HoloJest

Project Members

- **Aghin Shah Alin K**

  Department of Computer Science and Engineering,IIT Madras.

  2nd Year UG. aghinsa@gmail.com.
- **Akhil Sajeev**

  Department of Mechanical Engineering,IIT Madras.

  2nd Year UG. akhilsajeev9@gmail.com.
- **Vishnu Raj**

  Department of Electrical Engineering,IIT Madras.

  2nd Year UG. vishnurj99@gmail.com.

Content

Introduction

HoloJest is a combination of 3d reconstruction modules. We have divided the project to two modules,3d scene reconstruction from multiple images and reconstruction of humanoid characters from pencil drawings.The final stage is to present these reconstructed models and holograms.

Reconstruction of a humanoid character from pencil drawing is a non trivial task as lot of information is lost when the character is plotted in a specific two dimensional view.Developing a universal mathematical model for such a task is out of the scope as the characteristics for each input differs.Thus here we use Deep Learning techniques so that the system can learn to generate the required data whatever the input might be.

Since we didn't want the presented 3d models to be restricted to humanoids, we added a multi view reconstruction module based on Computer Vision techniques.Here multiple images of any scene can be the input.

Project Modules

Sketch To 3D

The module is based on the paper *3D Shape Reconstruction from Sketches via Multi-view Convolutional Networks .* Our model is a variant of the one described in the paper.The model takes as input the front view and side view of an humanoid character and outputs the depth maps and normal maps from different views (12 vertices of an Icosahedron).These outputs are later fused using an open source software resulting in the point cloud and mesh of the object.

The neural network model consists of an encoder which encodes the inputs to a latent space ,Twelve set of decoders each taking the output of the encoder as input.Each decoder is then trained to produce a separate view of the humanoid object.The decoders are trained to minimize pixel wise loss,aside from these we also have an adversarial loss.The adversarial model has the decoders as generators and encoder like model as advisory.Minimizing all these losses (weights are assigned) gives a decent output.

