Semi-transparent object recognition

Rishabh Ramteke
Indian Institute of Technology Bombay
Mumbai, India
rishabhramtekegsr@gmail.com

Markus Vincze
Automation and Control Institute
TU Wien, Vienna, Austria
Vincze@acin.tuwien.ac.at

***Abstract*—The internship project involves the topic of recognising semi-transparent objects where we work on methods to model these objects, study methods that detect and recognise them, and finally will contribute to methods that estimate 6DOF pose of objects for subsequent robotic grasping. The project will involve using self attention for instance segmentation of semi-transparent objects**
We worked towards learning and detecting transparent objects as needed for the project with plastic canisters and bottles. In particular we worked on the simulation of transparent objects and the rendering of realistic scenes with these object such that we can create data for learning neural networks.

***Index Terms*—Region Based Convolutional Neural Networks, Instance segmentation, self attention**

I. Introduction

The main goal of this work was to create synthetic transparent data so that it could be used as a training dataset for various neural networks. Currently, there isn't a lot of data of transparent objects like bottles, glass, etc. available and creating it from real scenes would be too much manual work as the neural network requires millions of images for each class of object in order to attain good recognizable rates. Hence, the reason why we wanted to create synthetic dataset. We use Blender software to setup scenes and then render it to obtain various outputs like depth, RGB image and image segmentation. Blender is a open source 3D creation suite. It supports the entirety of the 3D pipeline—modeling, rigging, animation, simulation, rendering, compositing and motion tracking. The particular neural network we will look at is the mask RCNN. Since, we will be using a network which performs instance segmentation, we need to obtain this type of output from Blender so that we can calculate the accuracy of our model. We also need the depth images as training dataset for pose estimation or detection tasks. From blender we can directly output Depth but instead we will use Block-matching algorithm to do this. The reason we do this is because the depth that we get from Blender is the perfect depth and in reality, there are some background effects like glare, shadows, etc which causes to have a very different depth image. And if we use the perfect depth images for training our model, then we will get unrealistic results.

Before dicussing about Mask RCNN, we need to look at the family of RCNNs.

Region Based Convolutional Neural Networks (R-CNN) is a machine learning model for computer vision and specifically object detection. Its goal was to take an input image and produce a set of bounding boxes as output, where each bounding box contains an object and also the category of the object. The RCNN first generates a set of proposals or region of interest (ROI) using selective search and then it runs all the ROIs through the pre-trained CNN to produce output features. These features are then run through a linear regression model. Fast RCNN enhances the performance by combining all the models and using ROI pooling, which slices out each ROI from the network's output tensor, reshapes it, and classifies it. While Fast R-CNN used Selective Search to generate ROIs, Faster R-CNN integrates the ROI generation into the neural network itself.

Masked RCNN is faster RCNN combined with a FCN (fully convolutional network). All the previous models focused only on object detection but Mask RCNN also focus on semantic segmentation. Thus, is allows us to perform instance segmentation which can be used for a variety of tasks (Ex: estimation of human poses). This new model also replaced the ROI Pooling with a new method called ROI align. Due to this, the strides in convolution are not quantised and thus it can represent fractions of a pixel. It preserves the spatial orientation of features with no loss of data. The definition of L_{mask} (mask loss function) allows the network to generate masks for every class without the competition among classes.

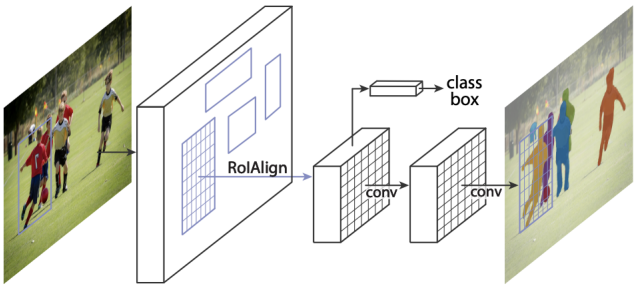


Fig. 1. The Mask R-CNN framework for instance segmentation. (He et al., 2017)

II. Related Works

A. Stereo depth vision [2]

In this blog post, the author explains the basic concepts of stereoscopic vision and how we can use the block-matching algorithm and extract depth from stereo using OpenCV library

of python.

