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.

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. In early development, the team is working with a small created dataset. Going forward, we plan to transition to a dataset titled "Natural Images" which includes nearly 7,000 images of cars, planes, flowers, and more.

First, the images must be transformed to a log-polar coordinate system centered around the middle of the image, as shown in Figure 1. This is currently done using the warpPolar function from the OpenCV Python Library, as shown in Figure 2.

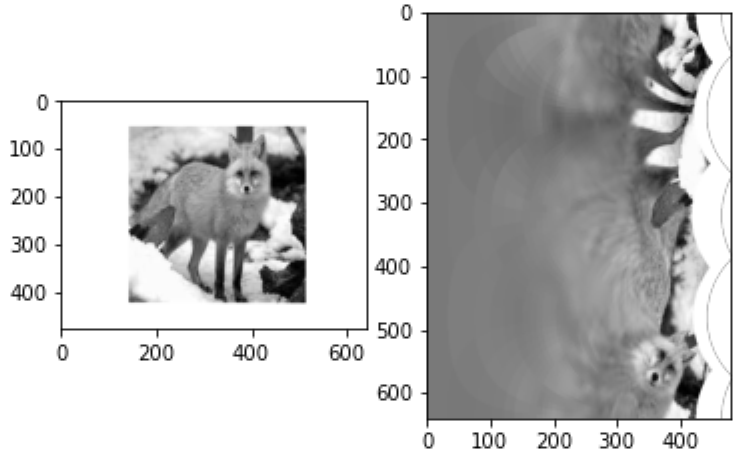
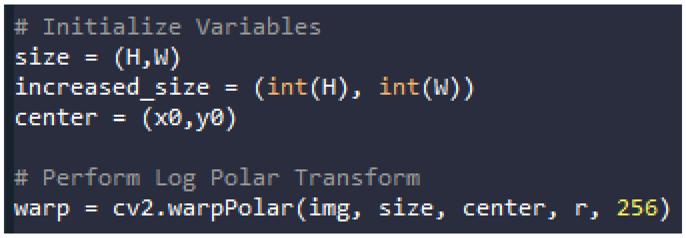


Figure 1) Original vs Log-Polar Transformed Image



```
# Initialize Variables
size = (H,W)
increased_size = (int(H), int(W))
center = (x0,y0)

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

Figure 2) Log Polar Transform Script

Second, a pixel density distribution similar to Figure 3 must be generated to use as input to the model. This is done by generating a distribution of weights ranging from 0 to 1. These weights are used to determine the importance of the pixels at each distance from the center. This pixel density distribution is used as the input to our machine learning model. While Figure 4 shows an inverse exponential distribution, the distribution can be easily modified to achieve better results as the project progresses. The script used to generate this distribution is shown in Figure 5.

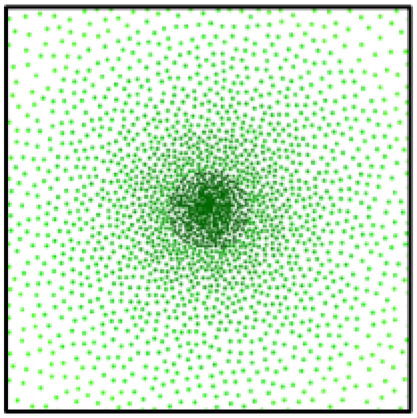


Figure 3) Pixel Density Distribution

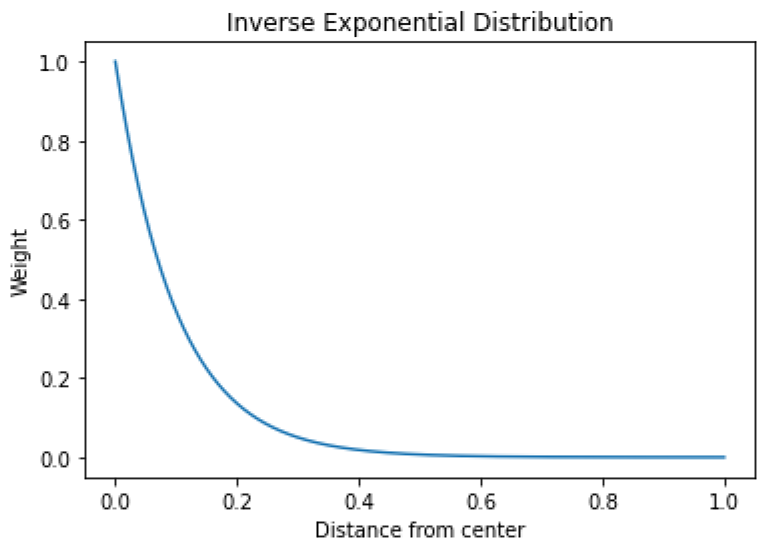
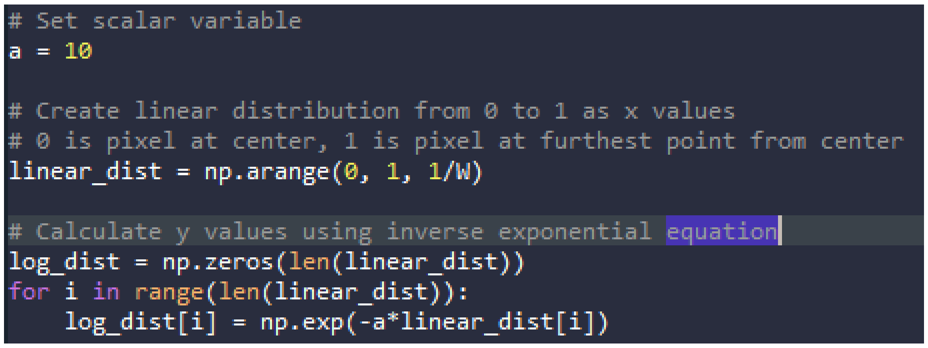


Figure 4) Pixel Weight Distribution



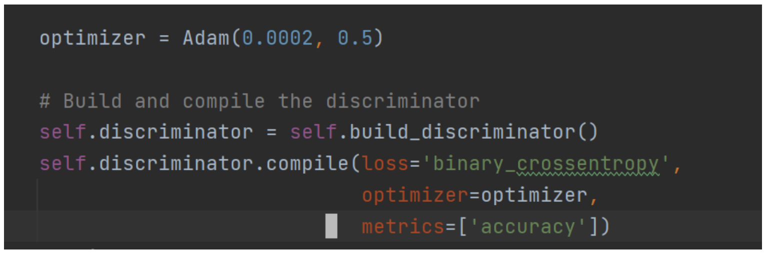
```
# Set scalar variable
a = 10

