Predicting Eye Gaze for better FPS

CV project connecting Gaze Capture with Foveated Rendering

Team:

Kannav Mehta 2019101044
Gurkirat Singh 2019101069
Shrey Gupta 2019101058

The Problem

What are we solving ?

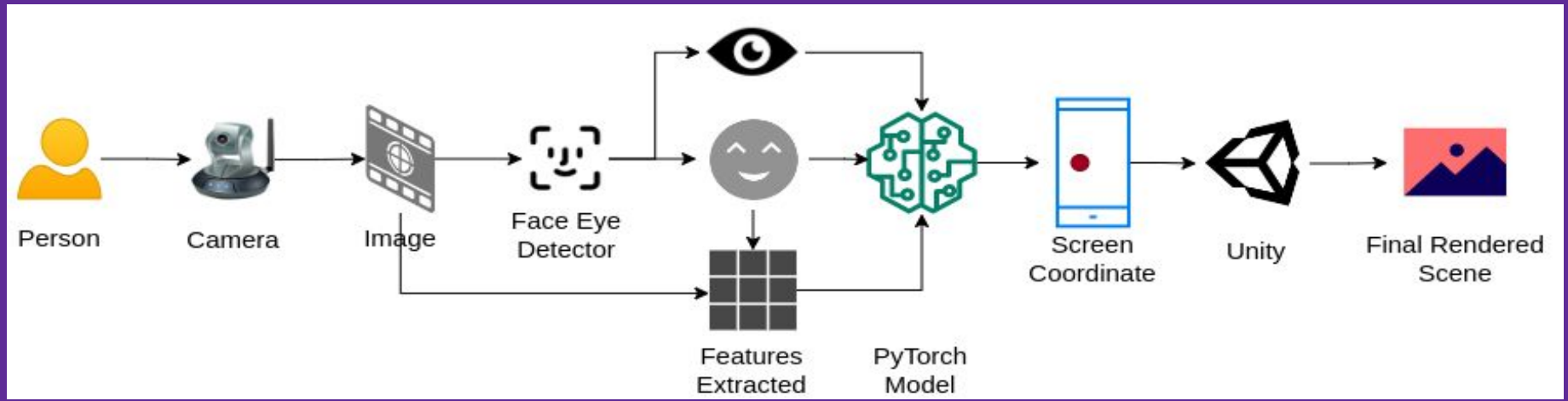
Today’s games and complex 3D scenes have pushed the current hardware to its limits. If you want to enjoy a 60FPS 4K gameplay of any modern AAA game, you need a top of the line GPU and CPU which are out of reach of many users for many reasons, hence depriving them of majority of the users to experience the latest technology.

But if we look closely, we don’t need high res for all parts of the screen, we only need for the part where the user is looking at, isn’t it?

So why don't we just render the scene of current focus at max resolution but all the other parts at low resolution, hence saving a tons of resources and compute.
We solve this problem in 2 parts:

- Detecting GazePoint using Deep Learning
- Rendering the scene around the gazed point at highest resolution (Foveated Rendering)

Proposed Solution



Detecting Gaze with Deep Learning

Dataset

GazeCapture Dataset

GazeCapture is an eye tracking dataset with data of nearly 2.5 million images of over 1400 participants gathered via crowdsourcing. The dataset has: RGB frame, Face / eye bounding boxes and face grid (a 25x25 occupancy grid for face overlaid over image) with corresponding x,y coordinate of the gazed point w.r.t camera

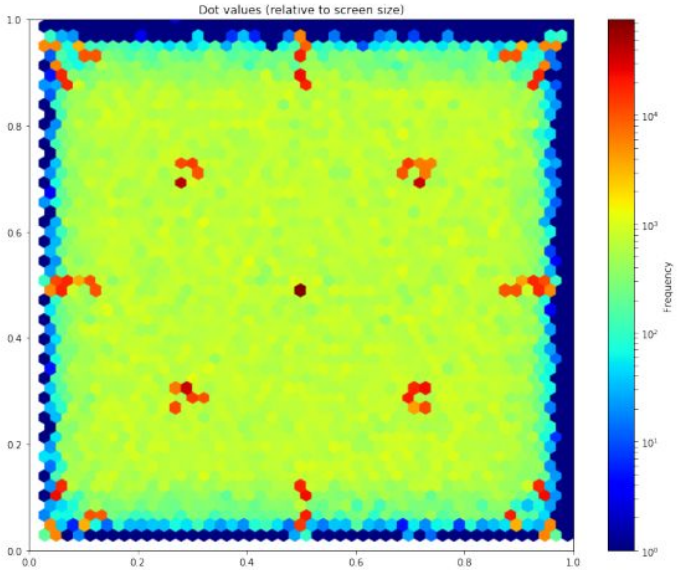
Before going in deep into deep learning models, we wanted to do some exploratory data analysis (EDA).