Stereo Depth is based on the human binocular vision system. It uses two parallel placed cameras in order to calculate a image whose intensities reflect the distance of that point in that image. This image is known as depth. We calculate it by estimating disparities between matching key-points in the left and right images. For this project, to estimate these disparities, we use the OpenCV built-in block-matching algorithm. We

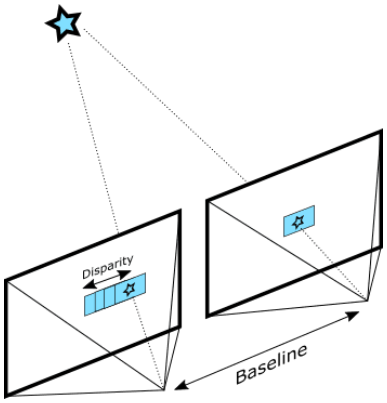


Fig. 2. Depth from Stereo algorithm finds disparity by matching blocks in left and right images

used a similar setup to this for now. The system uses a baseline of 9cm between the cameras, and the light projector is placed between the two sensors so the idea is to generate stereo images and compute depth using OpenCV library of Python.

B. Dot Pattern (Projected Texture)

Projected texture is a method of texture mapping that allows a textured image to be projected onto a scene. The quality of the results that we want to achieve with the block matching algorithm depends mainly on the density of visually distinguishable points so that the algorithm can find matching points. Thus, adding some texture increases this density and will significantly improve the accuracy.
In Blender, we created a filter in front of the light projector and added the Kinect-pattern texture. This process is explained in more detail in further sections.

C. Camera Matrix

An image pixel (u,v) is generated from world (x,y,z) coordinates through a 3x4 matrix using projective coordinates:
kx = PX,
where k is a constant, x=$(u, v, 1)^t$, X=$(X, Y, Z, W)^t$,
and P can be decomposed as:
P = K[I|0]$\begin{bmatrix} R & T \\ 0^t & 1 \end{bmatrix}$ = K[R|T]

where K can be written as : K = $\begin{bmatrix} \alpha_u & s & u_o \\ 0 & \alpha_v & v_o \\ 0 & 0 & 1 \end{bmatrix}$

where :
f = focal length

$$\alpha_u = \frac{f \times upixels}{unitlength} = \frac{f}{widthofapixelinworldunits}$$
$$\alpha_v = \frac{f \times vpixels}{unitlength} = \frac{f}{heightofapixelinworldunits}$$

u_o= u coordinate of principal point

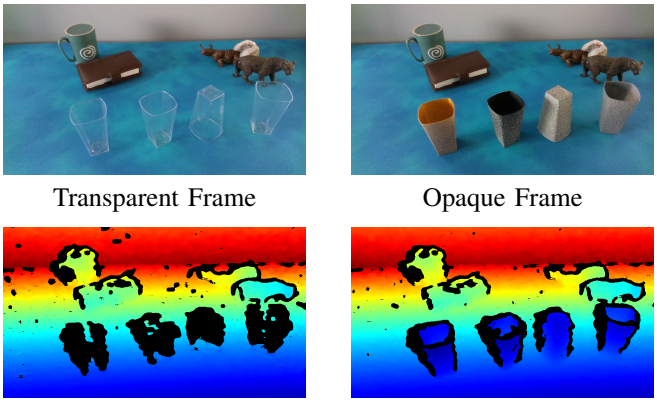
v_o=v coordinate of principal point
s=skew factor
Note : if we run the script "$get_camera_info.py$" in blender, we get the intrinsics K where entry [0,0] is f_x.

