Put a Ring On It:
Improved Object Segmentation for Apple ARKit Applications

Dea Dressel
ddressel@stanford.edu

Daniel Valner
dvalner@stanford.edu

Jessica Yeung
jyeung27@stanford.edu

March 16, 2022

Abstract

Building mobile augmented reality (AR) applications has become increasingly accessible to amateur developers in recent years. In 2017, Apple released ARKit, a development tool that enables creators to make smartphone-based AR experiences. In this project, we build off of the "SegmentationAndOcclusion" repository by Dennis Ippel, an ARKit repository that performs object detection, semantic segmentation and occlusion. The two major objectives for this project include (1) to learn about the built-in capabilities of ARKit and how the repository uses the YOLOv3 and DeepLabv3 models in its implementation and (2) to improve two technical features of the repository, namely changing from static to dynamic sizing of bounding rings during object segmentation and moving from single object recognition to multi-object recognition and segmentation during a singular, live feed. Both objectives were successfully accomplished.

1 Introduction

The surging demand for augmented reality applications in various industries has inspired developers across the globe. The vision began with AR glasses, a wearable device that could seamlessly blend augmented and physical reality. However, recent innovation in this field has directed its focus at a much more addressable market: mobile AR. Around 84% of the global population owns a smartphone. While not all devices today are AR compatible in terms of optical and processing components, most will be over the next replacement cycle of 2.5 years. Capitalizing on this widespread use of smartphones proves common among AR leaders in the tech sector.

In 2017, Apple released ARKit, a development tool that enables creators to make smartphone-based AR experiences. The main feature of an AR application is superimposing a computer generated image into the physical world. ARKit boasts a wide range of capabilities, including but not limited to instant placement of AR objects in the physical world through LiDAR plane detection, motion capture of a person in real time with a single camera, topological mapping of a space, and people occlusion. This democratization of advanced AR through widely available software and hardware has significantly lowered the barrier to AR.

While some applications of AR (e.g. face filters) may seem simple to the everyday user, they all rely on sophisticated computer vision algorithms. Object detection and semantic segmentation are core tasks in computer vision. Object detection aims to localize and recognize every instance of a specific object in an image and then mark it by a bounding box. Semantic segmentation assigns a category label to each pixel in an image. Combining these techniques together in the context of ARKit introduces a reliable framework for generating new mobile AR applications.

In this report, we will first unpack the inner workings of the "SegmentationAndOcclusion" repository by Dennis Ippel, an ARKit repository that performs object detection, semantic segmentation, and occlusion. This section will be two-fold: an overview of the built-in capabilities of ARKit and a deep-dive discussion into how the model uses YOLOv3 for object detection and DeepLabv3 for se-

mantic segmentation. Next, we will identify and address two major shortcomings of Ippel's work, namely changing from static to dynamic sizing of bounding rings during object segmentation and moving from single object recognition to multi-object recognition and segmentation during a singular, live feed. Finally, we comment on our results, technical difficulties, and future improvements to our app.

2 Background

2.1 ARKit

ARKit is Apple's suite of development tools for building AR applications. ARKit uses Visual Inertial Odometry (VIO) to accurately track the world, combining camera sensor data with the device's CoreMotion data. These inputs allow the iOS device to accurately sense how it moves within a room, eliminating the need for additional calibration.

ARKit's features include plane detection, person detection, as well as object anchoring. These features allow applications to place artificial objects on a scene, letting humans pass in front and behind of them. These recognition, segmenting, and occluding capabilities, however, are limited to humans. If one wishes for the application to detect and manipulate other objects in the scene, then they must use the ARReferenceObject class. With this class a user/developer "scans" a specific object, using structure from motion algorithms, and only then can they interact with it in the application scene. There are no other object detection methods built into ARKit for non-human objects. Our goal is to use external algorithms to expand object detection and segmentation in ARKit.

