valid, use".form$

img = cv2.imread(str(image$_path$))$img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)$

(height,width) = img$_gray.shape$

if count == 0: out = np.empty((height, width, num$_images$))

out[:,:,count] = img$_gray$

count += 1

Remove unlit noisy pixels error$_img = copy.copy(out)error_img = np.sum(error_img*$
$*2, axis = 2)filter_img = np.ones_like(out[:, :, 0])filter_img[np.where(error_img <$
$threshold)] = None$

for i in range(len(out[0,0,:])): out[:,:,i] = np.multiply(out[:,:,i], filter$_img$)

return out

def remove$_unlit_pixels(self, images, threshold) : error_img = copy.copy(images)error_img =$
$np.sum(error_img**2, axis = 2)filter_img = np.ones_like(images[:, :, 0])filter_img[np.where(e$
$threshold)] = None$

for i in range(len(images[0,0,:])): images[:,:,i] = np.multiply(images[:,:,i], filter$_img$)

return images

def phase$_detection(self, images) :''' Extracts the phase iamge for every shifted fringe image pat$
$images.shape$

r = dim[0] c = dim[1]

phase = np.empty((r,c)) num$_phase_steps = dim$[2]

ph = 2*np.pi*np.arange(0,num$_phase_steps)/num_phase_steps Plasser fase forskyningen i en vek$

sinph = np.sin(ph) Faseforskyvningen cosph = np.cos(ph) Faseforskyvningen print(sinph)
print(cosph)

image$_vector = np.array(images).reshape(r*c, num_phase_steps) Omformbildet til en vektor pri$

numerator = np.matmul(image$_vector, sinph) Multiplisermed fase forskyvningen denominato$
$np.matmul(image_vector, cosph) Multiplisermed fase forskyvningen$

print(numerator.shape)

phase = -np.arctan2(numerator, denominator) Finn vinkelen i sin og cos

phase = phase.reshape(r,c) Form bildet tilbake til originalt format phase = np.mod(phase,2*np
Map verdiene til 0 til 2pi

return phase

def absolute$_p$hase($self, phase1, phase2$) : """Computestheabsolutephaseusingtwowavehetero
$abs_phasereturntype$: $numpy.ndarray$"""

phase$_e q = np.mod(phase1 - phase2, np.pi * 2)$

l$_1 = phase1.shape[1]/self.periods[0]Wavelengthforpattern1l_2 = phase1.shape[1]/self.perio$
$l_2 * l_1/(l_2 - l_1)$

k = np.round(((l$_e q/l_1) * phase_e q - phase1)/(np.pi * 2)$)

temp = phase1 + k * 2*np.pi

abs$_p hase = temp/self.periods[0]$

return abs$_p hase$

def triangulate$_d epth(self, projector_r es_w idth, abs_p hase, camera_m atrix, projector_m atrix)$:

P$_c = camera_m atrix P_p = projector_m atrix$

depth$_i mg = np.empty_l ike(abs_p hase) count = 0$

for u$_c in range(len(depth_i mg[:, 0]))$:

for v$_c in range(len(depth_i mg[0, :]))$:

phase = abs$_p hase[u_c, v_c]$

if phase is not None or phase is not 'nan':

v$_p = projector_r es_w idth * (phase/(np.pi * 2))$

M = np.array([[P$_c[0, 0] - P_c[2, 0] * u_c, P_c[0, 1] - P_c[2, 1] * u_c, P_c[0, 2] - P_c[2, 2] *$
$u_c], [P_c[1, 0] - P_c[2, 0] * v_c, P_c[1, 1] - P_c[2, 1] * v_c, P_c[1, 2] - P_c[2, 2] * v_c], [P_p[1, 0] -$
$P_p[2, 0] * v_p, P_p[1, 1] - P_p[2, 1] * v_p, P_p[1, 2] - P_p[2, 2] * v_p]$])

vec = np.array([[P$_c[0, 3] - P_c[2, 3] * u_c], [P_c[1, 3] - P_c[2, 3] * v_c], [P_p[1, 3] - P_p[2, 3] * v_p]$])

world$_c oordinates = np.matmul(np.linalg.inv(M), vec)$

depth$_i mg[u_c, v_c] = world_c oordinates[2] count+ = 1$

return depth$_i mg$

REMOVE def temp$_s ave_i mage_t o_r ender_p ath(self, image) : path = Path.joinpath(Path(self.r$
$1 while path.exists()$:

name = Path('image.png'.format(count)) path = Path.joinpath(Path(self.render$_p ath), name$)

count += 1

path = str(path) cv2.imwrite(path, image)

