Foveated Rendering

Team Fovea

Scott McElfresh, Dietrich Kruse, Mallikarjun Edara

Introduction

One of the major constraints of XR (a term that encompasses Virtual, Augmented, and Mixed Reality) is the disconnect between the potential for extremely high quality graphics (photorealistic) and the current state of the devices required to render these graphics. In XR, the device must render the same image twice per frame, which becomes increasingly intensive as we explore the realm of photorealistic experiences. While some current gaming PC's have the capabilities to render these, standalone devices such as the Oculus Quest 2 do not. For XR to reach mass adoption, the cost of standalone devices must come down and the quality must improve.

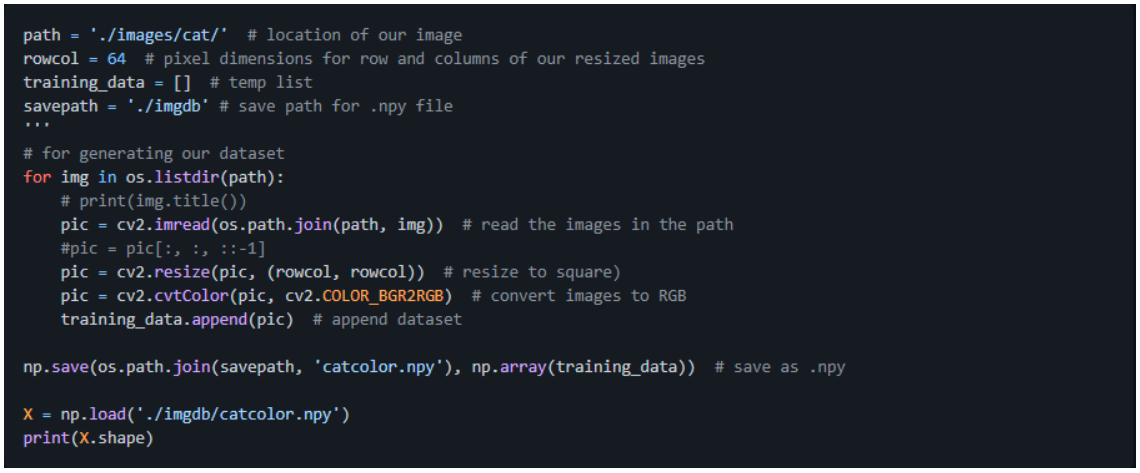
In early 2021, Team Fovea was created with the goal of reducing the cost of graphical rendering using a technique called Foveated Rendering, which mimics how the human eye works. The eye only sees full resolution of what the center of the retina, called the fovea, is pointed at. The peripheral vision is mostly generated by the brain. Foveated Rendering does this by assigning a fixation point and rendering only what is closest to the point at full resolution and decreasing resolution as distance from the point increases. Machine Learning can be integrated to "fill in the gaps" of what is being rendered in the peripheral vision, just as the brain does. In combination with eye tracking software, what the eye is looking at becomes the fixation point. This technique could be used to render only what the eye is looking at in full resolution, while rendering what is in the periphery at lower resolutions. This can reduce the cost of rendering by an order of magnitude, allowing for high quality graphics to be rendered on devices that would not have the capabilities otherwise.

For Team Fovea's deep learning project, we sought out to create a foveated image approximation and reconstruction method. Originally we explored GANS - Generative Adversarial Networks for our reconstruction but after developing our image foveation method, we pursued image generation through auto encoders. This was primarily because autoencoders allowed a more direct method for inducing our foveated images into the model

Design

Team Fovea decided to focus their efforts on just using machine learning to recreate the images from the downscaled versions, with the center of the image acting as the fixation point. For our project we used the cat directory of the Kaggle image library titled Animal Faces [https://www.kaggle.com/andrewmvd/animal-faces] This directory consists of 5153 images of cat faces.

Initially, Dietrich Kruse focused on the method for creating foveat images and Scott McElfresh focused on image reconstruction. Working through the process, we ended up collaborating on both to overcome challenges we were facing individually. After our library imports, the program starts with our image dataset creation.



```
path = './images/cat/'  # location of our image
rowcol = 64  # pixel dimensions for row and columns of our resized images
training_data = []  # temp list
savepath = './imgdb' # save path for .npy file
'''
# for generating our dataset
for img in os.listdir(path):
    # print(img.title())
    pic = cv2.imread(os.path.join(path, img))  # read the images in the path
    #pic = pic[:, :, ::-1]
    pic = cv2.resize(pic, (rowcol, rowcol))  # resize to square)
    pic = cv2.cvtColor(pic, cv2.COLOR_BGR2RGB)  # convert images to RGB
    training_data.append(pic)  # append dataset

np.save(os.path.join(savepath, 'catcolor.npy'), np.array(training_data))  # save as .npy

X = np.load('./imgdb/catcolor.npy')
print(X.shape)
```

Figure 1) Dataset Generation

In this section we define our image resize (64x64 pixels), the savepath for our numpy dataset (/imgdb), and create a list to append images to. Every image in the cat directory is a read in OpenCV, resized to 64 x 64, and gets converted from BGR to RGB (we're using OpenCV to read images but matplotlib.pylot.imshow to display images - the former operates in a BGR color space and the later in an RGB color space). These images are appended to a list and then saved as a numpy array that can be loaded as 'X'.

Next we define our sparse method that creates our representation of a foveated image. Then we define the fovea as the center of each image, perform a log polar transform on the image, record the inverse exponential distribution of pixels across the log polar transformed image, reverse the log polar transform, and then alter the pixels we recorded.