Model

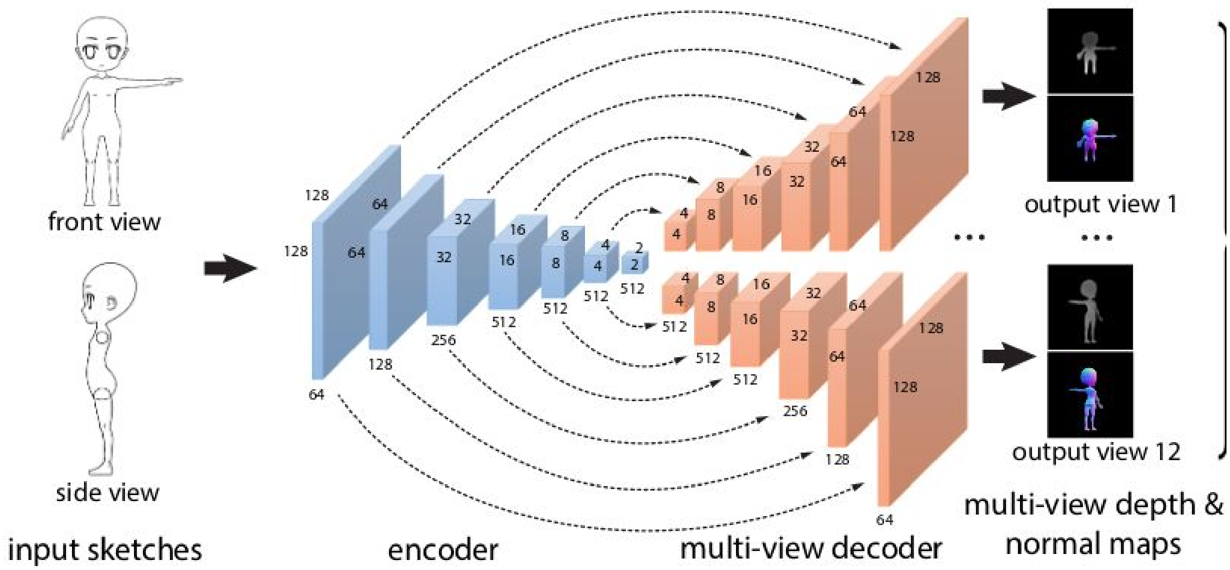


Image source was obtained from the mentioned research paper

Code for defining the model:

```
import numpy as np
import tensorflow as tf
import tensorflow.contrib.layers as tf_layers
import tensorflow.contrib.framework as framework
import module.config as config
from functools import partial
from tensorflow.nn import
```

```
sparse_softmax_cross_entropy_with_logits as cross_entropy
import module.adversial as adversial

main_dir = config.main_dir
training_iter = config.training_iter
batch_size = config.batch_size
learning_rate = config.learning_rate

def encoderNdecoder(
      images,
      out_channels=5,
      views=12,
      normalizer_fn=tf_layers.batch_norm,
      activation=tf.nn.leaky_relu):

      with tf.name_scope("model"):
      images=tf.reshape(images,[-1,256,256,2])
      #images = tf.cast(images, tf.float32)
      with tf.variable_scope("encoder"):
            with framework.arg_scope([tf_layers.conv2d],
                                          kernel_size=4, stride=2,
normalizer_fn=normalizer_fn,

activation_fn=tf.nn.leaky_relu, padding="same"):
                  e1 = tf_layers.conv2d(images,
num_outputs=64)
                  e2 = tf_layers.conv2d(e1, num_outputs=128)


                  e3 = tf_layers.conv2d(e2, num_outputs=256)
                  e4 = tf_layers.conv2d(e3, num_outputs=512)
                  tf.add_to_collection('checkpoints',e4)
                  e5 = tf_layers.conv2d(e4, num_outputs=512)
                  e6 = tf_layers.conv2d(e5, num_outputs=512)
                  encoded = tf_layers.conv2d(e6,
num_outputs=512)
tf.add_to_collection('checkpoints',encoded)
```

```
        va = []
        with tf.name_scope("decoders"):
            for count in range(views):
                with
    tf.variable_scope("decoder_{}".format(count)):
                    d6 = tf_layers.dropout(upsample(encoded,
    512))
                    d5 = tf_layers.dropout(
                        upsample(tf.concat([d6, e6], 3), 512))
                    d4 = upsample(tf.concat([d5, e5], 3), 512)
                    tf.add_to_collection('checkpoints',d4)
                    d3 = upsample(tf.concat([d4, e4], 3), 256)
                    d2 = upsample(tf.concat([d3, e3], 3), 128)
                    tf.add_to_collection('checkpoints',d2)
                    d1 = upsample(tf.concat([d2, e2], 3), 64)
                    tf.add_to_collection('checkpoints',d1)

                    decoded = upsample(
                        tf.concat(
                            [
                                d1,
                                e1],
                            3),
                        out_channels,
                        activation_fn=tf.nn.tanh,
                        normalizer_fn=tf_layers.batch_norm)

                    decoded = tf.nn.l2_normalize(
                        decoded,
                        axis=[1, 2, 3],
                        epsilon=1e-12,
                        name=None
                    )
                    va.append(decoded)

        results = tf.stack(
            (va[0],va[1],va[2],va[3],va[4],va[5],
            va[6],va[7],va[8],va[9],va[10],va[11]),axis=-1)
```

```
        results = tf.transpose(results, [0, 4, 1, 2, 3])

        return results
def depth_loss(pred, truth, mask,normalize):
        """
        pred=nx12xhxwx1
        truth="
        mask="
        return normalized loss scalar
        """

        with tf.name_scope("depth_loss"):
        loss = tf.subtract(tf.reshape(pred,[-1,12,256,256]),

tf.reshape(truth,[-1,12,256,256]))
        loss = tf.abs(loss)
        loss = tf.multiply(loss, mask)
        if(normalize):
                nloss = tf.reduce_mean(loss)

nloss=nloss*tf.constant(config.batch_size*12,dtype=tf.float32
)
        else:
                nloss=tf.reduce_sum(loss)

        return nloss


def normal_loss(pred, truth, mask,normalize):
        """
        pred=nx12xhxwx3
        truth="
        mask="
        return normalized loss scalar
        """
        with tf.name_scope("normal_loss"):
        loss = tf.subtract(pred, truth)
        m = mask
```

```
        mask = tf.stack((mask, m, m), -1)
        loss = tf.square(loss)
        loss = tf.multiply(loss, mask)