We found out the distribution of the data points was quite uniform except the special 13 calibration points.

GazeCapture Dataset (contd)

Mean, max and min of x,y

	X	Y
Mean	294.22	260.50
Max	40	40
Min	1326	1326



Distribution of points on the screen

Data Preprocessing

Approach 1: Viola Jones

The initial stage in our pipeline is to detect eyes and faces of a person from an image at a reasonably fast rate so that our python code does not start to bottleneck the pipeline. For this, we mainly use the Viola Jones algorithm as mentioned in the class.

It gives us reasonably well performance and is able to detect our eyes and faces in majority of the cases with false negatives mainly when the eyes were not clearly visible or in inappropriate lighting conditions, a downside which we mainly ignore due to its speed and accuracy in other cases. Using the eye and faces detected we create the 25 x 25 face grid as well and hence we generate all 4 features necessary for our model

Approach 2: Face Landmarks

Though **Viola Jones**, was incredibly fast and lightweight, the cases in which it was failing were quite substantial as when a user looks at extremes of the screen (generally at the bottom as eyes almost close), or tilts the head, Viola Jones fails to detect eyes. This caused the results to remain constant and not update in such extremities.

To overcome this, we also tried using the **DLib face landmarks** detector. On the basis of the coordinates of the detected landmarks we manually created bounding boxes for the eyes and the face. This detector, though slower than Viola Jones due to its inherent complexity performs much better and thus we are were able to detect the necessary features even when the user was looking at the extreme points even in varying ambient conditions. It also work well when the person was not in the entire frame.

However, which detector to finally chose depends on the exact priority of speed vs performance

Calibration

Need of calibration

GazeCapture dataset has images from iPhone and iPads so the x, y given are for a iPhone or an iPad.

Therefore we need to calibrate the model in order for it to give results for a custom camera, position of user and screen size.

Using Homography matrix

As the output is for a iPhone / iPad, transforming it to a laptop / desktop screen is just a plane to plane transform . Hence we try to find a **homography** to calibrate the model.

At system startup, a user needs to look at pre decided 8 positions on the screen and we capture the model output for those. For each point we took 5 images to overcome noise in the model. We compute the average for each point to get the actual model output Now we have 8 correspondences between model output and expected output, we fit a homography into it using RANSAC for robustness.

The results of this approach were not good. It was very very noisy and hence was not suitable enough for further pipeline.

Using Support Vector Regression

In **SVR** we did not consider camera's internal parameters will also differ and hence the transform will not be a direct Homography transform, but some other **non-linear** transformation.

To find the non-linear transformation we the approach. As SVR we had to train SVR for each user, we increased the number of calibration coordinates to **13**(the 13 calibration points mentioned in the dataset) after experimenting with less number of coordinates as well. The number of retries per coordinate remained at 5 to account for noise.

We used 2 separate SVRs for the x and y coordinates. The output coordinates were normalized and then used for training, however the input of the SVR came from the last layer of the model and hence was not normalized on the intuition that higher weight to a feature might be something the model would have learned. We experiment with different kernels for the SVR and found out that the **'linear' kernel** performs the best