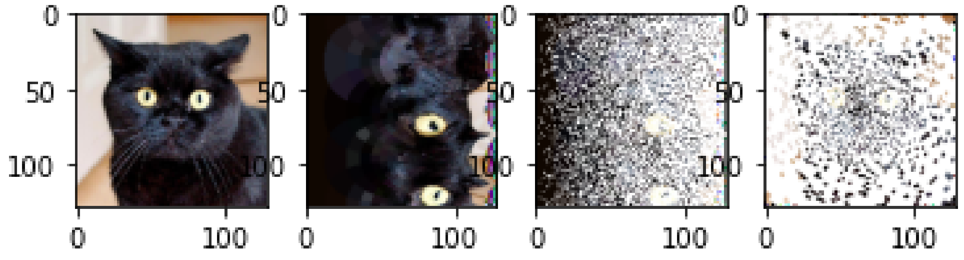
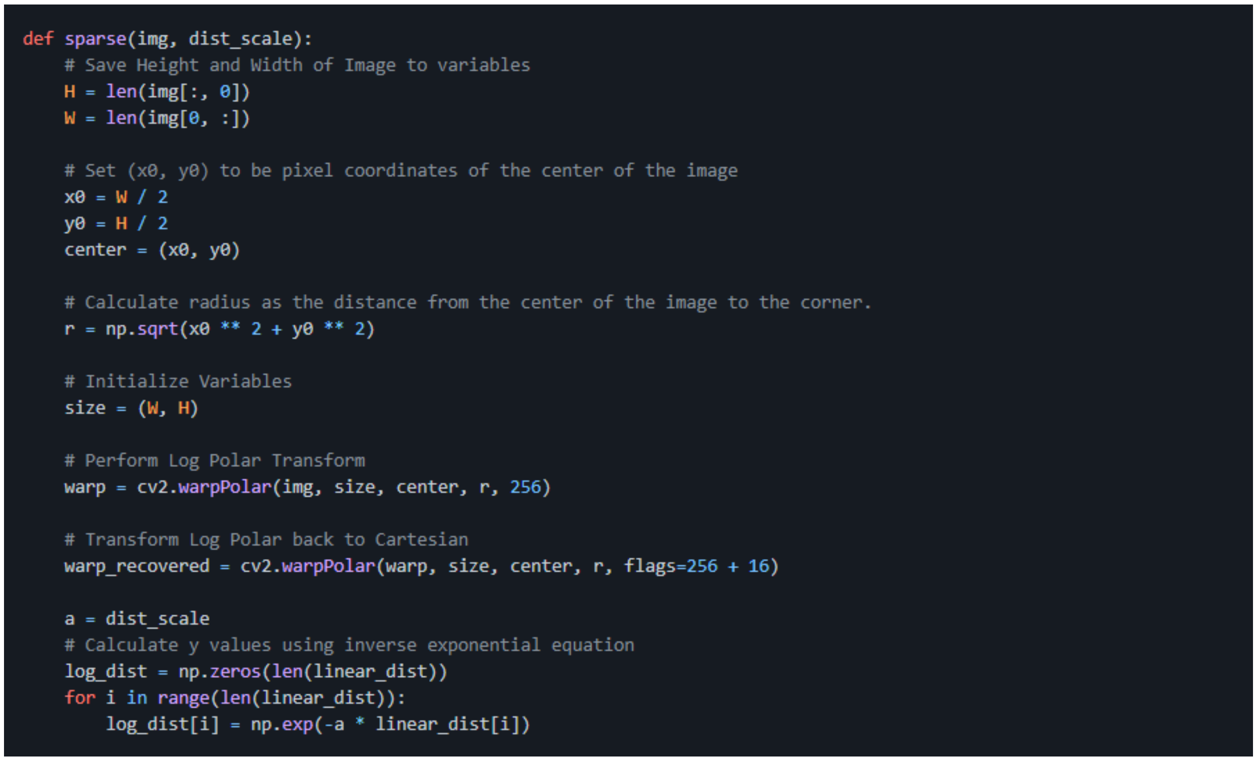


Figure 2) Original Image to Sparse Density Transformation

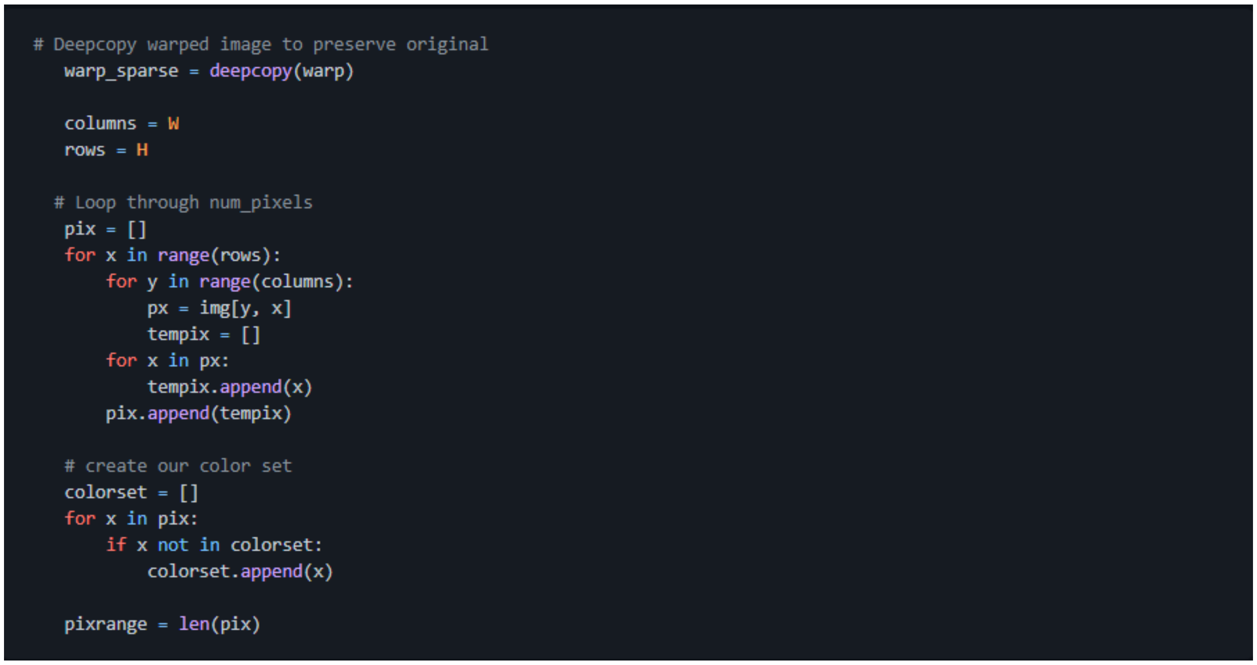


```python
def sparse(img, dist_scale):
    # Save Height and Width of Image to variables
    H = len(img[:, 0])
    W = len(img[0, :])

    # Set (x0, y0) to be pixel coordinates of the center of the image
    x0 = W / 2
    y0 = H / 2
    center = (x0, y0)

    # Calculate radius as the distance from the center of the image to the corner.
    r = np.sqrt(x0 ** 2 + y0 ** 2)

    # Initialize Variables
    size = (W, H)

    # Perform Log Polar Transform
    warp = cv2.warpPolar(img, size, center, r, 256)

    # Transform Log Polar back to Cartesian
    warp_recovered = cv2.warpPolar(warp, size, center, r, flags=256 + 16)

    a = dist_scale
    # Calculate y values using inverse exponential equation
    log_dist = np.zeros(len(linear_dist))
    for i in range(len(linear_dist)):
        log_dist[i] = np.exp(-a * linear_dist[i])
```

Figure 3) Sparse Image Generation Function, part 1

The method sparse takes in an image and dist_scale value as inputs. Sparse determines the image size and image center (our fovea). Then a radius is determined as the diagonal to the image corner. The radius is used in our cartesian reconstruction after a log polar transform is performed on the image. We determine the linear distribution across the image and the dist scale is a scaling factor for the distribution which gets transformed into an inverse exponential distribution.