        if(normalize):
            nloss = tf.reduce_mean(loss)

nloss=nloss*tf.constant(config.batch_size*12*3,dtype=tf.float
32)
        else:
            nloss=tf.reduce_sum(loss)
        return nloss


def mask_loss(pred, truth,normalize):
        # [-1,1] -> [0,1]
        with tf.name_scope("mask_loss"):
        pred = pred * 0.5 + 0.5
        #truth is already 0,1

        loss = tf.multiply(truth, tf.log(tf.maximum(1e-6,
pred)))
        loss = loss + tf.multiply((1 - truth),
tf.log(tf.maximum(1e-6, 1 - pred)))
        loss = tf.reduce_sum(-loss)
        if (normalize):
            nloss = loss / tf.constant(12*256 *
256,dtype=tf.float32)
        else:
            nloss=loss
        return nloss

def total_loss(pred, truth,normalize=config.loss_normalize):
        """
        pred=n,12,h,w,5
        truth is a tuple

        returns total pixel loss
        """
```

```
        with tf.name_scope("total_pixel_loss"):
        truth = truth[0]
        truth=tf.reshape(truth,[-1,12,256,256,5])
        depth_pred = pred[:, :, :, :, 0]
        depth_truth = truth[:,:, :, :, 0]
        normal_pred = pred[:, :, :, :, 1:4]
        normal_truth = truth[:,:, :, :, 1:4]
        mask_pred = pred[:, :, :, :, 4]
        mask_truth = truth[:,:, :, :, 4]
        dl = depth_loss(depth_pred, depth_truth,
mask_truth,normalize)
        nl = normal_loss(normal_pred, normal_truth,
mask_truth,normalize)
        ml = mask_loss(mask_pred, mask_truth,normalize)
        return (dl + ml + nl)
def
get_adversial_loss(prob_pred,prob_truth,total_pixel_loss):
        """
        pred :n,12,256,256,5
        returns loss_gen,loss_adv
        """
        with tf.name_scope("adversarial_loss"):


truth_labels=tf.ones(tf.shape(prob_truth)[0],dtype=tf.int32)

loss_on_truth=cross_entropy(logits=prob_truth,labels=truth_la
bels)
        loss_on_truth=tf.reduce_mean(loss_on_truth)


pred_labels=tf.zeros(tf.shape(prob_pred)[0],dtype=tf.int32)

loss_on_pred=cross_entropy(logits=prob_pred,labels=pred_label
s)
        loss_on_pred=tf.reduce_mean(loss_on_pred)

        loss_adv=loss_on_truth+loss_on_pred
```

```
      #generators loss


#loss_gen_adv=tf.reduce_sum(-tf.log(tf.maximum(prob_pred,
1e-6)))

pred_labels_adv=tf.ones(tf.shape(prob_pred)[0],dtype=tf.int32
)

loss_gen_adv=cross_entropy(logits=prob_pred,labels=pred_label
s_adv)
      loss_gen_adv=tf.reduce_mean(loss_gen_adv)
      #for the adversory prediction should be of class 1
,same as truth
      loss_gen=(config.lambda_pixel*total_pixel_loss) +
(config.lambda_adv*loss_gen_adv)

return loss_gen,loss_adv

def upsample(
      x,
      n_channels,
      kernel=4,
      stride=2,
      activation_fn=tf.nn.leaky_relu,
      normalizer_fn=tf_layers.batch_norm):
      """
      x is encoded
      """
      h_new = (x.get_shape()[1].value) * stride
      w_new = (x.get_shape()[2].value) * stride
      up = tf.image.resize_nearest_neighbor(x, [h_new,
w_new])

      return tf_layers.conv2d(
      up,
      num_outputs=n_channels,
      kernel_size=kernel,
```

```
            stride=1,
            normalizer_fn=normalizer_fn,
        activation_fn=activation_fn)
        def discriminate(images):
            """
            input:[n,h,w,5]
            returns probs n,1
            """
            images=tf.reshape(images,[-1,256,256,5])
            with tf.variable_scope("discriminator", reuse=
        tf.AUTO_REUSE):
            with
        framework.arg_scope([layers.conv2d],kernel_size=4,stride=2,ac
        tivation_fn=tf.nn.leaky_relu,

        normalizer_fn=tf.contrib.layers.batch_norm,padding="same"):
                net=layers.conv2d(images,num_outputs=64)
                net=layers.conv2d(net,num_outputs=128)
                net=layers.conv2d(net,num_outputs=256)
                tf.add_to_collection('checkpoints',net)
                net=layers.conv2d(net,num_outputs=512)
                net=layers.conv2d(net,num_outputs=512)
                net=layers.conv2d(net,num_outputs=512)
                tf.add_to_collection('checkpoints',net)
                net=layers.conv2d(net,num_outputs=512)

            probs=tf.reshape(net,[-1,2048])

        probs=layers.fully_connected(probs,num_outputs=2,activation_f
        n=tf.nn.sigmoid)


        return probs
```