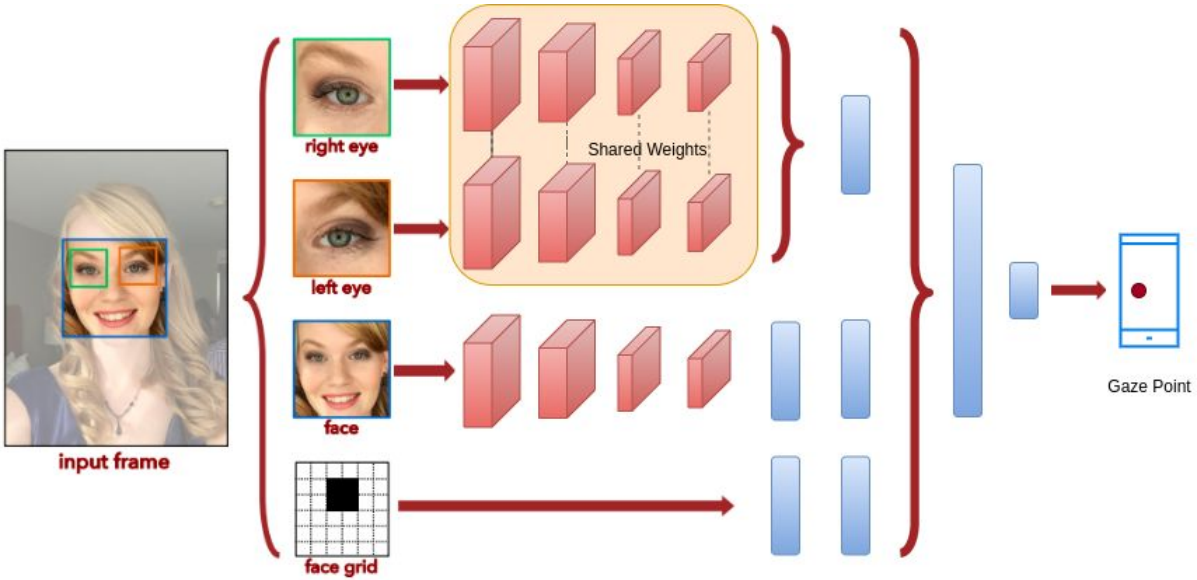
Network Architecture

Metrics

The main metric used for judging model performance was the MSE loss of the ground truth screen coordinate and the predicted coordinate from our trained model.

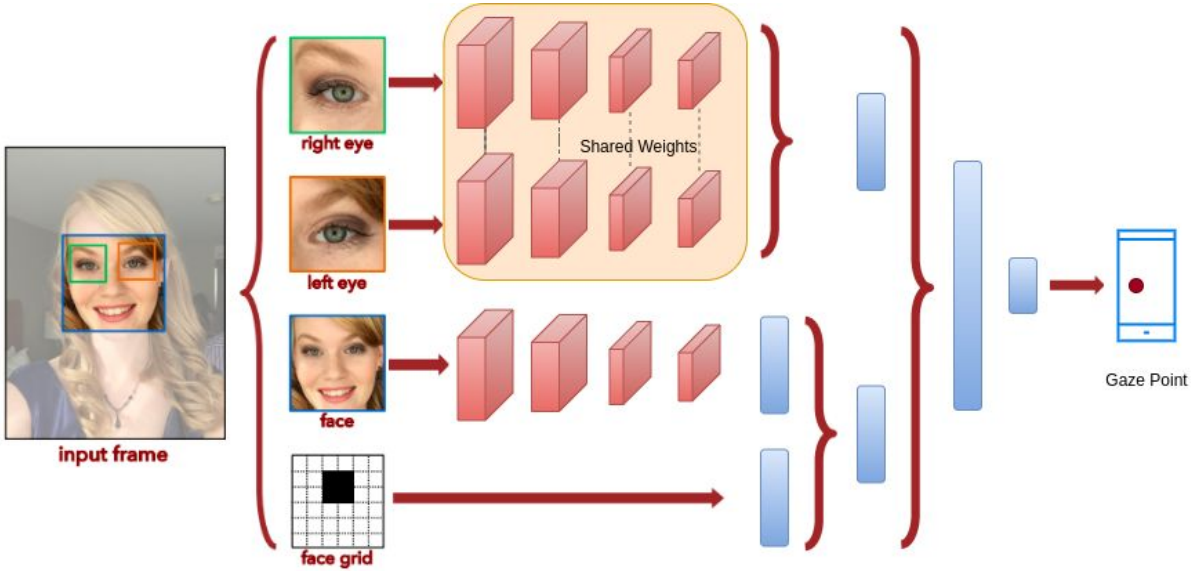
For our understanding and visualization we also calculated the euclidean distance between the ground truth and the prediction.