In an ideal application, we would only be rendering the logarithmic distribution of an image and then process the remaining pixels with an algorithm, ideally resulting in rendering speed increase. That is the whole concept in a nutshell. We weren't able to develop this full system so we do an approximation where instead, we record the colors of our original image and fill the inverse log distribution randomly with those colors. Earlier in development, we were replacing those pixels with white, but the reconstruction used those values and generated images were very light.



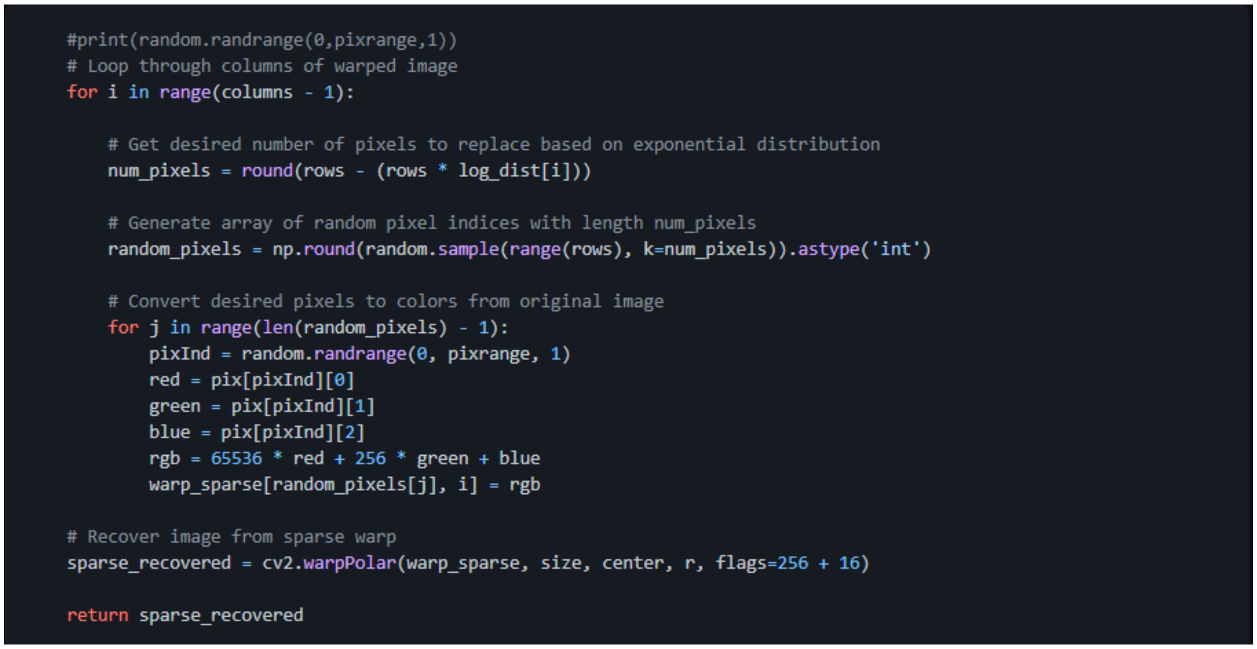
```
# Deepcopy warped image to preserve original
   warp_sparse = deepcopy(warp)

   columns = W
   rows = H

  # Loop through num_pixels
   pix = []
   for x in range(rows):
       for y in range(columns):
           px = img[y, x]
           tempix = []
       for x in px:
           tempix.append(x)
       pix.append(tempix)

   # create our color set
   colorset = []
   for x in pix:
       if x not in colorset:
           colorset.append(x)

   pixrange = len(pix)
```

Figure 4) Sparse Image Generation Function, part 2

This section loops through the image and creates a list of the RGB values of each pixel. Next we get the pixels that would have been removed with our log distribution and their indices. Those pixels are replaced with the color values recorded in our color array list. Finally the image is converted back into a cartesian representation and returned.

To convert the RGB values to an integer, this formula is used:

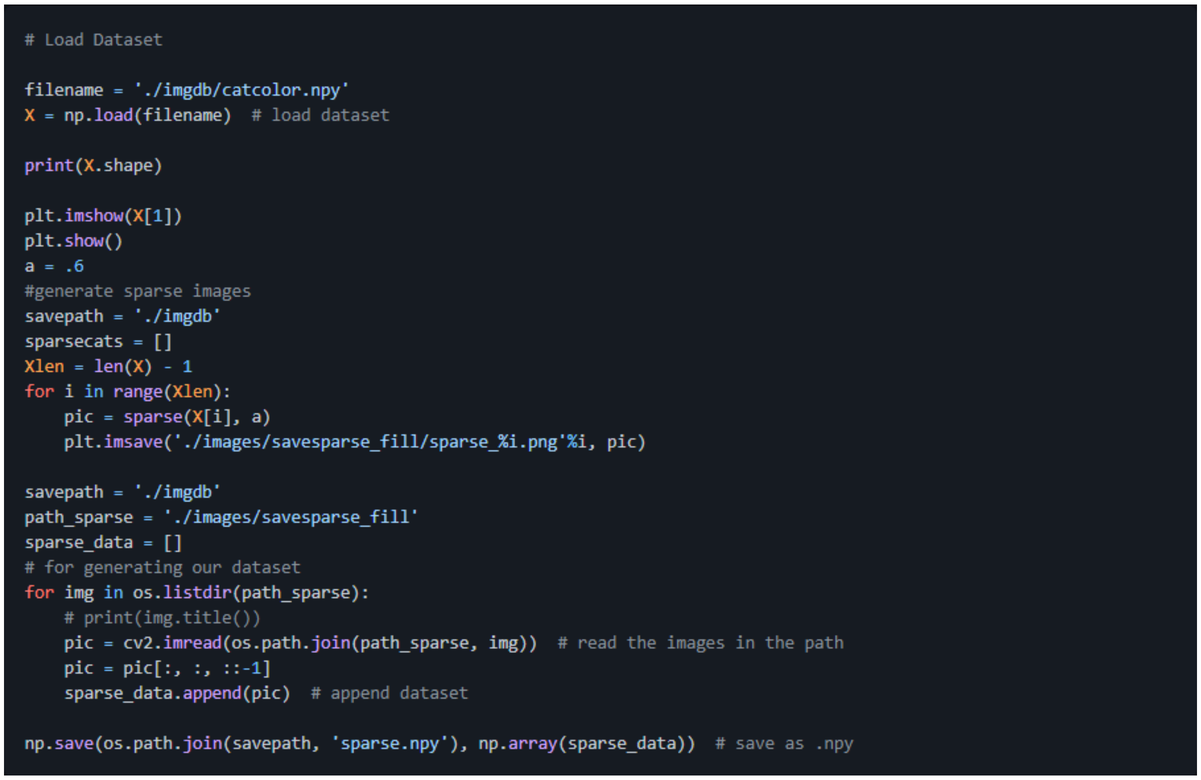
rgb = 65536 * red + 256 * green + blue



```
#print(random.randrange(0,pixrange,1))
# Loop through columns of warped image
for i in range(columns - 1):

    # Get desired number of pixels to replace based on exponential distribution
    num_pixels = round(rows - (rows * log_dist[i]))

    # Generate array of random pixel indices with length num_pixels
    random_pixels = np.round(random.sample(range(rows), k=num_pixels)).astype('int')

    # Convert desired pixels to colors from original image
    for j in range(len(random_pixels) - 1):
        pixInd = random.randrange(0, pixrange, 1)
        red = pix[pixInd][0]
        green = pix[pixInd][1]
        blue = pix[pixInd][2]
        rgb = 65536 * red + 256 * green + blue
        warp_sparse[random_pixels[j], i] = rgb

# Recover image from sparse warp
sparse_recovered = cv2.warpPolar(warp_sparse, size, center, r, flags=256 + 16)

return sparse_recovered
```

Figure 5) Sparse Image Generation Function, part 3

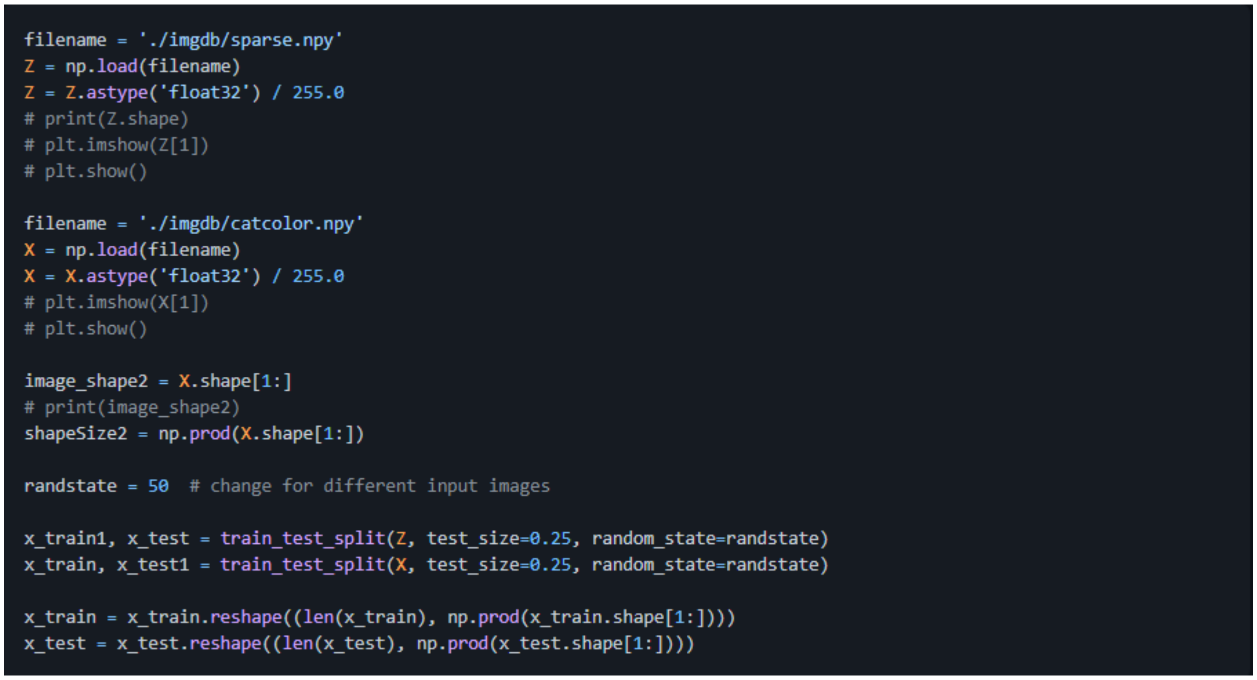
Next the program loads our original cat data set and runs the images through sparse. It saves those returned images and creates a new data set of them. This allows us to then load both the original and sparse datasets (commenting out the dataset creation sections).



```
# Load Dataset

filename = './imgdb/catcolor.npy'
X = np.load(filename)  # load dataset

print(X.shape)

plt.imshow(X[1])
plt.show()
a = .6
#generate sparse images
savepath = './imgdb'
sparsecats = []
Xlen = len(X) - 1
for i in range(Xlen):
    pic = sparse(X[i], a)
    plt.imsave('./images/savesparse_fill/sparse_%i.png'%i, pic)

savepath = './imgdb'
path_sparse = './images/savesparse_fill'
sparse_data = []
# for generating our dataset
for img in os.listdir(path_sparse):
    # print(img.title())
    pic = cv2.imread(os.path.join(path_sparse, img))  # read the images in the path
    pic = pic[:, :, ::-1]
    sparse_data.append(pic)  # append dataset

np.save(os.path.join(savepath, 'sparse.npy'), np.array(sparse_data))  # save as .npy
```