Code to run the module:

```
import os
```

```
import tensorflow as tf
import cv2
import numpy as np
import module.model as model
import sys

def read_input(source_directory, value=0):
      """
      return:[256,256,2] dtype=float32
      """
      f_dir = os.path.join(source_directory,
'sketch-F-{}.png'.format(value))
      #f_dir = os.path.join(source_directory, '1.jpeg')
      c0 = cv2.imread(f_dir, 0)
      c0=cv2.resize(c0,(256,256))
      c0 = normalize_image(c0)
      s_dir = os.path.join(source_directory,
'sketch-S-{}.png'.format(value))
      #s_dir = os.path.join(source_directory, '2.jpeg')
      c1 = cv2.imread(s_dir, 0)
      c1=cv2.resize(c1,(256,256))
      c1 = normalize_image(c1)
      temp = np.stack((c0, c1), axis=-1)
      return np.float32(temp)
def normalize_image(image):
      # normalize to [-1.0, 1.0]
      if image.dtype == np.uint8:
      return image.astype("float") / 127.5 - 1.0
      elif image.dtype == np.uint16:
      return image.astype("float") / 32767.5 - 1.0
      else:
      return image.astype("float")
def test(image,sess,train_dir):
      print("testing")
      saver=tf.train.Saver()

      ckpt = tf.train.get_checkpoint_state(train_dir)
      if ckpt and ckpt.model_checkpoint_path:
```

```
      saver.restore(sess, ckpt.model_checkpoint_path)
      try:
            self.step =
int(ckpt.model_checkpoint_path.split('/')[-1].split('-')[-1])
      except ValueError:
            self.step = 0
      else:
      print('Cannot find any checkpoint file')
      return
      print(ckpt)
def unnormalize_image(image, maxval=255.0):
    # restore image to [0.0, maxval]
    return (image+1.0)*maxval*0.5
def saturate_image(image, dtype=tf.uint8):
    return tf.saturate_cast(image, dtype)
def write_image(name, image):
    """
      input:
            name:  String   file name
            image: String   PNG-encoded string
    """
    path = os.path.dirname(name)
    if not os.path.exists(path):
      os.makedirs(path)
    file = open(name, 'wb')
    file.write(image)
    file.close()
def apply_mask(content, mask):
      content=tf.reshape(content,[256,256,-1])
      mask=tf.reshape(mask,[256,256,1])
      channel=content.get_shape().as_list()[-1]
      m=tf.tile(mask,[1,1,channel])
      masked=tf.multiply(content,m)
      return masked

#creating dummy target images

def collect(main_dir):
```