III. Dataset

The main dataset used in this project are the YCBV and ClearGrasp. YCB Object and Model Set is designed for facilitating benchmarking in robotic manipulation. The set consists of objects of daily life with different shapes, sizes, textures, weight and rigidity, as well as some widely used manipulation tests. YCB-Video dataset provides 6D poses of 21 objects from the YCB dataset observed in 92 videos with 133,827 frames. The ClearGrasp dataset contains more

Fig. 3. YCB objects

than 50,000 photorealistic renders with corresponding surface normals, segmentation masks, edges, and depth, useful for training a variety of 2D and 3D detection tasks. Each image contains up to five transparent objects, either on a flat ground plane or inside a tote, with various backgrounds and lighting.



Transparent Frame | Opaque Frame
Transparent Depth | Opaque Depth

Fig. 4. An example from ClearGrasp Dataset

IV. Tasks

A. setup a static scene and gather all outputs

My first task was just to mainly get familiarized with how to use Blender software. First, in the layout tab, we add a plane surface which will be used as a "floor" on which we place

the objects. We also add a 'area' light source with default intensity to light up the scene. Then we add a camera. Then, we import YCBV and ClearGrasp objects and set up a static scene. We can give different locations and rotations to various objects. We can also play around with their dimensions and scales. We can use the camera perspective option to make sure that the objects can be seen by the camera. Using the above

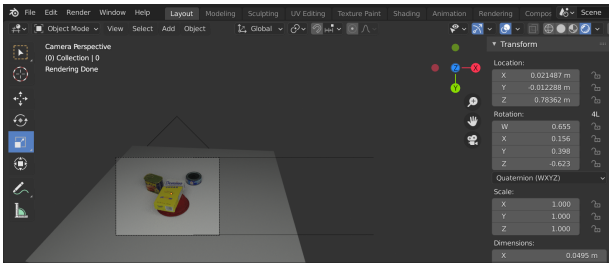


Fig. 5. A static scene setup using YCBV objects

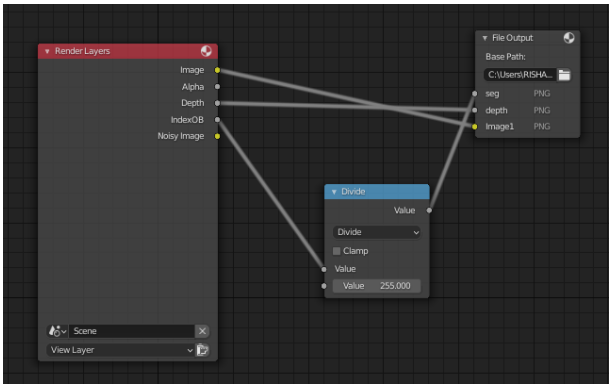


Fig. 6. Composition Tab

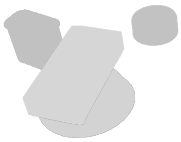
composition tab fig.6, we can output RGB images and instance segmentation. The depth that we get from this is the perfect depth which we dont need for our experiments. To get instance segmentation, first we need to give each object a specific pass index whose value ranges from 0 to 255. Then as seen from the composition tab, we add a "divide" tab to divide all these index values by 255 and then save it to a file location.

B. Setup to obtain depth and dot pattern

We add two cameras with the same camera intrinsics which are at a distance of 9cm apart. In between the cameras, we add a light projector and set its radius to 0. Then, in front of the projector, we add a plane called "filter" and give it a texture. For the "filter", in the material properties, we set the surface as 'Transparent BSDF' and then we chose 'Image Texture' and set 'kinect-pattern' as the texture. We activate the nodes and then in the shading tab, we make the following setting as shown in the fig 8. We can play around with the values in the mapping section and check which settings give the best results.