2.2 YOLOv3

YOLOv3 is a real time object classification algorithm that detects certain objects in images and generates bounding boxes around them. It's the newest iteration of the YOLO and YOLOv2 models made by the same researchers, Joseph Remon and Ali Frahadi. This model applies a CNN to an image, divides the image into grid cells, and produces the probabilities that each cell is a part of one of 80 object classes.

To learn the image features, YOLOv3 uses Darknet-53, a feature extractor with 53 convolutions. Using these features, the model makes object detections at 3 distinct layers using different strides in the convolutions at each layer to detect objects of various sizes. To do so, each cell keeps track of the probability that it is the center cell of a bounding box around a certain object. Additionally, the object includes the dimensions of the relative bounding box it would be the center of which are also learned parameters although initialized through unsupervised learning. Each cell keeps track of the attributes and probabilities of 3 bounding box predictions.

The dimensions of the 3 anchor boxes are generated through k-means clustering. The model is trained with one ground truth bounding box per object and the center cell is assigned to predict the object. The model uses a Leaky ReLU loss function and implements batch normalization. Other features in the network that optimize performance include residual blocks, skip connections and up-sampling. There are no pooling layers but rather just convolutional layers with 2 x 2 convolutions with stride 2 to prevent loss of low level features to help detect small objects. The model was trained on the COCO dataset which establishes the 80 classes of objects that can be detected.

2.3 DeepLabv3

DeepLabv3 is an image semantic segmentation model that classifies each pixel in an image as belonging to one of the 20 Pascal Visual Object classes. Features are extracted from a backbone network (VGG, DenseNet, ResNet) and fed into DeepLabv3. The model is trained on the Pascal Visual Object classes and uses batch normalization. To control the size of the feature map, the model uses dilated/atrous convolution (holes between pixels in convolution) to get a larger field-of-view while using a similar computation standard convolution filter. This controls the output of one atrous convolution from being too small which is often a problem in semantic segmentation. Multiple cascading atrous convolutions are combined as an Atrous Spatial Pyramid Pooling (ASPP) at the end of the network to segment objects at different scales. The output from the ASPP network is passed through a 1 x 1 convolution to get the actual size of the image which will be the final segmented mask for the image.

3 Technical Approach

Our project builds off of a skeletal ARKit framework titled "SegmentationAndOcclusion" by Dennis Ippel. Built in ARKit, the codebase uses YOLOv3 to locate and classify an object and DeepLabv3 to segment the detected object's pixels and calculate depth in a Metal fragment shader.

3.1 Implementation

The first major shortcoming of the existing project was how it relied on statically sized bounding rings. When identifying different objects in the world space where objects can be varying sizes, it is important to adapt the bounding rings to the size of the newly detected objects. For example, moving from segmenting a person to a couch in the same live-feed is a large difference in size, and we believed that our implementation needed to address this issue.

The second major shortcoming of the existing project was how it could only recognize one type of object at a time. However, for most AR application uses, it is necessary to be able to segment multiple types of objects.

3.2 Code Structure

There are three main files that handle the object detection and segmentation: `ViewController.swift`, `HighlighterNode.swift`, and `segmentationMaskNode.swift`.

3.2.1 ViewController.swift

The overall structure of the rendering pipeline is defined in our `ViewController` class. The current scene frame is captured and an image request handler instance is instantiated from the `VNImageRequestHandler` class. The `VNImageRequestHandler` is created with a frame to be used for any requests the app client wants to schedule, like our object detection and segmentation requests. The handler never modifies the image source but rather retains it for its entire lifetime. After defining the image request, we pass in our array of function requests for the handler to perform: `objectDetectionRequest` and `segmentationRequest`, both variables which are defined within the `ViewController` class. The order is important since we need to retrieve the detected object first from YOLOv3 and then apply the segmentation to the rendered camera view using DeepLabv3.