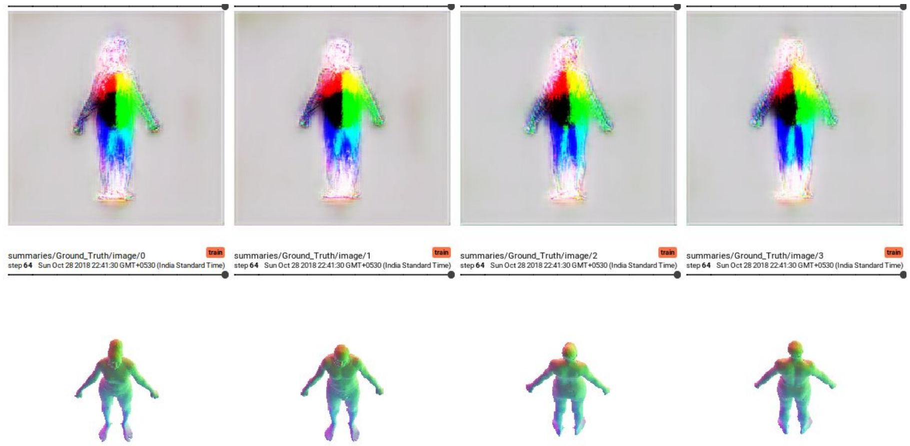
```
        #run this from input image folder

        #main_dir='./Datasets/model_name/'
        #input_image=read_input('./Datasets/model_name/images')
        input_image=read_input(os.path.join(main_dir,'images'))
        print("Reading image")
        input_image=np.reshape(input_image,[-1,256,256,2])
        train_dir=('./checkpoints')

x=tf.placeholder(dtype=tf.float32,shape=[None,256,256,2])
        pred=model.encoderNdecoder(x)


        output_dir=os.path.join(main_dir,'output')
        output_image_dir=os.path.join(output_dir,'images')
        output_results=os.path.join(output_dir,'result')
        output_prefix = 'dn14'


        with tf.Session() as sess:

        #writing input image
        in_reshape=tf.reshape(input_image,[256,256,2])
        #stack verticaly

in_ver_stack=tf.concat([in_reshape[:,:,0],in_reshape[:,:,1]],a
xis=1)
        img_input =
saturate_image(unnormalize_image(in_ver_stack,
maxval=65535.0), dtype=tf.uint16)
        img_input=tf.reshape(img_input,[256,512,-1])
        png_input = tf.image.encode_png(img_input)
        png_input = sess.run(png_input)
        name_input = os.path.join(output_image_dir,'input.png')
        write_image(name_input, png_input)

        saver = tf.train.Saver()
        #saver.restore(sess,train_dir+'/model.ckpt-8500')
```

```
        saver.restore(sess,train_dir+'/model.ckpt-36500')

        feed_dict={x:input_image}
        preds=sess.run(pred,feed_dict)

        ################################################
        preds=preds*255*2
        #preds=apply_mask(preds)
        for view in range(12):
              #print(view+1)
              sys.stdout.write('\r')
              print('processing views {}%
completed'.format((view+1)*100/12))


              preds_mask=preds[0,view,:,:,4]
              preds_mask=tf.reshape(preds_mask,[256,256,1])
              preds_depth=preds[0,view,:,:,0]
              preds_depth=apply_mask(preds_depth,preds_mask)
              preds_depth=tf.reshape(preds_depth,[256,256,1])
              preds_normal=preds[0,view,:,:,1:4]
              preds_normal=apply_mask(preds_normal,preds_mask)
              #dummy truth
              target_image = tf.ones([1,256, 256, 4])
              img_gt =
saturate_image(unnormalize_image(target_image,
maxval=65535.0), dtype=tf.uint16)
              name_gt =
os.path.join(output_image_dir,('gt-'+output_prefix+'--'+str(vi
ew)+'.png'))
              png_target = tf.image.encode_png(img_gt[0,:,:,:])
              png_target=sess.run(png_target)
              write_image(name_gt, png_target)

              #result

img_output=saturate_image(unnormalize_image(preds[0,view,:,:,0
:4],maxval=65535.0),dtype=tf.uint16)
```