This way, we obtain the dot pattern that we want. Then, we render the output of both these cameras and then feed these

RGB image



Instance segmentation

True depth

Fig. 7. Outputs from the scene

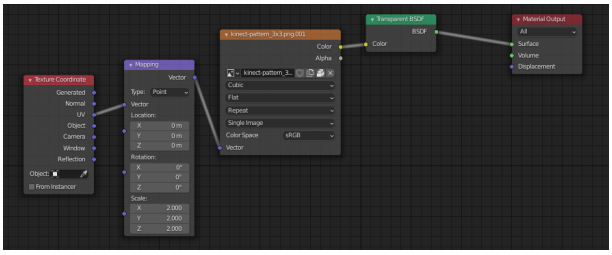
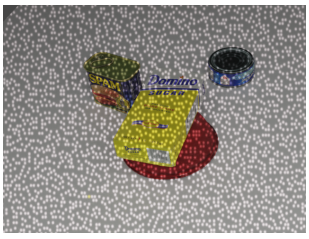


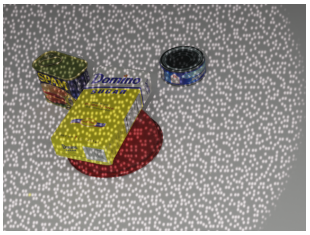
Fig. 8. Nodes in shading tab for texture mapping

two images in the blockmatching code to obtain the depth.

Blockmatching code :



left camera



right camera

Fig. 9. Rendered images of camera after applying dot pattern

```
    import numpy as np
import cv2
from google.colab.patches import cv2_imshow
from matplotlib import pyplot as plt
from skimage import io
import os
from PIL import Image
# settings
use_pattern = True
use_sgbm = False
# parameters
K = np.array([[1066.7800,    0.0000, 312.9869], [ 0.0000, 1067.4894, 241.3109],
[0.0, 0.0, 1.0]])  # -> get this from blender
fx = K[0, 0]  # lense focal length -> px disparity to mm
baseline = 90  # distance in mm between the two cameras
disparities = 128  # num of disparities to consider ->
#also limits the maximum disparity (larger image - larger value)
block = 31  # block size to match -> smallest detail vs smoothness
units = 1000.0  # depth units -> mm to meters
```

```
# load and convert image pair
left = io.imread(os.path.join(path, f'ycbv49left{"_p" if use_pattern else ""}.png'))
right = io.imread(os.path.join(path, f'ycbv49right{"_p" if use_pattern else ""}.png'))
left_gray = cv2.cvtColor(left, cv2.COLOR_BGR2GRAY)
right_gray = cv2.cvtColor(right, cv2.COLOR_BGR2GRAY)
# compute disparity (via block matching) and convert to depth
if use_sgbm:
    sbm = cv2.StereoSGBM_create(numDisparities=disparities, blockSize=block)
else:
    sbm = cv2.StereoBM_create(numDisparities=disparities, blockSize=block)
disparity = sbm.compute(left_gray, right_gray).astype(float)/16
depth = np.zeros(shape=left_gray.shape).astype(float)
depth[disparity > 0] = (fx * baseline) / (units * disparity[disparity > 0])
depth = depth
# plot disparity and depth
plt.subplot(1, 2, 1)
plt.title('disparity [px]')
plt.imshow(disparity, 'gray')
plt.subplot(1, 2, 2)
plt.title('depth [m]')
plt.imshow(depth.clip(0, 3), 'gray')
# clip depth to range [0, 3m] to remove outliers for visualization
plt.show()
```

The Blockmatching code has various parameters. f_x is the focal length of the camera which we set in Blender. You can use the default value of blender. Later on, for testing, we used Intrinsic camera matrix from the YCBV BOP file of a particular scene, to calculate the focal length. Baseline is the distance between the two cameras. For now, I set it as 9cm as mentioned in the Intel blog post. Disparity parameter controls the number of disparities to be considered between the two images. If the size of the image increases, then we need to set a larger value for this. This algorithms divides the image into various blocks and then checks if it matches with the blocks in the other image. The block parameter sets the size of this block. There is a tradeoff between smallest detail vs smoothness. OpenCV library has built-in functions to compute the disparities. By using this, we can find the depth. Check the code for reference.