Figure 6) Generating/Saving Dataset

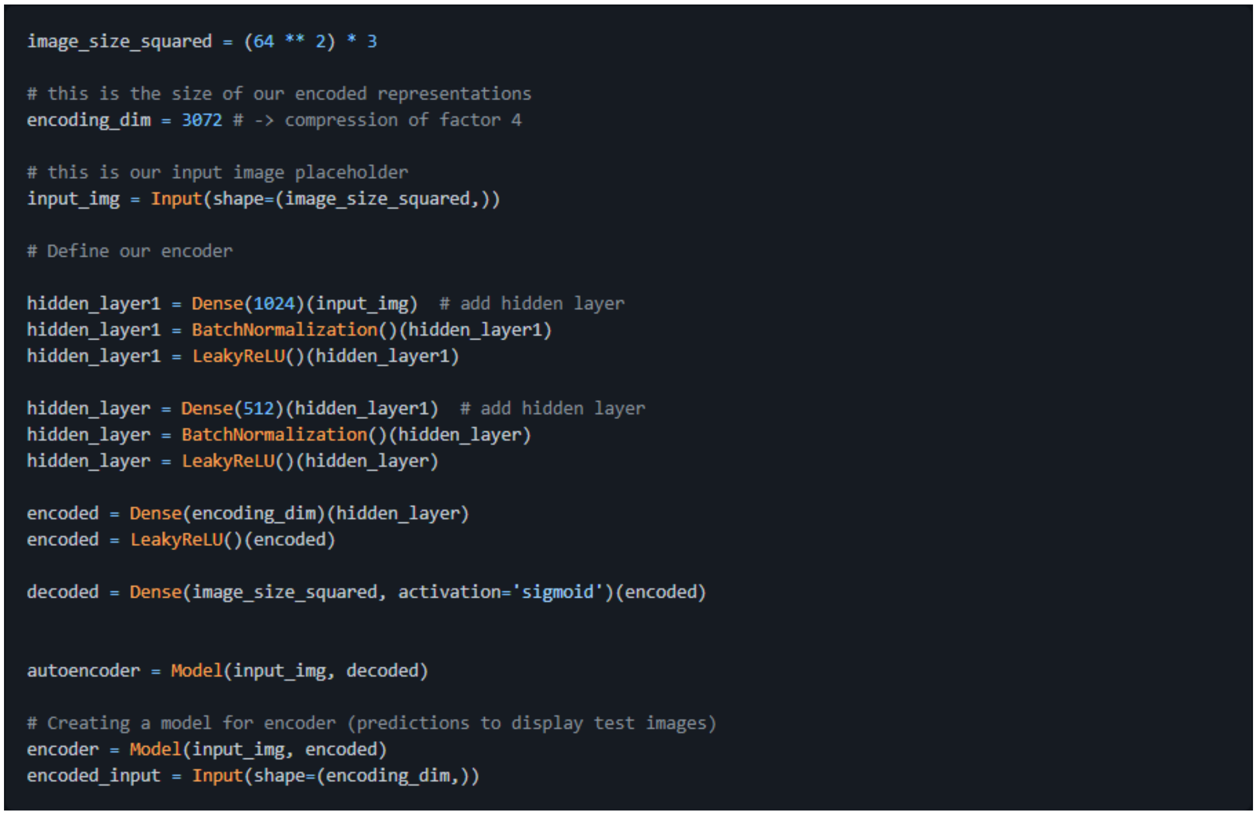
Full and Sparse image datasets are loaded as X and Z then normalized by dividing by 255.0. A training set from the full image set is created and test set from the sparse images. Utilizing the random_state argument in train_test_split ensures that the sets will be consistent in sizes and that our testing images will not be part of our training set.



```
filename = './imgdb/sparse.npy'
Z = np.load(filename)
Z = Z.astype('float32') / 255.0
# print(Z.shape)
# plt.imshow(Z[1])
# plt.show()

filename = './imgdb/catcolor.npy'
X = np.load(filename)
X = X.astype('float32') / 255.0
# plt.imshow(X[1])
# plt.show()

image_shape2 = X.shape[1:]
# print(image_shape2)
shapeSize2 = np.prod(X.shape[1:])

randstate = 50  # change for different input images

x_train1, x_test = train_test_split(Z, test_size=0.25, random_state=randstate)
x_train, x_test1 = train_test_split(X, test_size=0.25, random_state=randstate)

x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))
```

Figure 7) Loading and Splitting Dataset

Finally we define our undercomplete autoencoder. I kept a large output representation (encoding_dim) of 3072 which is a compression factor of 4 for the input images (65 * 64 * 3 = 12288). Our encoder goes from our initial input dimension to a dense layer of 1024 neurons and then bottlenecks to a layer with 512 neurons. Each hidden layer also includes a batch normalization and uses Leaky Relu activation. Batch normalization greatly improved our image color range and contrast output and also decreased training time (the model stabilizes around 80 epochs). Like in our GAN model, Leaky Relu prevents 'dead-pixels' by not allowing values to ever reach zero.



```
image_size_squared = (64 ** 2) * 3

# this is the size of our encoded representations
encoding_dim = 3072 # -> compression of factor 4

# this is our input image placeholder
input_img = Input(shape=(image_size_squared,))

# Define our encoder

hidden_layer1 = Dense(1024)(input_img)  # add hidden layer
hidden_layer1 = BatchNormalization()(hidden_layer1)
hidden_layer1 = LeakyReLU()(hidden_layer1)

hidden_layer = Dense(512)(hidden_layer1)  # add hidden layer
hidden_layer = BatchNormalization()(hidden_layer)
hidden_layer = LeakyReLU()(hidden_layer)

encoded = Dense(encoding_dim)(hidden_layer)
encoded = LeakyReLU()(encoded)

decoded = Dense(image_size_squared, activation='sigmoid')(encoded)


autoencoder = Model(input_img, decoded)

# Creating a model for encoder (predictions to display test images)
encoder = Model(input_img, encoded)
encoded_input = Input(shape=(encoding_dim,))
```

