Egocentric Hand Tracking for Virtual Reality

Hershed Tilak and Royce Cheng-Yue
Stanford EE 267: Virtual Reality
Instructors: Gordon Wetzstein and Robert Konrad

Abstract

Among the contemporary VR problems, creating an immersive and realistic environment is one of the most challenging tasks. In this respect, having the ability to touch and interact with the VR environment is extremely important. Through this project, we created an egocentric hand tracking system that could be embedded inside a VR head mounted display. The pipeline involves reading RGB-D images from an Intel RealSense, piping the RGB components to YOLO, a CNN model, for bounding boxes, retrieving the corresponding depths at these bounding boxes, and displaying the resulting rendered hands in VR. For the hand detection portion of the pipeline, we trained three YOLO models with the best model achieving a test mAP of 64.6%, which is on par with the YOLO results on the VOC 2012 dataset. With more training, we should be able to achieve better accuracies with the other dropout and jitter models. We showcase the results of this pipeline in a video presented in Section 4.2.

1 Introduction

Interest in the field of virtual reality (VR) has grown drastically in recent years as commercial head mounted displays have become more viable. Research into VR shows many promising applications in fields ranging from medical [11] to industrial manufacturing [7]. Many of these applications require inferring interactions between a user and his or her environment [10].

One possible method for realizing these interactions is through the use of user hand tracking. Many of the proposed hand tracking implementations for VR require accessories external to the headset, such as a colored glove worn by the user [15] or an external, user-facing sensor such as a Microsoft Kinect [2]. In this report, we explore an egocentric hand tracking approach, which can ideally be embedded into a VR head mounted display.

Egocentric hand tracking has a number of challenges when compared to front-facing hand tracking. Firstly, tracking is less reliable since the egocentric viewpoint causes hands to frequently move outside of the camera view frustum [10]. Secondly, fingers are often occluded by the hand, as well as any objects being manipulated, in egocentric views. After attempting to model the hand through joints with an object detection model, we realize that these models perform poorly for hand joints due to small bounding boxes and occlusions. As a result, we simplify this problem to only detect hand positions.

Our input to the pipeline is RGB-D images. The output is the rendered 3D hands in Unity that are used to interact with a virtual environment.

2 Related Work

Object detection has been a popular problem in the past few decades. HOG-based SVMs [1] and deformable parts models (DPMs) [8] used to be the state of the art. In particular, HOG-based DPMs achieved 33.7% mAP on the VOC 2007 dataset [3].

After 2013, convolutional neural networks (CNNs) became the standard for object detection. Some examples of these CNNs include R-CNN [4], which involves a region proposal step and a regression step within each region proposal. R-CNNs proved to be particularly slow to train and test. Specifically, R-CNNs take about 47 seconds for a single pass of object detection. Fast R-CNNs [9] and Faster R-CNNs [14] mitigate the initial region proposal selective search process [13] by introducing a CNN. In the end, Faster R-CNN is able to achieve 73.2% mAP with 142 ms latency on the VOC 2007 and 2012 dataset.

YOLO [5] treats the object detection problem differently. Instead of having a two-step process, YOLO regresses on the entire image in a single forward pass. This allows YOLO to achieve a latency of 22 ms; however, the accuracy reduces to 63.4% mAP.

3 Approach / Method

Figure 1 shows our pipeline for egocentric hand tracking. In general, the Intel RealSense provides RGB-D images. The RGB component goes into a CNN to detect bounding boxes and the x, y positions of the left and right hands. After we retrieve the z position from the depth images corresponding to the bounding boxes, we send them to a server in Unity which accepts the incoming x, y, z hand positions to be rendered. Unity updates the left and right hand positions based on the outputted x, y, z values and detects object collisions in a VR game.

Our proposed system consists of four major components: Image Acquisition, Hand Detection, Data Transportation, and Virtual Environment Rendering. We now discuss each of these components in more detail.

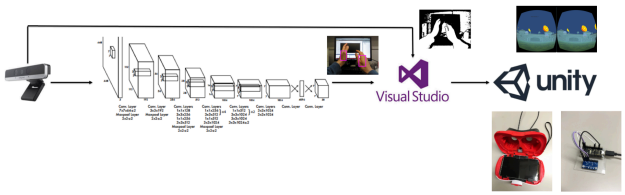

Figure 1: *Pipeline for Egocentric VR hand tracking.*

3.1 Image Acquisition

We use an Intel RealSense camera mounted on top of the VR headset built in class to acquire RGB and depth images. A picture of the hardware setup is included in Figure 2. These images are read into a C# program. The RGB image is passed to a CNN, which processes it and returns the bounding boxes of a user's left and right hands. The C# program then uses the depth image to extract the associated depth information at the hand centers. At this point we have the x, y, and depth information for the predicted hand centers, stored in meters. This information is passed to the Unity Engine via our Data Transportation mechanism.