To store the current segmentation and apply the yellow highlight visual to our object, we instantiate two class variables within the `ViewController`: `quadNode` and `highlighterNode`. The variable types are defined in two other Swift class files, `SegmentationMaskNode.swift` and `HighlighterNode.swift`.

The `objectDetectionRequest` executes the YOLOv3 model and calls the `processSegmentations()` class function within the request and returns this request to the image handler explained above. The `segmentationRequest` executes the DeepLabv3 model and calls the `processObjectDetections()` class function within the request and returns this request to the image handler explained above as well.

Within the `processObjectDetections()` function, the object observations are returned and sorted based on the confidence scores from highest to lowest based on the matching of the detected object to the target object label. Once the best observation is returned (the first item in the sorted array of detections), the area of the observation is expanded to account for edge noise and provide a more accurate detection mask. This region is then passed to the segmentation mask node/quad node variable in order to label the according pixels as the segmented area. The segmentation mask that displays what is being segmented in the camera view is also passed to the segementation mask node variable in order to know which pixels to shade yellow in the `processSegmentations()` function.

Within the `processSegmentations()` function, we retrieve the object observation from the DeepLabv3 model request. Then, the observation's pixel buffer with the object mask is copied over to an Apple Metal compatible pixel buffer. A texture cache is generated using this pixel buffer, which holds the texture for the image frame as well as the texture for the quad node variable that stores the current segmentation mask node.

3.2.2 HighlighterNode.swift

The highlighter node manages the rotating bounding rings in the scene. Two torus geometries are generated and added as children nodes to the highlighter class parent node. A constant rotation animation is applied to both, and the radii are determined by the bounding box size of the detected object.

3.2.3 segmentationMaskNode.swift

The `SegmentationMaskNode` stores the texture, region, classification label, depth buffer, and aspect ratio adjustment for a segmentation observation. A single segmentation mask node is initialized as the variable `quadNode` in the ViewController class. The segmentation node also stores the pixel information to display the segmentation mask that flashes yellow in our application.

3.3 Dynamic Sizing of Bounding Rings

In order to switch from static to dynamic sizing of the rings, there are two main code files that we had to modify: `ViewController.swift` and `HighlighterNode.swift`. The object detection is handled in the `processObjectDetections()` function. After a classification request is made to the YOLOv3 model, an array of possible object classifications are returned and the observations are sorted from highest to lowest confidence. The best object is found by indexing the first item in the sorted observations list.

Once the best observation is returned, we can capture the bounding box width and height of the object from the observation data. This information is then passed to an `updateNode()` function in the `HighlighterNode` class and used to scale the radii of the bounding rings. We chose to use the largest of the width or height as the base radius, scaled it by 1.1x for the outer ring, and scaled it by 0.9x for the inner ring. We made the thicknesses to be 0.02x the radius for the outer ring and 0.01x the radius for the inner ring.

In the `HighlighterNode` class, the 3D modeled rings are added as children nodes to the parent `HighlighterNode` object instance in the `setupNode()` function. This function is called in the initialization of the highlighter node. The highlighter node is turned on when an object is detected by setting a boolean `isHidden` parameter of the parent `HighlighterNode` instance. In order to update the size of the rings, we ensured that any previous child node for the outer and inner ring are removed before calling the `setupNode()` function within the `updateNode()` function as shown below.

```
public func updateNode(w: Double, h:
    Double) {
   self.childNodes.filter({ $0.name ==
       "outerRing" }).forEach({
       $0.removeFromParentNode() })
   self.childNodes.filter({ $0.name ==
       "innerRing" }).forEach({
       $0.removeFromParentNode() })

   setupNode()
}
```

We decided to keep the main `HighlighterNode` instance instantiated and instead replace the child ring nodes because the initial view controller of the ARKit app instantiates both the segmentation mask and highlighting mask nodes once - this helps improve performance so an entire class instance does not have to be continuously removed and added. It is faster and simpler to remove the children nodes that only handle the 3D rings and keep updates to components of the class to functions within the parent class.