Figure 8) Encoder

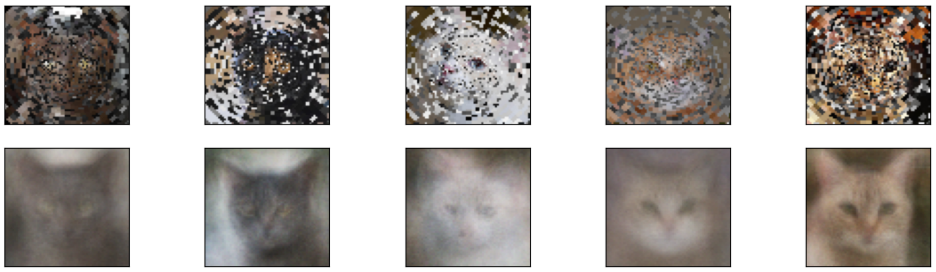


Figure 8) Output without Batch Normalization

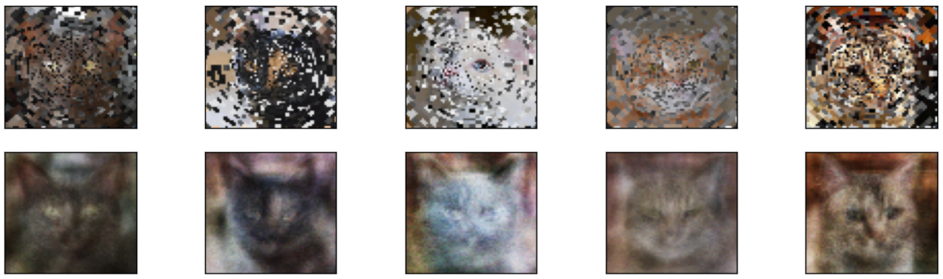
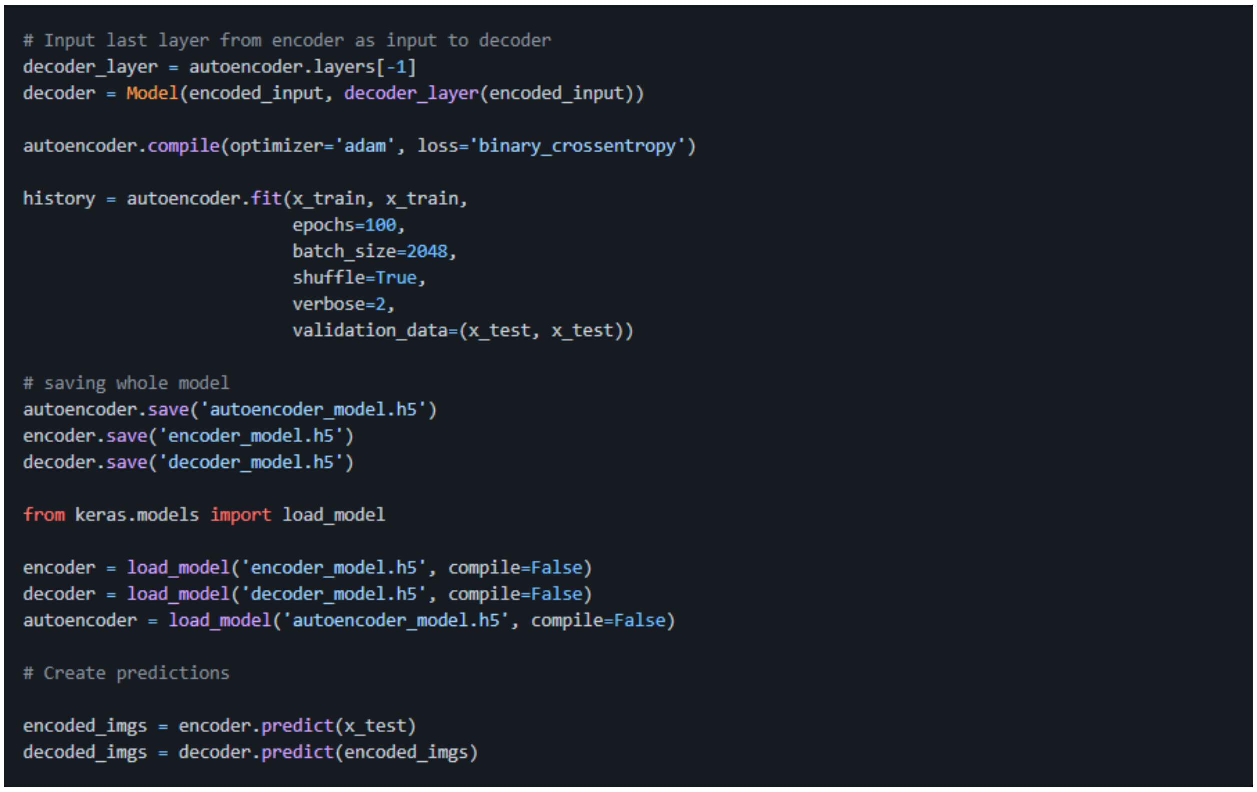