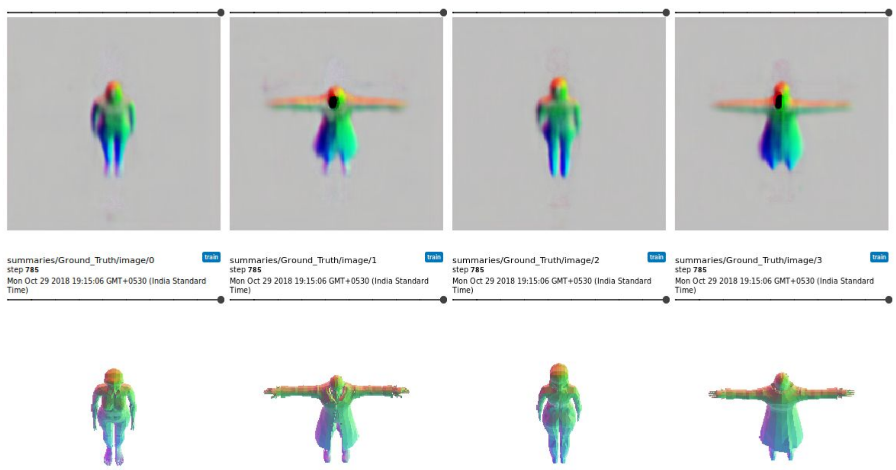
```
            png_output=tf.image.encode_png(img_output)
            name_output =
os.path.join(output_image_dir,('pred-'+output_prefix+'--'+str(
view)+'.png'))
            png_output=sess.run(png_output)
            write_image(name_output,png_output)
            #normals
            name_normal =
os.path.join(output_image_dir,('normal-'+output_prefix+'--'+st
r(view)+'.png'))
            img_normal =
saturate_image(unnormalize_image(preds_normal,
                                 maxval=65535.0),
dtype=tf.uint16)
            png_normal = tf.image.encode_png(img_normal)
            png_normal=sess.run(png_normal)
            write_image(name_normal,png_normal)
            #depth
            name_depth =
os.path.join(output_image_dir,('depth-'+output_prefix+'--'+str
(view)+'.png'))
            img_depth =
saturate_image(unnormalize_image(preds_depth,
                                 maxval=65535.0),
dtype=tf.uint16)
            png_depth = tf.image.encode_png(img_depth)
            png_depth=sess.run(png_depth)
            write_image(name_depth,png_depth)
            #mask
            name_mask =
os.path.join(output_image_dir,('mask-'+output_prefix+'--'+str(
view)+'.png'))
            img_mask =
saturate_image(unnormalize_image(preds_mask,
                                 maxval=65535.0),
dtype=tf.uint16)
            png_mask = tf.image.encode_png(img_mask)
            png_mask=sess.run(png_mask)
```

```
            write_image(name_mask,png_mask)
            #Export to results

img_output=saturate_image(unnormalize_image(preds[0,view,:,:,1
:4],
                                                    maxval=65535.0),
dtype=tf.uint16)
            png_output=tf.image.encode_png(img_output)
            name_output =
os.path.join(output_results,('pred-'+output_prefix+'--'+str(vi
ew)+'.png'))
            png_output=sess.run(png_output)
write_image(name_output,png_output)
```

Training Step 50



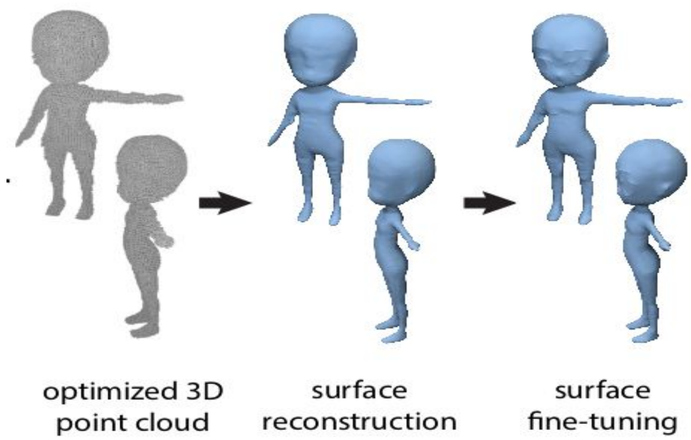
Training Step 3000



Losses

The code to configure data ,train etc and further instructions on how to run the module is provided in the github repo of the project.