Figure 2: *Hardware Setup for VR Headset*

3.2 Hand Detection

3.2.1 Dataset

To train our CNN, we need to gather a dataset of egocentric hands. We looked into several hand datasets, but many of them were comprised of front facing hand positions. We ended up combining the EgoHands dataset [12] and the General-HANDS dataset [6]. The EgoHands dataset is normally used for complex egocentric interactions with another human; as a result, many of the images had four pairs of hands. This bias is not desired as the model could expect an extra pair of hands in order to predict a correct egocentric hand. The General-HANDS dataset is normally used for joint detection; however, we transformed the dataset into a hands detection dataset by taking the joints with the minimum and maximum x and y positions. In the end, we have 2,829 training images, 403 validation images, and 807 test images. Figure 3 shows examples from the EgoHands and General-HANDS dataset respectively.

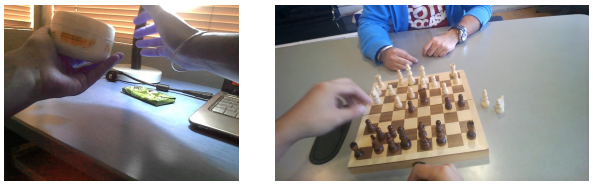

Figure 3: *Sample Images from the EgoHands and General-HANDS Dataset.*

3.2.2 Technical Approach

There are multiple methods for hand detection; however, we decided to use YOLO as our framework because it offers realtime latency while achieving high accuracy.

3.2.3 Original YOLO Architecture

Instead of the two-step process proposed by R-CNN, YOLO directly regresses on the entire image. Specifically, YOLO splits the image into an S x S grid. Each cell within this grid then regresses on B bounding box coordinates and confidence scores, as well as C class probabilities. In the end, each image will produce S x S x (B x 5 + C) outputs. Overall, the architecture consists of 24 convolutional layers, 2 fully connected layers, and a final output layer. Figure 4 shows the original YOLO architecture.

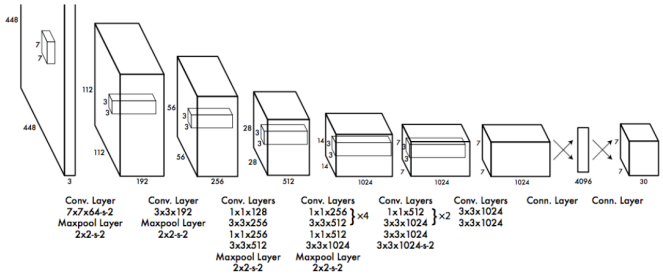

Figure 4: *Original YOLO Architecture.*

3.2.4 Our YOLO Architecture

To fit YOLO to our problem, we needed to make a few changes. Because the original YOLO was pre-trained on ImageNet, we had to replace the last fully connected layer with a hand detection layer. In particular, since we are only looking for hands, we set C = 1. We also set S = 7, which splits each input image into a 7 x 7 grid, and B = 2, regressing on 2 bounding box coordinates and confidence scores within each cell. In total, each image in our pipeline outputs S x S x (B x 5 + C) = 539 outputs. We also added a downsampling layer before the two fully connected layers so that our YOLO architecture could fit inside our GPU memory. This change reduces our total number of parameters by 84%. Our final YOLO architecture is presented in Figure 5.

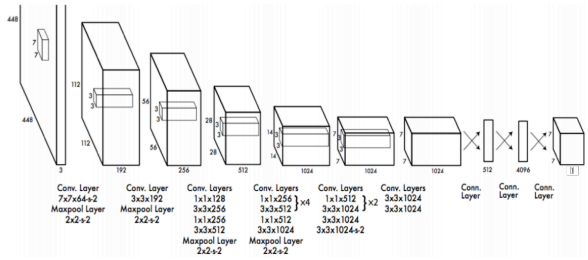

Figure 5: *Our YOLO Architecture.*

3.2.5 Loss Function

Because YOLO directly regresses on the entire image, its loss function captures both the bounding box locations, as well as the classification of the objects. The loss function is specified in Figure 6.