Figure 9) Output with Batch Normalization

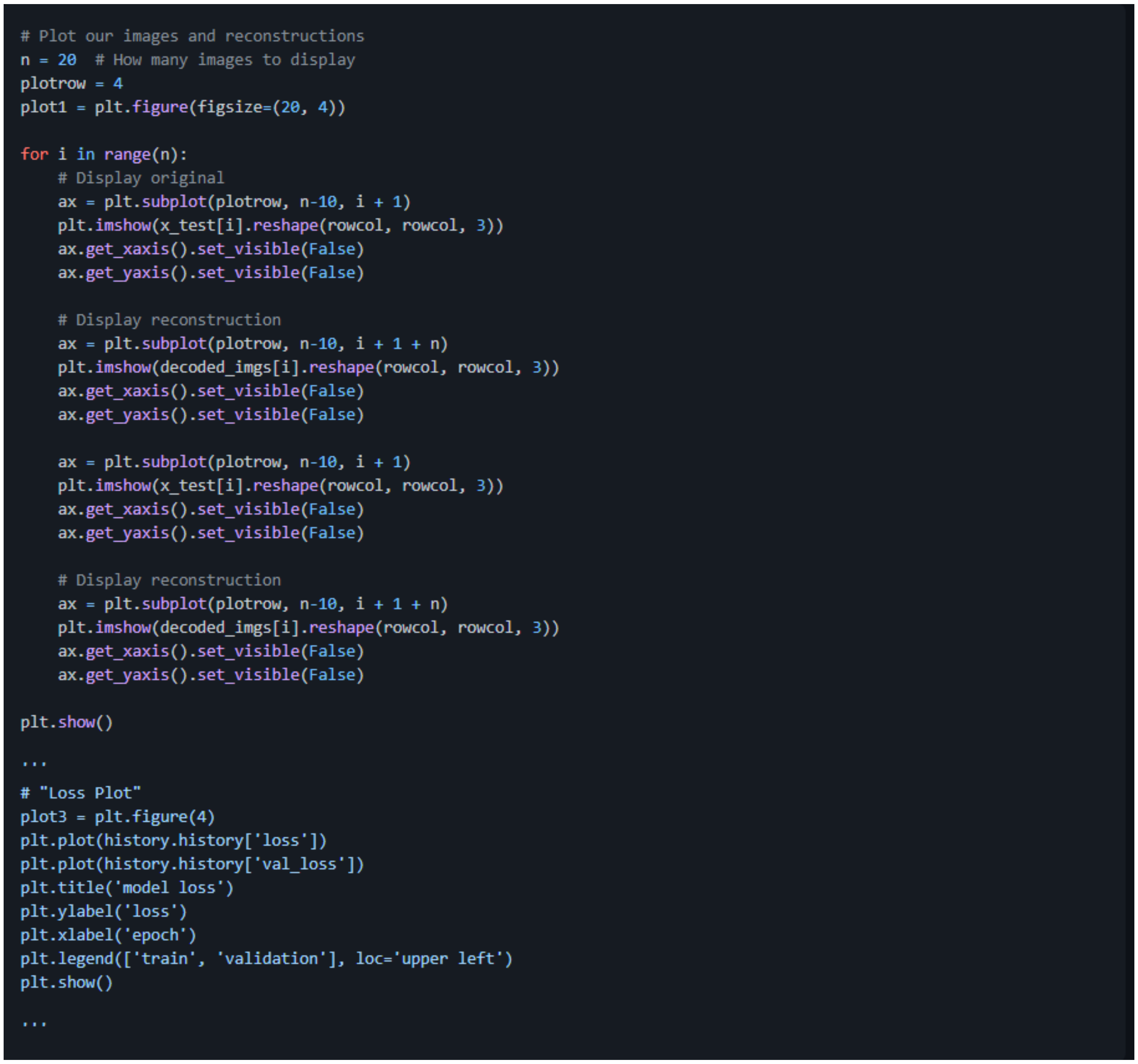
Our decoder is created as a single dense layer which is inputted the last layer of the encoder, which is of our encoding dimension size. Decoder complexity was experimented with (mirroring the architecture to the encoder) but the images generated from the network had less clarity. The simpler model that produced better results was chosen instead of continuing to pursue solutions to improving the more complex one.



```python
# Input last layer from encoder as input to decoder
decoder_layer = autoencoder.layers[-1]
decoder = Model(encoded_input, decoder_layer(encoded_input))

autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

history = autoencoder.fit(x_train, x_train,
                          epochs=100,
                          batch_size=2048,
                          shuffle=True,
                          verbose=2,
                          validation_data=(x_test, x_test))

# saving whole model
autoencoder.save('autoencoder_model.h5')
encoder.save('encoder_model.h5')
decoder.save('decoder_model.h5')

from keras.models import load_model

encoder = load_model('encoder_model.h5', compile=False)
decoder = load_model('decoder_model.h5', compile=False)
autoencoder = load_model('autoencoder_model.h5', compile=False)

# Create predictions

encoded_imgs = encoder.predict(x_test)
decoded_imgs = decoder.predict(encoded_imgs)
```

Figure 10) Decoder

The autoencoder was trained and encoder, decoder, and autoencoder models and weights saved. After training, only the datasets and models need loaded to run predictions and generate images. Originally the model was trained with a batch size of 2048 but after experimentation, a batch size of 128 was chosen for image output.



```python
# Plot our images and reconstructions
n = 20  # How many images to display
plotrow = 4
plot1 = plt.figure(figsize=(20, 4))

for i in range(n):
    # Display original
    ax = plt.subplot(plotrow, n-10, i + 1)
    plt.imshow(x_test[i].reshape(rowcol, rowcol, 3))
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Display reconstruction
    ax = plt.subplot(plotrow, n-10, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(rowcol, rowcol, 3))
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    ax = plt.subplot(plotrow, n-10, i + 1)
    plt.imshow(x_test[i].reshape(rowcol, rowcol, 3))
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Display reconstruction
    ax = plt.subplot(plotrow, n-10, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(rowcol, rowcol, 3))
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

plt.show()

...

# "Loss Plot"
plot3 = plt.figure(4)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()

...
```