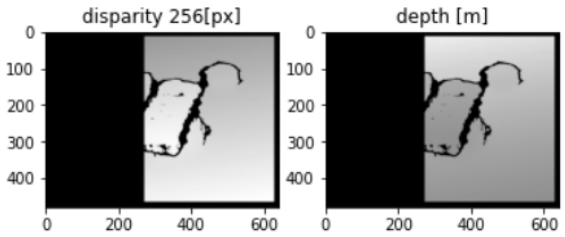


Fig. 10. Ouput of blockmatching code for Block = 31 and disparity = 256

V. Testing with YCBV scenes to check pipeline

A. proper scene set-up: load poses from bop files

RGB images when rendered should align with the real RGB image for the same scene/frame. If they align, that means our camera settings and object poses are correct

As explained earlier in subsection IV-A, we setup a scene. For my first test, I used folder 000049's 1st scene from the ycbv_test_bop_2019 dataset. Later I did the same testing for folder 000049's 1130 scene and folder 000056's 1049 scene. The folder 000049 contains scene_camera.json BOP file which contains information on the camera setups. Cam_K is the intrinsic matrix. We use the equations given in subsection II-C to calculate the camera properties. These equations help us to calculate the focal length, Shift X, Shift Y, height and width and then set these values in the blender. We chose the camera sensor fit setting as "Horizontal". We choose these same setting for both the cameras. We place the left camera at origin and the right camera at baseline distance (x) away from the left camera. We set it at position (x,0,0) to make sure both the cameras have a parallel view port. Then we set the rotation of both these cameras as $(0\circ, 180\circ, 180\circ)$ as we need a top view of the scene according to the BOP file. Then, we need to setup a plane surface "floor". Cam_R_w2c is the rotation matrix of the floor. We convert it into quarternion values and then set these value in the "floor" location. Cam_t_w2c is the coordinates of the floor. We set the dimensions and scales of the floor such that it covers the camera perspective.

scene_gt.json BOP file contains the coordinates and rotation matrix of all the objects present in that particular scene. We follow the same process as we did for the floor to setup the location and orientation of all the objects.

Note: Some of the YCBV objects have offset values which need to be taken into account while setting up the scene. Moreoever, these offsets are in the model space so we have to apply the negative offset to all the objects in the model space.

After our scene is setup, we output a RGB image and then compare it with the real image and its instance segmentation. As we can see from the fig 11, they match perfectly. Hence our camera setting and poses are correct.

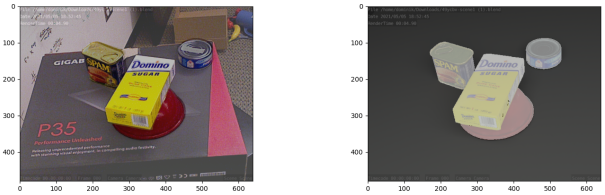


Fig. 11. Checking the alignment of RGB image

B. render the dot-pattern image and run the depth estimation pipeline

In this step, we have to visualize the absolute difference between the estimated and real depth image. If they align and error on the objects is in the range of mm, then the pipeline is valid.

We place a spotlight projector in between the two cameras. Make sure that the radius of this projector is 0. We setup the dot pattern as mentioned in subsection IV-B. After applying the dot pattern, we render the RGB image outputs and feed it into the blockmatching code. We compute the absolute difference between the depth estimated from the algorithm and the real depth (present in the BOP files). From the fig 12, we can see that there is still some error in the absolute difference.

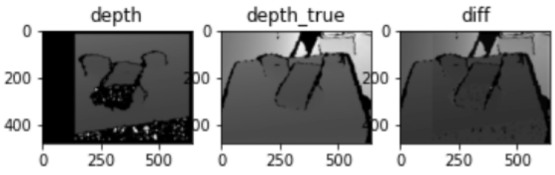


Fig. 12. Absolute difference between estimated and real depth

C. Analysis of various parameters of the block matching algorithm

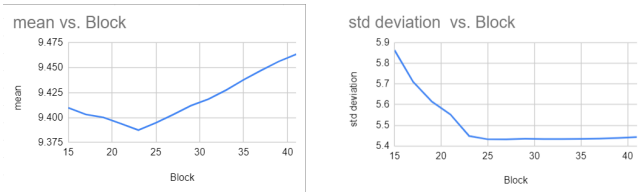
For the same scene, we conducted experiments by changing disparity and block size parameters and output the min,max,mean and std deviation of the absolute difference of the estimated and real depth.