The loss function is split into five parts:

(1) Loss according to the bounding box x and y centers
(2) Loss according to the square root of the width and height of the bounding boxes
(3) Penalization of predicted objects
(4) Penalization of unpredicted objects
(5) Penalization in the difference of class probabilities

Some interesting observations are that we use the square root of the width and height for the loss function to take care of differences in bounding box sizes. For example, errors for smaller bounding boxes incur a higher penalty than those for bigger bounding boxes. λ_{coord} is a scaling factor on the bounding box coordinates to ensure bounding box penalties and class probability penalties contribute equally to the loss. λ_{noobj} is a scaling factor to penalize object identification when there is no object. The default YOLO configuration assigns λ_{coord} to be 5 and λ_{coord} to be 0.5.

$$Loss = \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} (x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \quad (1)$$

$$+\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} \left(\sqrt{w_i} - \sqrt{\hat{w}_i}\right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i}\right)^2 \quad (2)$$

$$+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} \left(C_i - \hat{C}_i\right)^2 \quad (3)$$

$$+\lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{noobj} \left(C_i - \hat{C}_i\right)^2 \quad (4)$$

$$+ \sum_{i=0}^{S^2} 1_{i}^{obj} \sum_{c \in classes} \left(p_i(c) - \hat{p}_i(c)\right)^2 \quad (5)$$

Figure 6: YOLO *Loss Function*.

3.2.5 Training Configuration

We found that GPU training is significantly faster than CPU training. As a result, we utilized the Stanford Rye cluster, which has 6 NVIDIA Tesla C2070 GPUs. Although each of these GPUs have 6 GB memory, they are shared among Stanford students. We trained our model in batches of 64 images.

3.2.6 Hand Detection Models

We initially trained a single model using the configuration presented in Section 3.2.3. We quickly realized that this model started to overfit, so we added dropout in our next models. We also added random crop jittering, which augments our dataset by a factor of 2, as well as changing the S parameter in hopes of training a better hand detection model. In total, we have three models: S=7, S=7 with Dropout=0.5 and Jitter=0.2, and S=11 with Dropout=0.5 and Jitter=0.2. The evaluations and error analyses are discussed in Section 4.

3.3 Data Transportation

Data transport to the Unity Engine is accomplished through two scripts. On the Unity side, we implemented a C# script consisting of two threads. One of the threads acts as a server which listens on port 9999 of the local host for incoming connections. When a connection is established, the script reads in the data and parses it to extract the x, y, and depth information. It also parses whether this position information is for the left or right hand. This information is then stored in a shared data structure, and a flag is set to indicate that the structure has been updated. The main UI thread continually checks this flag, and when it sees that the values have been upated, it uses the new values from the data structure to update the hand representations in the virtual environment. This multithreaded approach is necessary because Unity game objects cannot be updated outside of the main UI thread, and implementing the server on the main UI thread is unfeasible since listening for incoming connections is a blocking operation.

The second script handles the interaction between the neural network and the C# server. It establishes a new connection and then continually sends the predicted hand center information that is output by the neural network for each new frame.

3.4 Virtual Environment Rendering

The virtual environment is rendered inside of the Unity Engine and was designed to showcase our hand tracking system. The environment immerses the user in a medieval courtyard surrounded by stone gateways on all sides. Four turrets are evenly distributed across one of these gateways. Each of these turrets is associated with a unique fruit. The turrets randomly fire their fruit projectiles at the user. Figure 7 shows a downward facing view of our virtual enviornment.

The user's job is to use his or her hands to block the incoming fruits. On a collision between a virtual hand and a virtual fruit, the virtual fruit is split in half and sent flying in the other direction. A counter keeps track of the number of fruits that were blocked during a game session.

New hand information is received by the Unity Engine in the form of an x, y, and z distance, all in meters, from the center of the RealSense camera. Conveniently, distances in the virtual environment are already in meters by default. The associated position of the virtual hands can easily be established by traveling these same distances from the center of the camera object in the virtual world.

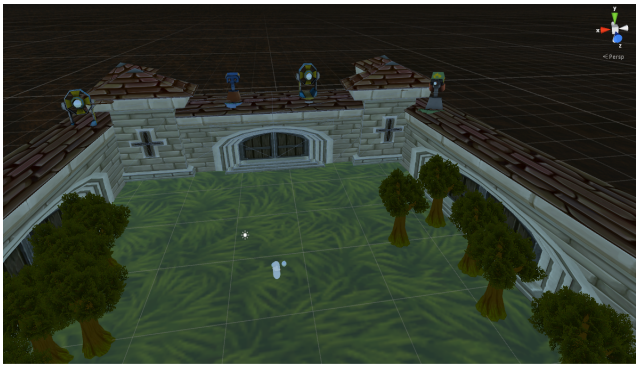

Figure 7: *Rendering of Our Virtual Environment*.