The output of the module can be fused with help of external softwares to produce the 3d mesh of the object

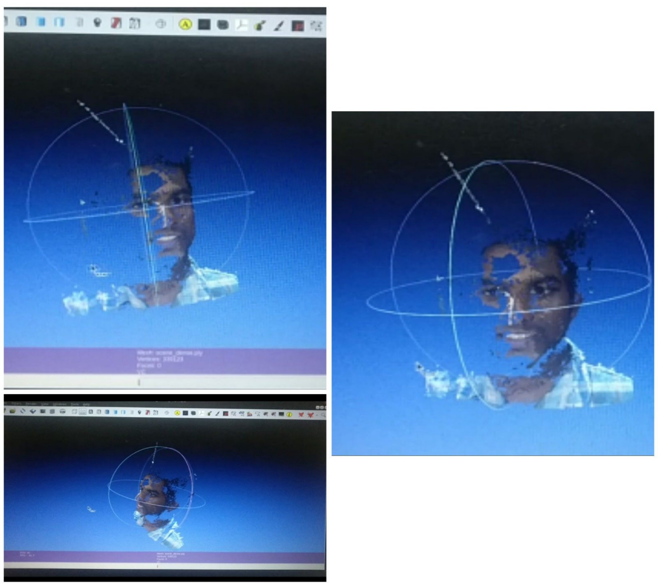


Multi-view Reconstruction

For a human, it is usually an easy task to get an idea of the 3D structure shown in an image. Due to the loss of one dimension in the projection process, the estimation of the true 3D geometry is difficult and a so called ill-posed problem, because usually infinitely many different 3D surfaces may produce the same set of images. To infer geometrical structure of a scene captured by a collection of images;using a mathematical model,some information about the setup is required.Here we provide the width of the lens used.Other parameter like camera position and internal parameters are assumed to be known or they can be estimated from the set of images. By using multiple images, 3D information can be (partially) recovered by solving a pixel-wise correspondence problem.

We are using a pipeline of open source modules to solve this correspondens problem.Namely *OpenMVG* and *OpenMVS.*We have built a container with all the required packages and necessary scripts.The dockerfile,sensor width data etc is provided in the github repo of the project.

Sample reconstruction (4 input images)



Code for the pipeline

```
import subprocess
import os
import config
import time

manual=config.manual
width=config.width
image_dir = config.image_dir
camera_dir= config.camera_dir
matches_dir = config.matches_dir
output_dir = config.output_dir


print('Starting Reconstruction...')
tic = time.clock()
#starting OpenMVG
```

```
#max_h_w=4000
if(manual):
    command = "openMVG_main_SfMInit_ImageListing -i '{}' -d '{}'
-o '{}' -f '{}' ".format(image_dir,
                    camera_dir,matches_dir,(1.2*width))
else:
    command = "openMVG_main_SfMInit_ImageListing -i '{}' -d '{}'
-o '{}' ".format(image_dir,camera_dir,matches_dir)
process = subprocess.call(command, shell=True)

command = "openMVG_main_ComputeFeatures -i '{}' -o
'{}'".format(matches_dir + '/sfm_data.json',matches_dir)
process = subprocess.call(command, shell=True)

command = "openMVG_main_ComputeMatches -i '{}' -o
'{}'".format(matches_dir + '/sfm_data.json',matches_dir)
process = subprocess.call(command, shell=True)

command = "openMVG_main_IncrementalSfM -i '{}' -m '{}' -o
'{}'".format(matches_dir +
'/sfm_data.json',matches_dir,output_dir)
process = subprocess.call(command, shell=True)

command = "openMVG_main_openMVG2openMVS -i '{}' -o
'scene.mvs'".format(output_dir + '/sfm_data.bin')
process = subprocess.call(command, shell=True)

print('Starting OpenMVS...')
#starting OpenMVS

#os.chdir(output_dir)
command = "DensifyPointCloud scene.mvs"
process = subprocess.call(command, shell=True)

command = "ReconstructMesh scene_dense.mvs"
process = subprocess.call(command, shell=True)

if(config.obj):
    command = "TextureMesh scene_dense_mesh.mvs --export-type
obj"
else:
    command = "TextureMesh scene_dense_mesh.mvs"
```

```
process = subprocess.call(command, shell=True)
toc = time.clock()
print('Completed in {} minutes'.format( (toc - tic)/60 ))
```