3.4 Detecting Multiple Types of Objects

In order to detect multiple objects, we focused on editing the structure of the `SegmentationMaskNode` class and the `processObjectDetections()` function in the `ViewController.swift` class. There are several private variables that encode the label information: `labels`, `targetObjectLabel`, and `targetObjectLabelindex`. Initially, both the `targetObjectLabel` and `targetObjectLabelindex` were singular values. To accommodate multiple types of objects, we changed these to a string array and `UInt` array, respectively.

In the `SegmentationMaskNode` class, we also updated the `Uniforms` and class variables to reflect the change to arrays. Because we are performing the same op-

erations on many pixels, we can use `simd` type variables to encode the `Uniforms` to improve processing speed, meaning the computations will be performed in parallel across blocks of pixels. Thus, for detecting 3 types of objects, we can use a `simd_uint3` to encode an array of three `UInt` variables for the indices of the labels in the full label array.

In the `processObjectDetections()` function, we updated the sorting and retrieving of the best observation code. Now, when the observations are sorted, we add checks to see if each of the target labels is detected:

```
objectObservations.sort {
    return $0.labels.firstIndex(where: {
        $0.identifier ==
        targetObjectLabel[0] }) ?? -1 <
        $1.labels.firstIndex(where: {
        $0.identifier ==
        targetObjectLabel[0] }) ?? -1 ||
    $0.labels.firstIndex(where: {
        $0.identifier ==
        targetObjectLabel[1] }) ?? -1 <
        $1.labels.firstIndex(where: {
        $0.identifier ==
        targetObjectLabel[1] }) ?? -1 ||
    .
    .
    .
    $0.labels.firstIndex(where: {
        $0.identifier ==
        targetObjectLabel[numLabels - 1]
        }) ?? -1 <
        $1.labels.firstIndex(where: {
        $0.identifier ==
        targetObjectLabel[numLabels - 1]
        }) ?? -1
}
```

Additionally, after we retrieve the best observation, we can check if the label matches any of the object labels in the `targetObjectLabels` array variable before returning the best, label-matching observation.

3.5 Results

Image stills from our AR app are shown below. We were successfully able to size the rings and segment objects of type car (Fig. 1), sofa (Fig. 2), and person (Fig 3).

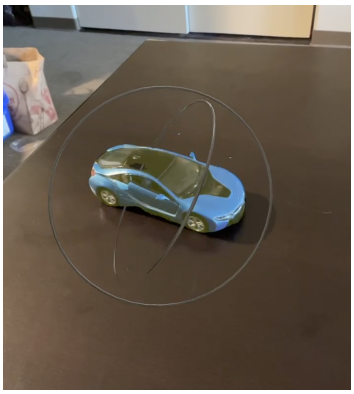

Figure 1. Toy car segmentation

Figure 2. Couch segmentation

Figure 3. Person segmentation

It should be noted that, as you can see in the person still, the rings do not recognize the table as being in front of the person. This is because the bounding rings only use the depth map of the segmented object.

4 Conclusion

Our attempt to understand the inner workings of the "SegmentationAndOcclusion" repository by Dennis Ippel was successful. We developed a solid understanding of the built-in capabilities of ARKit, including depth buffers, understanding the standard graphics pipeline, and Swift syntax. We also learned how the provided repository uses YOLOv3 for real time object classification that enables the model to detect certain objects in images and generate bounding boxes around them. YOLOv3 relies on a CNN to divide an image into grid cells and produce the probabilities that each cell is a part of one of the 80 object classes. We learned how the provided repository then uses DeepLabv3 for semantic segmentation to classify each pixel in an image as belonging to one of the 20 Pascal Visual Object classes. Together, YOLOv3 and DeepLabv3 outperform the built-in ARKit capabilities. ARKit does not use a live segmentation model for other objects and requires multiple-view photos of a real object to be able to execute any object tracking/segmentation/AR projection. Our framework gives developers more flexibility in the types of objects their programs can interact with.