4 Evaluation

4.1 Hand Detection

After training our model, we evaluated our model using mAP as our metric. We compared three different models and their performances based on several hyperparameter configurations.

4.1.1 Metrics

To evaluate our model, we use interpolated mean average precision mAP from VOC 2007. This metric has become a standard for the VOC dataset. In general, this metric models the shape of the precision and recall curve as a step function. For every recall level between 0 and 1, we interpolate the precision by finding the maximum precision across all recall levels larger than the current recall level. We then find the average of all of these precisions to find average precision (AP). The equations for AP is defined in Figure 8.

A caveat is that this metric is slightly different from other mAP metrics, in which they do not interpolate the precisions at recall levels. Because we are using YOLO and want to compare our

mAP with the YOLO mAP results on VOC 2012, we will use the metric defined in VOC 2007.

In order to determine true and false positive bounding boxes, we find their intersection over union (IOU) with the ground truth bounding boxes. In particular, IOU is defined as the intersection of the predicted bounding box and the ground truth bounding box divided by their union. We consider predicted bounding boxes to be correct if they have IOU > 0.5, which is a standard among object detection papers. The equation for IOU is defined in Figure 8.

$$p_{interp}(r) = \max_{\tilde{r}:\tilde{r}\geq r} p(\tilde{r})$$

$$AP = \frac{1}{11} \sum_{r\in\{0,0.1,\ldots,1\}} p_{interp}(r) \qquad a_0 = \frac{area(B_p \cap B_{gt})}{area(B_p \cup B_{gt})}$$

Figure 8: *Equations for Average Precision (AP) and Intersection Over Union (IOU).*

4.1.2 Models

We trained a total of three YOLO models for hand detection. For our first model, we set S=7, B=2, and C=1. As a result, each image produces 539 regression outputs. After training for about 200 epochs, the training mAP for this model is 73.3% mAP and the test mAP is 64.6% mAP. This accuracy is on par with the results of YOLO on the VOC 2012 dataset. From this model, we notice that there is an 8.7% gap between the training and test accuracies, indicating this model overfits to the training data. Specific details of the training over time are captured in Figure 9.

To alleviate this problem, we looked into adding dropout and jittering to the model. Dropout is a form of regularization by randomly dropping neurons during training, which hopefully allows each individual neuron to be less sensitive to the other neurons. In turn, this reduces overfitting for neural networks. We decided to choose a dropout of 0.5, randomly dropping half the neurons during training. Jittering randomly crops input images using x and y translations, which augments our dataset by a factor of two. We chose a jittering parameter of 0.2, which translates the image vertically and horizontally by 20%.

With these changes, we introduce two more models. The first model maintains the original S=7 parameter with the specified dropout and jittering parameters. The second model changes S to be 11 in hopes of capturing more granular bounding boxes since YOLO is only able to capture one class per cell within the S x S grid.

After about 200 epochs, we find that the first model has a training mAP of 52.1% and a test mAP of 50.0%. The second model has a training mAP of 39% and a test mAP of 37.3%. Specific details of the mAP for these two models are in Figure 10 and Figure 11.

We see that gap between the training mAP and test mAP is small relative to the original model, which means that the model is not overfitting to the training data anymore. Although the original model outperforms these two models after 200 epochs, we believe the dropout and jitter models should outperform the first model with more training. Specifically, the original model loss plateaus after 200 epochs, but the dropout and jitter models are still improving. Because dropout causes neurons to be exposed to less data, more training is required to reach the same levels of accuracy. We also notice that the S=7 model performs significantly better than the S=11 model. One possibility is that the S=7 grid is sufficient enough to capture the hand bounding boxes. With the S=11 model, we find that each image produces 1331 outputs, which is double that of the S=7 model. As a result, we are able to capture more granular bounding boxes; however, this granularity comes at the expense of producing potentially erroneous bounding boxes.