iTracker



This is the original model as proposed by the paper and thus we directly use the same model architecture for our purpose. It consists for 4 input features: left, right eye, the face and a 25x25 faze grid highlighting the position of the face in the image. The eyes are passed through a common convolutional layer and the end feature representations of the both the eyes, face and face grid are passed through the fully connected layers to get the final gaze point prediction

iTracker v2.0



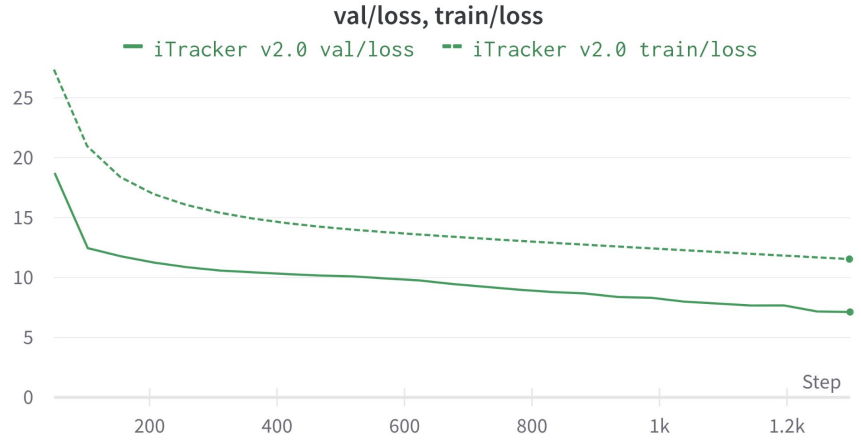
The main essence of this model is the same as its previous version. However, we believed that the face and the face grid are more closely related in comparison to the eyes so they should be first passed through a new fully connected layer to possibly extract more important features related to face orientation. The resultant face orientation and the eye orientation features are again combined and passed through fully connected layer to get the final predicted screen point

Training

We trained both the model (iTracker v1.0 and v2.0) on GazeCapture

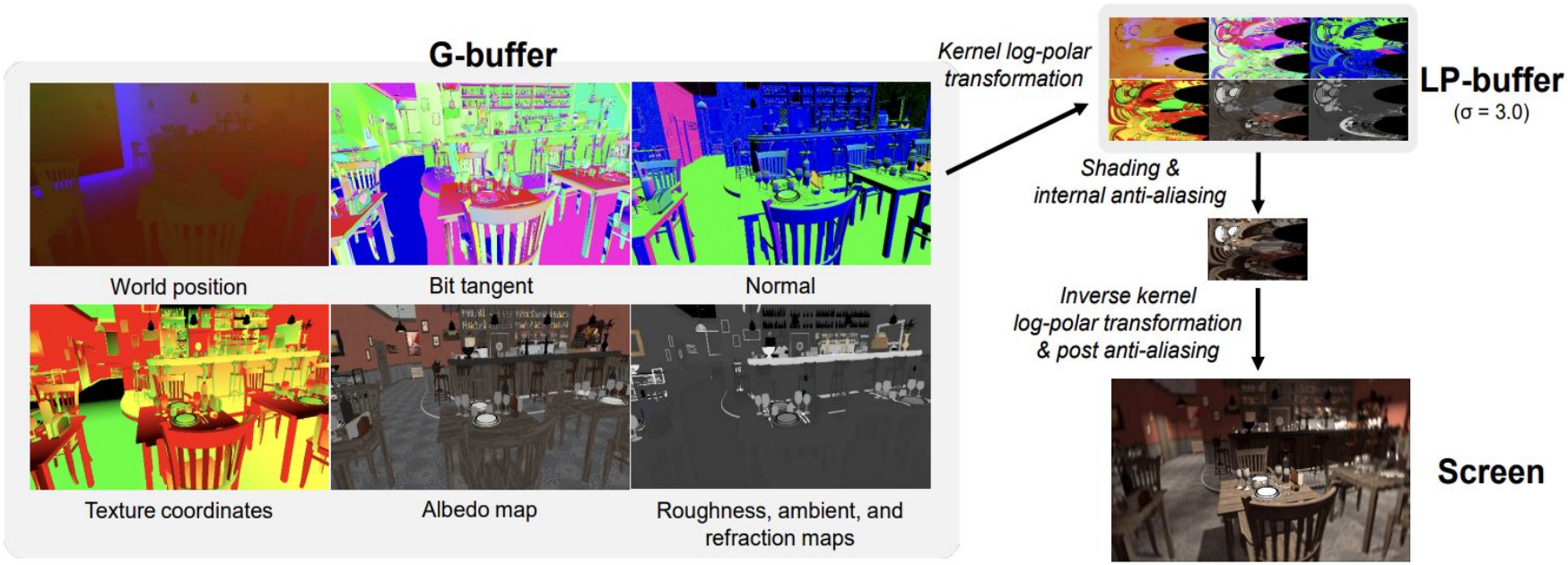
- Due to resources constraints, we were not able to train on full dataset, so we trained on a subsampled dataset (1/10th). We subsampled from each GazeCapture sequence uniformly so as not to disturb the underlying sequence.
- iTracker v1.0 took about 1 day to converge and iTracker v2.0 also took a similar time.
- Graphs for both training have been given bellow.
- iTracker v2.0 performed slightly better (.5 cm less loss). Reason for this being due to lack of variety in the dataset, both the models had learned all the available information