Disparity	Max	Min	Mean	Std Deviation
16	172.0904	0	10.5254518	10.46495031
32	170.6848	0	9.925622579	7.892287211
64	163.0648	0	9.779262983	6.093765765
128	35.72472558	0	9.418392191	5.433711155
256	25.58	0	9.499913867	5.482522733

TABLE I
VS DISPARITY. HERE, BLOCK = 31

Block	Max	Min	Mean	Std Deviation
15	170.6848	0	9.409773496	5.862876202
17	170.6848	0	9.403088239	5.71029012
19	132.2512	0	9.400118331	5.614950379
21	132.2712	0	9.393967102	5.552326695
23	132.2612	0	9.387412999	5.448593863
25	51.6632	0	9.394742657	5.43287788
27	35.72472558	0	9.403104762	5.432608308
29	36.57531429	0	9.411953376	5.435234286
31	35.72472558	0	9.418392191	5.433711155
33	25.58	0	9.42747528	5.433707234
35	25.58	0	9.437775842	5.434491126
37	25.58	0	9.447197217	5.436133139
39	25.58	0	9.456099276	5.438868621
41	25.58	0	9.463306198	5.443257039

TABLE II
VS BLOCK. HERE, DISPARITY = 128



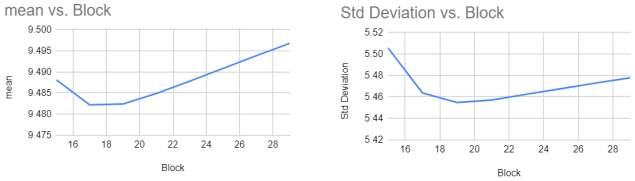
Mean vs Block Std Deviation vs Block

Fig. 13. Plots w.r.t table 2

As we can see from the above tables and plots, we should get the best results for this particular scene for Block = 23 and disparity = 128.
From fig 15, we can see that the absolute difference looks good enough and thus we can say that our pipeline is doing a good job. We also conducted experiments to check the effect of baseline parameter. We realised that its better to keep it a lower value like 2cm. On decreasing the baseline, we can get better results for lower values of disparity and thus we don't lose parts of the image.

Block	Max	Min	Mean	Std Deviation
15	153.61632	0	9.488126049	5.5058279
17	101.7959429	0	9.482188533	5.463616555
19	87.2602	0	9.48237901	5.454683467
21	25.58	0	9.484882824	5.456999555
23	25.58	0	9.487777756	5.462426938
25	25.58	0	9.490827167	5.467644527
27	25.58	0	9.493842781	5.472822304
29	25.58	0	9.496771429	5.47779685

TABLE III
VS BLOCK. HERE, DISPARITY = 256



Mean vs Block Std Deviation vs Block

Fig. 14. Plots w.r.t table 3

REFERENCES

Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 2980–2988, 2017. doi: 10.1109/ICCV.2017.322.

"The basics of stereo depth vision – Intel® RealSense™ Depth and Tracking Cameras", Intel® RealSense™ Depth and Tracking Cameras, 2021. [Online]. Available: https://www.intelrealsense.com/stereo-depth-vision-basics/. [Accessed: 14- Jun- 2021]

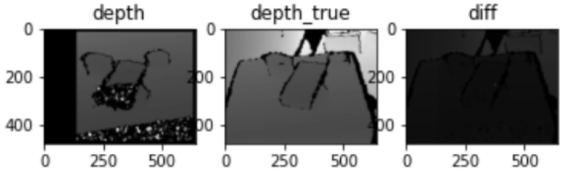


Fig. 15. Ouput of blockmatching code for Block = 23 and disparity = 128