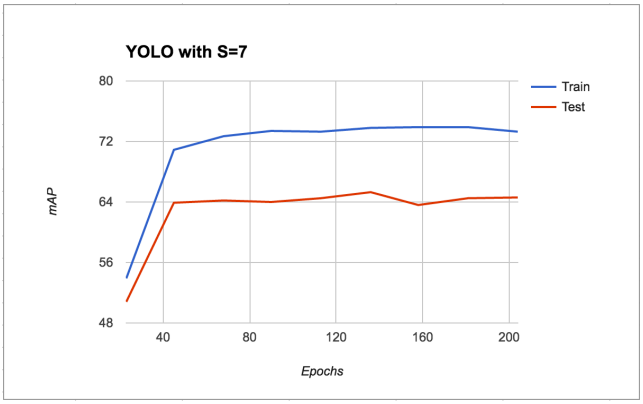

Figure 9: *Training and Test mAP for S=7 Model.*

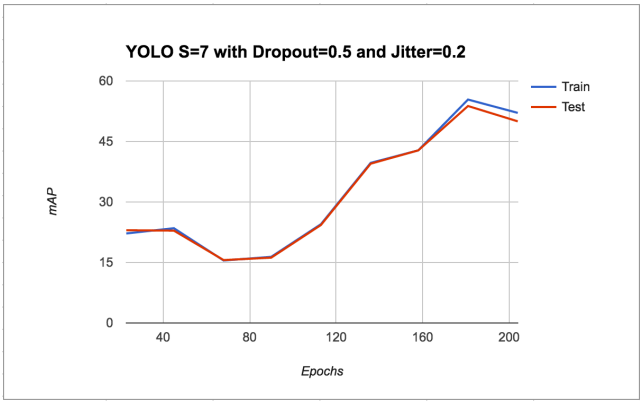

Figure 10: *Training and Test mAP for S=7 Model with Dropout=0.5 and Jitter=0.2.*

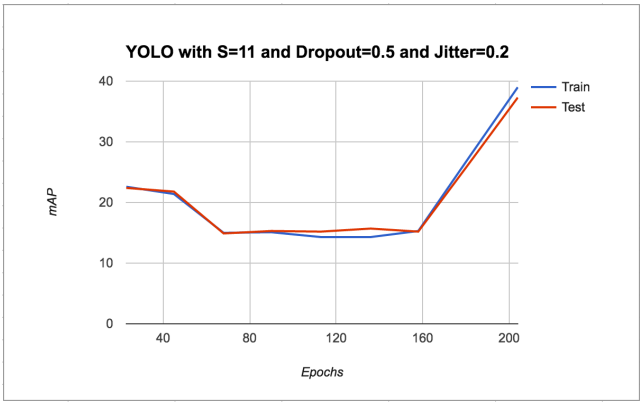

Figure 11: *Training and Test mAP for S=11 Model with Dropout=0.5 and Jitter=0.2.*

4.2 End to End Pipeline

We were successfully able to construct and use the entire pipeline. Unfortunately, we were unable to have a realtime configuration because we were unable to port YOLO over to Windows. Instead, we performed a three step process:

(1) Save the RGB-D images from the Intel RealSense in Windows.

(2) Pipe the RGB component to YOLO in CPU mode on our Macbook Pro Retina laptop to retrieve the bounding boxes.

(3) Replay the bounding boxes with the provided depth images to Unity on Windows.

As a means to showcase our pipeline, we created a demo video that shows x-axis translations, z-axis translations, and a combination of translations in our VR game. The main screen is the virtual environment and the upper right part of the screen contains the outputted bounding boxes from YOLO. The demo video is located at https://youtu.be/w3eN461QXks. Figure 12 shows an example frame within the demo video.

Overall, the system works well and is able to track the hands for most frames. There were a few frames with two hands and our pipeline is able to recognize both hands within the virtual environment. Sometimes, YOLO would produce some false positives. For example, YOLO would occasionally output a bounding box for a shadow from the nearby printer. This incorrect bounding box is most likely due to the color of the printer and shape of the shadow. Naturally, there is more that can be done to improve YOLO and, with more training, the dropout and jitter models should outperform the original model.

Figure 12: *Example Demo Video Showcasing the Predicted Bounding Boxes from YOLO for the Hands and the Corresponding VR Hands in the Virtual Environment.*

5 Conclusion and Future Work

In the end, we were able to provide an end-to-end system to detect and track hands in virtual reality. However, because we were unable to run YOLO on Windows, we could not determine the actual latency of the system with the CNN model. Because YOLO computes a forward pass in about 20 ms, we suspect that the real-time version of this system would have low latency, possibly in the hundreds of milliseconds.