Listing A.13: Algorithm

A.14. Utility functions

```
from genericpath import exists
import types
import bpy
import subprocess
import numpy as np
from scipy.spatial.transform import Rotation
import os
from pathlib import Path

from .scene_module import BlendScene
from .config_module import input_storage

class PathUtility:

    @staticmethod
    def get_addon_path():
        """
        Returns path to addon folder.
        """
        resource_path = Path(bpy.utils.resource_path(type='USER'))
        addon_sub_path = Path("scripts/addons/DataPipe")
        addon_path = Path.joinpath(resource_path, addon_sub_path)
        print("addon_path:\n{}".format(addon_path))
        return str(addon_path)

    @staticmethod
    def get_patterns_path():
        """
        Returns path to patterns folder.
        """
        resource_path = Path(bpy.utils.resource_path(type='USER'))
        pattern_sub_path = Path("scripts/addons/DataPipe/utility/
    SL_patterns")
        pattern_path = Path.joinpath(resource_path, pattern_sub_path
    )
        print("$$$$$$$\nResource path:\n{}\n--> Type: {}\n\nPattern
    sub path:\n{}\n--> Type: {}\n\nTotal path:\n{}\n--> Type: {}\
    n$$$$$$$".format(resource_path, type(resource_path),
    pattern_sub_path, type(pattern_sub_path), pattern_path, type(
    pattern_path)))
        return str(pattern_path)
```

```
36
37      @staticmethod
38      def get_pipeline_run_output_path(path: Path):
39          path = Path(bpy.path.abspath(path)).resolve()
40          dir_name = Path("DataPipe_run")
41          out_path = Path.joinpath(path,dir_name)
42          print("Try 1 at path_\n{}".format(str(out_path)))
43          if not os.path.exists(out_path):
44              return str(out_path)
45
46          exists = True
47          index = 1
48
49          while exists:
50              dir_name = 'DataPipe_run.{:04d}'.format(index)
51              out_path = Path.joinpath(path, dir_name)
52              print("Try {} at path:\n{}".format(index+1, str(out_path
    )))
53              if not Path.exists(out_path):
54                  exists = False
55                  return str(out_path)
56              index += 1
57
58
59
60
61  class PackageControll:
62      """
63      Class for installing python dependencies for the pipeline addonq
64      """
65
66      package_list = ["opencv-python", "scipy"]
67
68      @classmethod
69      def installDependencies(cls):
70          """
71          installing package dependencies to Blenders bundled python
72          """
73
74          #Path to python executable
75          py_exec = str(bpy.app.binary_path_python)
76
77          #Ensure that pip is installed
78          subprocess.call([py_exec, '-m', 'ensurepip', '--user'])
79
80          #Install latest version of pip
81          subprocess.call([py_exec, '-m', 'pip', 'install', '--upgrade
    ', 'pip'])
82
83          #Loop package list to install all of them
84          for package_name in cls.package_list:
```

```
            subprocess.check_call([py_exec, '-m', 'pip', 'install','{}'.format(package_name)])


def quat2rot(quat):
    """
    Converts blender quaternion representation to rotation matrix

    :param: quat
    :type quat: array
    """
    #Converts from blender quaternion representation
    q = [quat[1], quat[2], quat[3], quat[0]]

    quat = Rotation.from_quat(q) #Stores as a rotation
    R = quat.as_matrix() #Rotation matrix
    return R

def xyz2rot(xyz):

    xyz_rot = Rotation.from_euler(seq='XYZ',angles=xyz)
    R = xyz_rot.as_matrix()
    q = xyz_rot.as_quat()
    quat = np.array([q[3], q[0], q[1], q[2]])
    return R, quat

def rot2quat(rot):
    """
    Converts a rotation matrix to  blender quaternion representation

    :param: rot
    :type rot: array
    """
    R = Rotation.from_matrix(rot) #Stores as a rotation
    q = R.as_quat() #Quaternions