Training losses



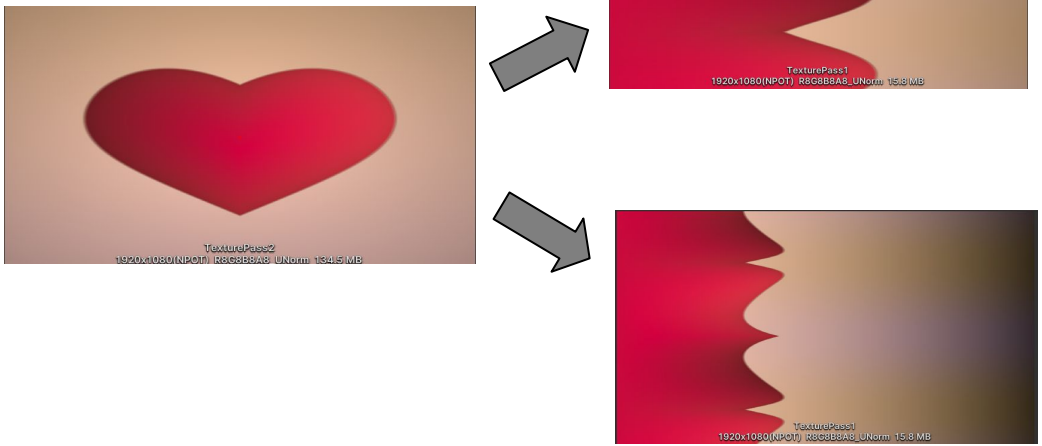
Foveated Rendering

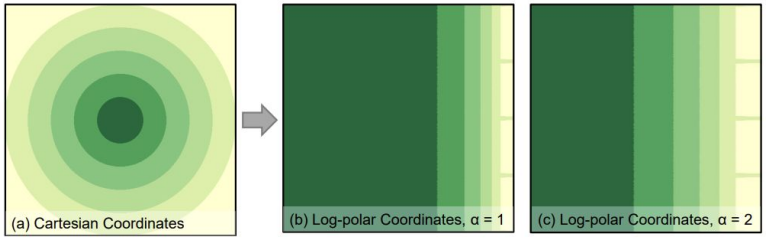
Foveated Rendering Overview



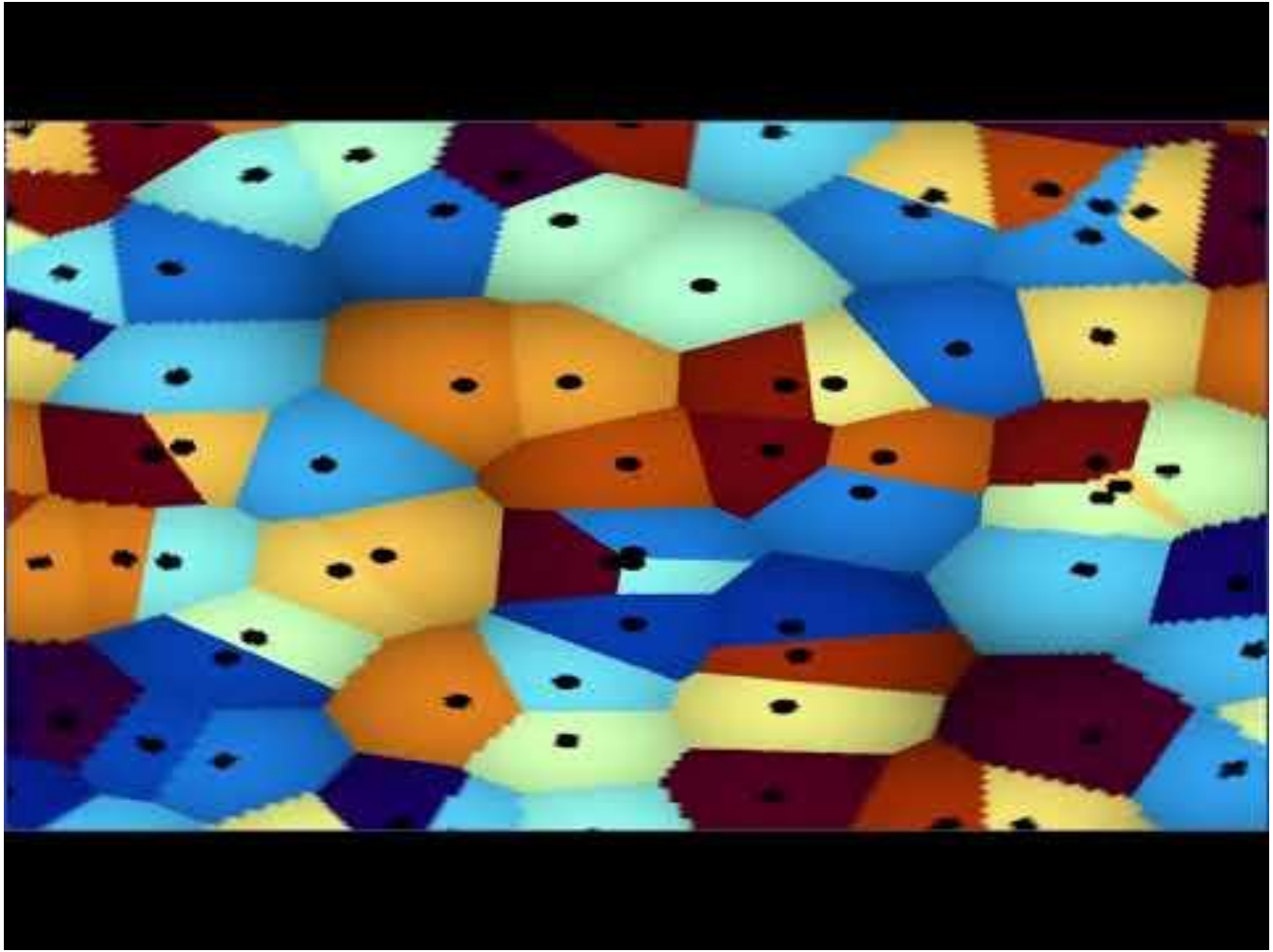
Foveated Rendering

We first transform the image from cartesian coordinates to log-polar coordinates. For a 3D scene this can be thought of as converting the render-texture to log polar coordinates. The heart figure you see on the right is a GPU only scene and the following images are the dumped texture after converting to log polar coordinates. We can clearly see the heart now takes up much more area of the space and the background is sampled very less. For a 3D scene the shader would be called on the center pixels the same but for the area outside the fovea the shader calls will be reduced by a lot, thus saving us a lot of FPS and giving us higher fidelity. This is followed by a second pass that converts these back to cartesian coordinates to be rendered on the screen.





Final Demo



References and other Implementations

- Gazecapture: https://arxiv.org/pdf/1606.05814.pdf
- MPIIGaze: https://arxiv.org/pdf/1711.09017.pdf
- GOO Gaze: link
- https://github.com/repalash/CinematicFoveatedUnity
- https://github.com/slipfre/FoveatedRenderingLab

Thanks

Link to code:
Foveated Rendering: link
Gaze Detection: link

Work Distribution: All members contributed equally to all components of the project.