In general, we had many issues in setting up this pipeline. One of the issues is that deep neural networks require a large amount of GPU memory. In particular, YOLO requires 2 GB of GPU memory while Faster R-CNN requires 4 GB. Most of these networks also only run on Linux but our pipeline only ran on Windows. After spending countless hours trying to port YOLO over to Windows, as well as trying to set up other models, such as Faster R-CNN, on Windows, we were unable to produce a real-time pipeline. Another requirement was that we needed to use CUDA for our model, which was unavailable for AMD GPUs. One possible way to get around all of these issues is to use a Linux computer with a large NVIDIA GPU; unfortunately, we did not have the resources to attain such a computer.

For hand detection, YOLO does a decent job in predicted hand bounding boxes, achieving a test mAP of 64.6% after 200 epochs. There are a few cases in which YOLO has trouble with false positives. Many of these involve objects that have the same color and shape as hands. With more training, the other two models should be able to outperform this model, providing a better hand detection system.

One possible method to drastically improve the hand detection system is to use temporal information. Specifically, the current hand position frame is dependent on the previous hand position frame. As a result, we could use features extracted from YOLO and pass these features to a Long Short-Term Memory (LSTM) network. The LSTM would then predict the bounding boxes corresponding to the next frame.

If we did not want to rely on depth images and only wanted to use pure RGB images for this pipeline, we could use the size of the bounding boxes as a proxy for depth. This method would have issues as hands move in and out of the frustum though. In particular, as users move their hands into view, the hand bounding boxes become bigger until the whole hand is in view. Another way to use only RGB images is to have YOLO also regress on depth. This method involves collecting a dataset that contains depth information and changing the loss layer to include depth as a regression output. The number of outputs for each image would then be S x S x (B x 6 + C) since we have an additional depth field.

6 Acknowledgements

We want to thank Professor Gordon Wetzstein and Robert Konrad for an amazing course, as well as the computer resources needed to run this pipeline.

7 References

[1] Dalal, N., Triggs, B., 2005. Histograms of oriented gradients for human detection. In: Proc. CVPR 2005, vol. 1, pp. 886-893. <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1%467360>.

[2] Frati, Valentino, and Domenico Prattichizzo. "Using Kinect for hand tracking and rendering in wearable haptics." *World Haptics Conference (WHC), 2011 IEEE*. IEEE, 2011.

[3] Girshick, R. B. and Felzenszwalb, P. F. and McAllester, D.: Discriminatively Trained Deformable Part Models, Release 5.

[4] Girshick, R. B., Donahue, J., Darrell, T., and Malik, J. Rich feature hierarchies for accurate object detection and semantic segmentation. CoRR, abs/1311.2524v5, 2014. Published in Proc. CVPR, 2014.

[5] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. arXiv preprint arXiv: 1506.02640, 2015.

[6] J. Supancic, G. Rogez, Y. Yang, J. Shotton, D. Ramanan. "Depth-based hand pose estimation: methods, data, and challenges " *arXiv preprint arXiv:1504.06378* 2015.

[7] Mujber, Tariq S., Tamas Szecsi, and Mohammed SJ Hashmi. "Virtual reality applications in manufacturing process simulation." *Journal of materials processing technology* 155 (2004): 1834-1838.

[8] P. Felzenszwalb, R. Girshick, D. McAllester, D. Ramanan Object Detection with Discriminatively Trained Part Based Models. IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 32, No. 9, September 2010.

[9] R. Girshick, “Fast R-CNN,” in IEEE International Conference on Computer Vision (ICCV), 2015.

[10] Rogez, Grégory, et al. "3d hand pose detection in egocentric RGB-D images." *Computer Vision-ECCV 2014 Workshops*. Springer International Publishing, 2014.

[11] Rosen, Joseph M., et al. "Evolution of virtual reality [Medicine]." *Engineering in Medicine and Biology Magazine, IEEE* 15.2 (1996): 16-22.

[12] S. Bambach, S. Lee, D. J. Crandall, and C. Yu. Lending a hand: Detecting hands and recognizing activities in complex egocentric interactions. In ICCV, 2015.

[13] Segmentation as Selective Search for Object Recognition. Koen E. A. van de Sande. Jasper R. R. Uijlings. Theo Gevers. Arnold W. M. Smeulders.

[14] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. arXiv preprint arXiv:1506.01497, 2015.

[15] Wang, Robert Y., and Jovan Popović. "Real-time hand-tracking with a color glove." *ACM transactions on graphics (TOG)* 28.3 (2009): 63.