Additionally, we were able to identify and address two major shortcomings of the "SegmentationAndOcclusion" repository. The first improvement we made to the model was moving from static to dynamic sizing of the bounding rings. After an object observation during a live-feed is returned by the model, we captured the bounding box width and height of the object from the observation data. We then used this information to dynamically scale the radii of the inner and outer bounding rings. Now, the bounding rings adjust according to the dimensions of the detected object instead of being a static size for any type of object. With this improvement, the model displays accurate bounding rings that allow users to gain insight into the dimensions of the object in the physical world.

The second improvement we made to the model was moving from singular- to multi-object detection from a live-feed. Previously, the model could only look for one type of object at a time and required the user to kill the current process, reset the object label of interest, and rerun the code in order for the model to detect a different object type. We changed the implementation to instead take an array of object types at the beginning of runtime. This required many downstream code changes to accommodate for an array of labels versus a singular label. After we made those changes, the model is able to detect multiple types of objects while only highlighting one of the objects at a time.

This work demonstrated that it is possible to learn about ARKit, YOLOv3, and DeepLabv3 in order to improve a preexisting object detection and segmentation ARKit repository.

Further work includes increasing the types of objects segmented, since not all the DeepLab object labels worked consistently. Additionally, we would like to explore recognizing multiple objects at once so 3D geometry can be integrated into a scene more realistically. This would improve the segmentation shown in Figure 3.

Code for the project is available in a public repository at https://github.com/jyeung27/cs231a-final-project.

5 References

"ARKit Tutorial: The Complete ARKit Developer Course for iOS 11." YouTube, uploaded by Codestars, 2018, https://www.youtube.com/watch?v=f3xFpRWZEz8.

Capturing Photographs for RealityKit Object Capture. Apple. https://developer.apple.com/documentation/realitykit/capturing_photographs_for_realitykit_object_capture/.

Capturing Photos with Depth. Apple. https://developer.apple.com/documentation/avfoundation/cameras_and_media_capture/capturing_photos_with_depth.

Celik, Eren. (May 19, 2021). SwiftUI + Core ML + ARKit — Create an Object Detection iOS App. https://betterprogramming.pub/swiftui-core-ml-arkit-create-an-object-detection-ios-app-2c74fc57d984.

Dennis, Ippel. (2020). SegmentationAndOcclusion, GitHub Repository. https://github.com/MasDennis/SegmentationAndOcclusion.

Dennis, Ippel. (2020). Using CoreML in ARKit for Object Segmentation and Occlusion. Medium. https://rozengain.medium.com/using-coreml-in-arkit-for-

object-segmentation-and-occlusion-988a6c7e18dd.

Dennis, Ippel. (2020). Using Vision Framework Object Detection in ARKit. Medium. https://rozengain.medium.com/using-vision-framework-object-detection-in-arkit-c0b5366f465d.

Displaying an AR Experience with Metal. Apple. https://developer.apple.com/documentation/arkit/displaying_an_ar_experience_with_metal.

How DeepLabv3 Works. ArcGis Developer. https://developers.arcgis.com/python/guide/how-deeplabv3-works/.

RealityKit. Apple. https://developer.apple.com/documentation/realitykit/.

Recanatesi, Stefano, Matthew Farrel, Madhu Advani, Timothy Moore, Guillaume Lajoie, and Eric Shea-Brown. (2019). "Dimensionality Compression and Expansion in Deep Neural Networks."

Redmon, Joseph, Ali Farhadi. "YOLOv3: An Incremental Improvement." University of Washington.

Turner, Ash. (2022). How Many Smartphones Are In The World? https://www.bankmycell.com/blog/how-many-phones-are-in-the-world.

YOLOv3 Object Detector. ArcGis Developer. https://developers.arcgis.com/python/guide/yolov3-object-detector/.