Softwares and Libraries Used

- Tensorflow : TensorFlow is an open source software library for high performance numerical computation.Originally developed by researchers and engineers from the Google Brain team within Google’s AI organization, it comes with strong support for machine learning and deep learning and the flexible numerical computation core is used across many other scientific domains.Further details and instructions to install can be found here.
- Docker:Docker is a computer program that performs operating-system-level virtualization, also known as "containerization".We have used docker to containerise our multiview reconstruction pipeline to avoid dependency problems.Other details can be found on the official website.
- Gradien Checkpointing (Optional):Training very deep neural networks requires a lot of memory. Using the tools in this package, you can trade off some of this memory usage with computation to make your model fit into memory more easily. This package is optional as its only required in the training phase and if you have a high end Gpu training should not be a problem.The package and regarding instructions can be found here.
- OpenMVG:OpenMVG (Multiple View Geometry) is a library for computer-vision scientists and targeted for the Multiple View Geometry community.More details and instructions can be found here.
- OpenMVS:OpenMVS (Multi-View Stereo) is a library for computer-vision scientists and especially targeted to the Multi-View Stereo reconstruction community.OpenMVS provides a complete set of algorithms to recover the full surface of the scene to be reconstructed. The input is a set of camera poses plus the sparse point-cloud and the output is a textured mesh.

- Anaconda:Anaconda is the most popular Python data science platform.It comes with many of the libraries pre installed.You can download it from the official site.

Mechanical Structure

The main component of the structure is a 0.5mm transparent acrylic/plastic (of size bigger than that of the screen used).The sheet is placed in an angle again depending on the screen size and settings.The sheet with the correct angle is fixed in a wooden box with all except two sides (front and bottom) covered.The color of the box is preferred to be black.The structure is placed on top of the screen reflecting the 3d models and producing a holographic effect.

The principle behind the holographic illusion is the Pepper's Ghost effect. Pepper's Ghost is a special effects technique for creating transparent ghostly images. It works by reflecting the image of a ghost off of a sheet of plexiglass.More about the phenomenon can be read here.

Problems Faced

- Multi view reconstruction doesn't always produce reliable outputs,and the nature of the outputs couldn't be predicted.
- Even on datasets of similar nature the outputs differed.
- Since 3d reconstruction is computationally very heavy it took a lot of time to run a test and this hindered testing with a variety of inputs.
- Installation of dependencies caused lot of trouble initially,especially when migrating to other systems,use of docker helped in solving this.
- Since the neural network is very deep we couldn't run it on our system and didn't get enough gpu time to train.
- Presenting the outputs as holograms prefers dark surroundings,which caused problems in bright areas.

Further Improvements

- Improve the adversarial loss
- Replace the vanilla GAN with an advanced version

Citations

- Zhaoliang Lun, Matheus Gadelha, Evangelos Kalogerakis, Subhransu Maji, Rui Wang, "3D Shape Reconstruction from Sketches via Multi-view Convolutional Networks",Proceedings of the International Conference on 3D Vision (3DV) 2017
    - https://github.com/happylun/SketchModeling
    - https://people.cs.umass.edu/~zlun/SketchModeling/
    - https://arxiv.org/pdf/1707.06375.pdf

Acknowledgments

The success and final outcome of this project required a lot of guidance and assistance from many people and we are extremely privileged to have got this all along the completion of our project. However, it would not have been possible without the kind support and help of many individuals and organizations. We would like to extend our sincere thanks to all of them.

We are highly indebted to Envisage™ for their guidance and constant supervision as well as for providing necessary information regarding the project & also for their support in completing the project.

We owe our deep gratitude to the Compute Vision and Intelligence Group IITM,and the cores Varun and Lokesh,for helping us with the required theoretical knowledge and the necessary hardware for the training of our model.

We heartly thank our super coordinators Madhurima Mahajan ,Sai Ratna Sajja for their support and guidance throughout the project.We thank our Cores Adhitya and Pravallika,project mentor Deepak, for their guidance and suggestions during this project work.