    quat = [q[3], q[0], q[1], q[2]] #Saves quaternion in Blender representation
    return quat


def pose_to_tranformation_matrix(rotation, location):
    """
    Changes Blenders quaternion or Euler rotational representation and location
    to transformation matrix representation
    """

    if len(rotation) == 4:
```

```
        R = quat2rot(quat=rotation)
    else:
        R = xyz2rot(xyz= rotation)

    t = location

    T = np.array([[R[0,0], R[0,1], R[0,2], t[0]],
                  [R[1,0], R[1,1], R[1,2], t[1]],
                  [R[2,0], R[2,1], R[2,2], t[2]],
                  [0,      0,      0,      1   ]])
    return T, R, t


def transformation_matrix_to_quat_and_translation(matrix):
    """
    Converts pose from transformation matrix format to quaternions
    and translation

    return: [quaternions] and [x,y,z]
    """
    quat = rot2quat(rot=matrix[0:3,0:3])
    trans = [matrix[0,3], matrix[1,3], matrix[2,3]]
    return quat, trans


def transform_inverse(matrix):
    R = matrix[0:3, 0:3]
    t = matrix[0:3, 3]

    T = np.zeros_like(matrix)
    T[0:3, 0:3] = np.transpose(R)
    T[0:3, 3] = - np.transpose(R)@t
    T[3,3] = 1
    return T


def cam2obj_transform(blender_object, cam_pos_matrix):
    obj_translation = blender_object.location #Get object location
    vector
    obj_quaternions = blender_object.rotation_quaternion #Get object
     rotation on quaternion
    obj_trans_quaternions = rot2quat(quat2rot(obj_quaternions))

    cam_translation = cam_pos_matrix[0:3,3] #Get camera location
    vector
    cam_quaternions = rot2quat(cam_pos_matrix[0:3,0:3]) #Get camera
    rotation on quaternion

    T_so, R_so, t_so = pose_to_tranformation_matrix(obj_quaternions,
     obj_translation) #Object transformation matrix in world
    coordinates
```

```
177     T_sc, R_sc, t_sc = pose_to_tranformation_matrix(cam_quaternions,
         cam_translation) #Camera transformation matrix in world
        coordinates
178
179     T_cs = transform_inverse(T_sc) #World coordinate system in
        camera coordinate system
180
181     T_co = np.matmul(T_cs,T_so) #Object transform from camera
        coordinate system
182     R_co = T_co[0:3,0:3] #Rotation
183     t_co = T_co[0:3,3] #Translation
184
185     return T_co, R_co, t_co
186
187 def file_exists(file_path: str):
188
189     return os.path.exists(file_path)
190
191
192 def initialize_pipeline_environment():
193     """
194     Setting blender variables to the required specifications for the
         pipeline.
195     -Rendering engine is set to cycles and gpu-compute.
196     -Scene units are set to metric.
197     -Native object
198     """
199     print("###### INITIALIZING PIPELINE ######")
200
201     bpy.context.scene.render.engine = 'CYCLES'
202     bpy.context.scene.cycles.device = 'GPU'
203
204     bpy.context.scene.unit_settings.system = 'METRIC'
205     bpy.context.scene.unit_settings.length_unit = 'METERS'
206     bpy.context.scene.unit_settings.system_rotation = 'RADIANS'
207     bpy.context.scene.unit_settings.mass_unit = 'KILOGRAMS'
208     bpy.context.scene.world.color = (0,0,0)
209
210     BlendScene.reset_scene_number()
211     input_storage.reset_config_dict()
212
213
214     #Make all existing meshes be rigid bodies.
215     for obj in bpy.data.objects:
216
217         obj.pass_index = 0
218
219         bpy.context.view_layer.objects.active = obj
220
221         if bpy.context.object.type == 'MESH':
222             bpy.ops.rigidbody.object_add(type='PASSIVE')
```

```
223             bpy.context.object.rigid_body.collision_shape = 'MESH'
224
225
226             obj.rigid_body.collision_margin = 0.001
```

Listing A.14: utility functions

William Eikrem

6D Synthetic Data Generation Pipeline with Digital Representation of Structured Light Sensor

NTNU
Norwegian University of
Science and Technology