# Create linear distribution from 0 to 1 as x values
# 0 is pixel at center, 1 is pixel at furthest point from center
linear_dist = np.arange(0, 1, 1/W)

# 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 5) Pixel Weight Distribution Script

After image transformation and pixel sampling, a latent space image will be generated from the sampling and random noise to serve as the initial generator image. Using a binary cross entropy loss function, gives the average difference between predicted and actual probability distributions. It is used for binary classifications, 0 and 1, ie: how our discriminator is evaluating the output of the generator. Adam is used for algorithm optimization, replacing gradient descent. It uses a combination of the Adaptive Gradient Algorithm (AdaGrad) and Root Mean Square Propagation (RMSProp) which gives improvement on problems with sparse gradients and noisy data.



```
optimizer = Adam(0.0002, 0.5)

# Build and compile the discriminator
self.discriminator = self.build_discriminator()
self.discriminator.compile(loss='binary_crossentropy',
                           optimizer=optimizer,
                           metrics=['accuracy'])
```

Figure 6) Optimizer and Loss Function

Our GAN's generator consists of an input layer, three dense layers with Leaky ReLU activation functions, and an output layer with a TanH activation function. Leaky ReLU solves the issue of standard rectified linear activation functions having dead neurons (values that go negative or zero). Instead, Leaky ReLU sends values close to zero that would otherwise be turned to zero with a standard ReLU. (Figure 7)

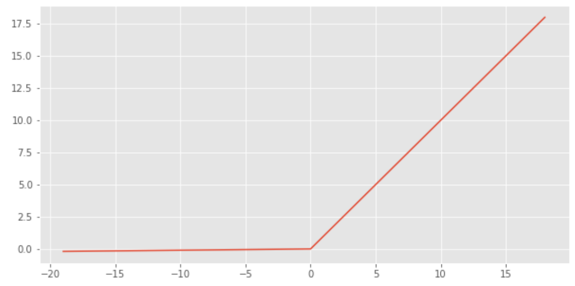


Figure 7) Leaky ReLU Activation Function Graph

The Discriminator consists of 2D convolution layer with LeakyRelu Activation and a 2D max pooling operation, a Dense layer with LeakyRelu, and output. The convolution layer uses a small matrix of weights (the kernel) and performs piecewise multiplication on the 2D input data (our image pixels array). This allows the generation of a smaller parameter set instead of needing to have 256x256 (our image size) feature weights. The pooling layer calculates the largest value in each patch of each feature map. These results are then down sampled, highlighting the most present feature in each patch. This provides a way of down sampling feature maps by giving a summary of the average features present.

Future Work

In the current state of the project, the team has a set of weights to apply to each pixel but the code to actually apply the weights and generate the density distribution of a specific image has not been created. The team's GAN structure is also derived from published examples based on noise for image generation. We are working to improve our knowledge of GANs to refine & rewrite code. A method for saving our trained model and evaluating performance is also necessary.

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