Figure 11) Plotting Results

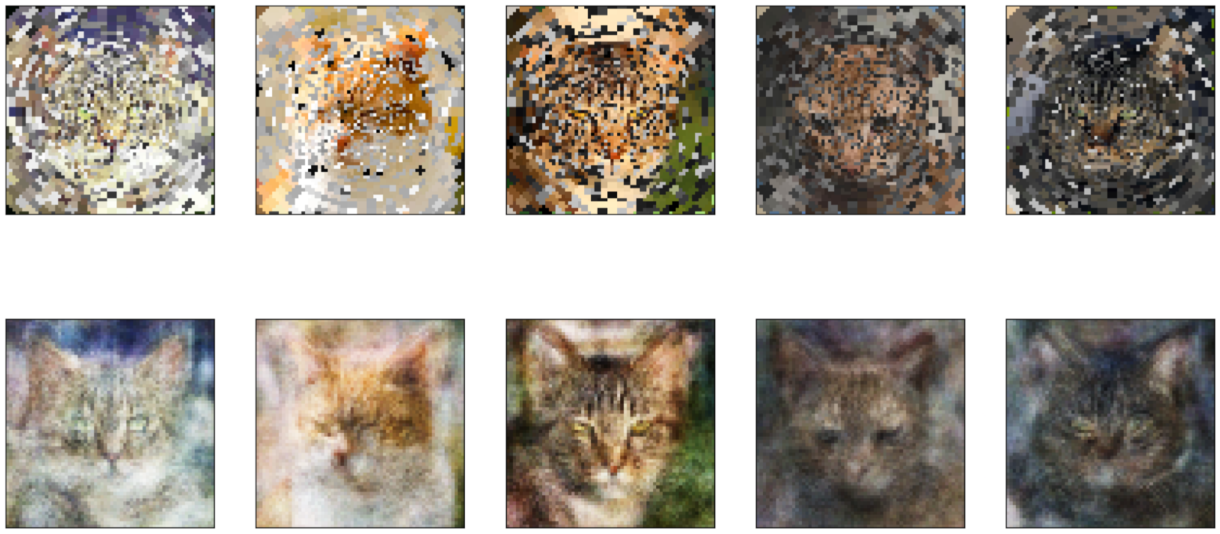


Figure 12) Final Results, Batch Size 128

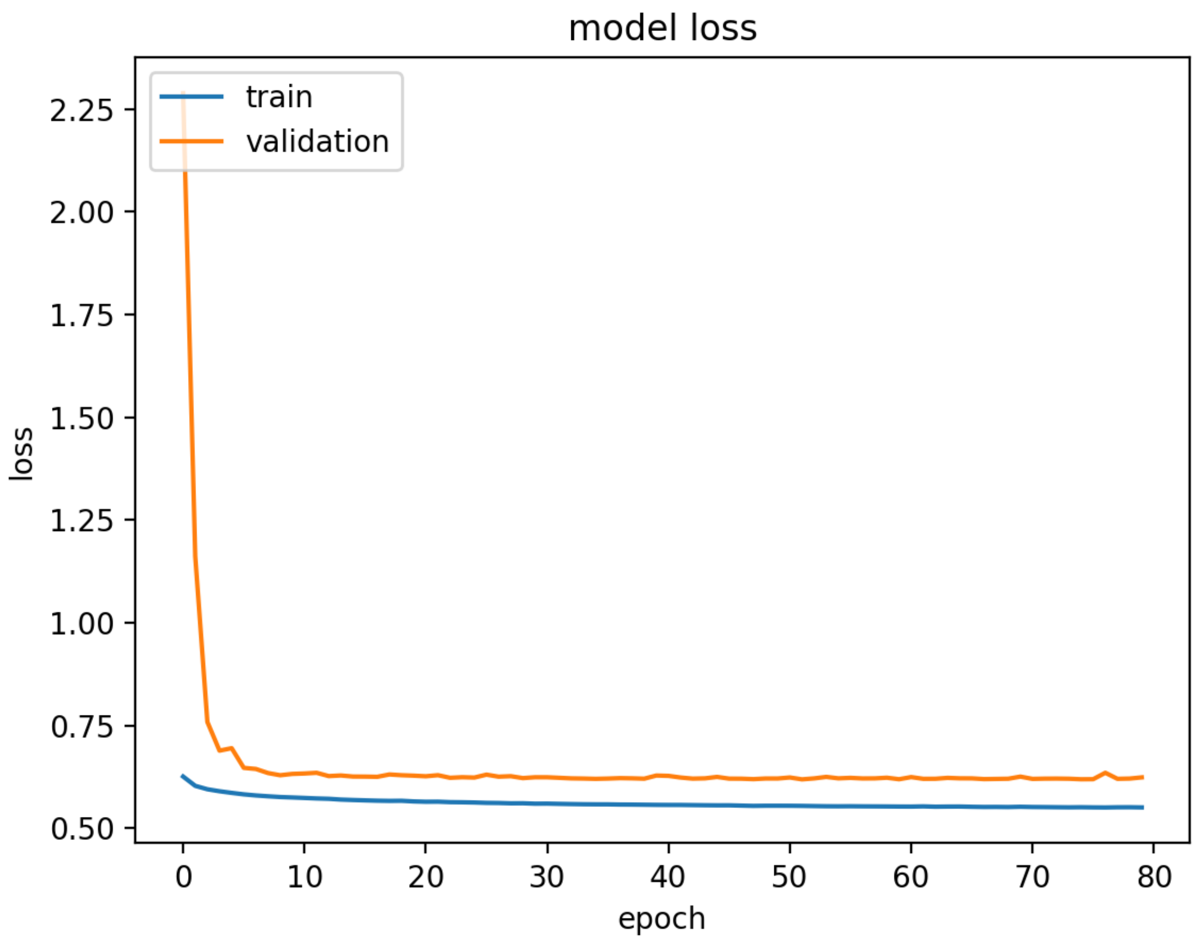


Figure 13) Model Loss Plot

Conclusion

Overall, the results were decent, but we think results could be improved in several ways. First, a larger dataset is always helpful. Second, the use of a Generative Adversarial Network may produce better results, especially when combined with machine learning in-painting techniques.

In terms of Future Work, the model could be combined with an AI Eye Tracking system to render only the portion of the image the user is looking at in full resolution. The following Shadertoy [Meng, https://www.shadertoy.com/view/lsdfWn] is a good representation of the full scope of Foveated Rendering, although it uses a different method than machine learning.

Video Link - Overview of Program

https://youtu.be/sk8KLYn9ua0

Presentation Link:

https://github.com/sme1d1/Foveated_Rendering/blob/main/Foveated%20Rendering%20Presentation.pptx

References

Brownlee Jason. 2019. How to develop a generative adversarial network for an minst handwritten digits from scratch in keras. Am J Vet Res. 62(11):1812–1817.

(Jason Brownlee 2019)

Cheuk Yiu Ip, Adil M¸David L, Amitabh V. 2009. PixelPie: Maximal Poisson-disk Sampling with Rasterization.Nvidia Research. 9(1):49–58.

(Cheuk et al. 2009)

Brownlee Jason. 2019.A Gentle introduction to Generative Adversarial Networks(GANs). Am J Vet Res. 62(11):1812–1817.

(Jason Brownlee 2019)

Brownlee Jason. 2019.How to Train a Progressive Growing GAN in Keras for Synthesizing Faces. Am J Vet Res. 62(11):1812–1817.

(Jason Brownlee 2019)

Anton S. K, Anton S, Thomas L , Mikhail O, Todd G, and Gizem R.DeepFovea: Neural Reconstruction for Foveated Rendering and Video Compression using Learned Statistics of Natural Videos.Facebook Reality Labs

(Anton et al.2019)

Brownlee Jason. 2019.How to Train a Progressive Growing GAN in Keras for Synthesizing Faces. Am J Vet Res. 62(11):1812–1817.

(Jason Brownlee 2019)

Anton S. K, Anton S, Thomas L , Mikhail O, Todd G, and Gizem R.DeepFovea: Neural Reconstruction for Foveated Rendering and Video Compression using Learned Statistics of Natural Videos.Facebook Reality Labs

(Anton et al.2019)

Meng, Xiaoxu. 2018. Kernel Foveated Rendering. University of Maryland, College Park