# Camera-based occlusion for AR in Metal via depth extraction from the iPhone dual camera

Cristobal Sciutto
Stanford University
cristobal.space

Breno Dal Bianco
Stanford University
bdbianco@stanford.edu

## Abstract

*In this project, we will explore the development of a proof-of-concept of occlusion with camera-based AR on iOS's graphics platform Metal. Per Cutting and Vishton's survey of the relative importance of visual cues, it is clear that occlusion is the most important for generating a realistic virtual experience [1]. We hope that our developments pave the way for future compelling consumer-grade applications in the celular space. Our implementation involves tapping into both the phone's capture system, extracting disparity information from an iPhone X's dual-camera, and implementing appropriate shaders for occlusion. The difficulties tackled in this project are two-fold: (i) iOS API deficiencies, and (ii) processing of inaccurate depth information. Source code and GIF examples of the app in usage are available at* `https://github.com/tobyshooters/teapottoss`.

## 1. Introduction

In this section we explore the motivation behind the development of a solution to occulsion, particularly in the context of camera-based AR on phone.

### 1.1. Motivation

Per GSMA's 2019 Mobile Economy report, of the 5.3 billion humans over the age of 15, 5 billion of them own a smartphone [2]. Of the 5 billion users, iOS and Android constitute the largest user base. Consequently, most consumer-grade AR experiences are being developed for Android and iOS devices. They are made using ARCore or ARKit, Apple and Google's proprietary AR frameworks, which specifically target their own hardware.

The problem with mobile development, however, is that without specialized hardware, it is hard to get the sensor input needed for the generation of a quality AR or VR experience. Namely, there is the lack of a depth map, i.e. a notion of how far objects in the real world are from the phone. This information is essential to to be able to occlude virtual objects which are behind real objects. For example, looking at the Ikea Place app in 1, which allows users to place virtual furniture into the physical world, we see that it does not support occlusion. While the sofa is behind the man, it is pasted on top of him in the image plane. Rather, the region of the sofa that the man is covering should not be rendered.

This is severely detrimental to the user's experience as occlusion is the most important visual cue, per Vishton and Cutting [1]. In other words, it is the most important real world effect through which a human is capable of distinguishing where objects are located in the world. Without occlusion, a user experience cannot be compelling.

Devices such as Microsoft's Hololens have time-of-flight sensors that query the real world and measure the time it

Figure 1. Ikea Place app. Note that lack of occlusion.

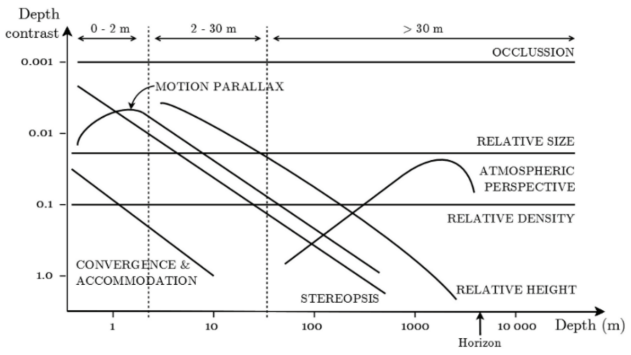


Figure 2. Relative importance of visual cues per [1].

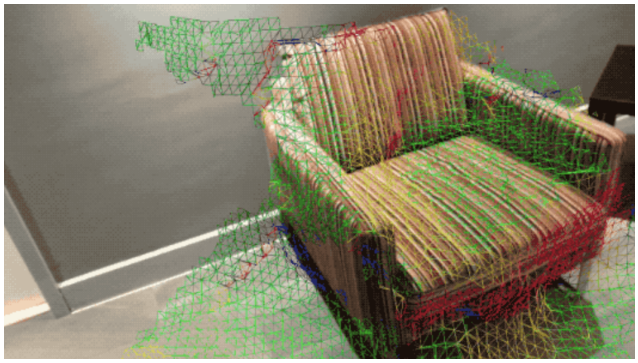

Figure 3. Example of 3D scene mesh reconstruction with Selerio

takes for light to reflect back into the device. With this information, along with the intended locaiton of an virtual object, it is possible to conditionally render an object only in the case that it is the closest to the screen. While a time-of-flight sensor is the best possible solution to the problem, a low resolution sensor is sufficient for most existing experiences. Since most objects are of considerable scale, e.g. a couch, the resolution of depth needed is closer to meters than milimeters.

Fortunately, a solution to this problem is possible within the form-factor of a phone. Namely, the usage of two cameras that generate a stereo pair is sufficient for calculating disparity, which can be then approximated to depth. All iPhones since the 7 and the Android competitor, Google's Pixel, have two cameras. Currently, the dual cameras are primarily used for computational photography. A disparity map is sufficient for simulating effects such as depth-of-field or face-from-backgroudn isolation. This has led to many applications such as the iPhone's portrait-mode. Nevertheless, this information has not yet been tapped for AR experiences, beyond face filters.

Using this information is the first frontier for bettering current consumer-grade AR experiences, and the area in which the greatest number of users can be impacted the quickest.

## 2. Related Work

In the mobile space, there existing solutions to AR occlusion use one of two methods: (i) 3D scene reconstruction, or (ii) preprocessing of scene. In the first case, a series of images captured from the phone are processed together to create an estimate of the real world scene. With this 3D scene, along with the phone's position in the scene, one can estimated the depth in a given camera direction. The reconstruction is often done with help of a machine learning algorithm. The Selerio SDK for cross-platform AR development implements this approach [5]. The obvious drawbacks are the inaccuracies of reconstruction, and the computational intensity of the approach. Furthermore, most existing algorithm's require an initial calibration of the scene at hand. ARKit requires similar initialization, so it could be argued that this is not a further cost to the user experience, however, it is clear that the ability to have occlusion out-of-the-box would be ideal.

In the second case, the user is prompted to create bounding boxes for existing objects in the scene before beginning his experience. With these bounding boxes oriented in world space, it is similarly possible to estimate depth from them and thus occlude virtual objects. This initialization phase is even more costly to user experience than the reconstruction, as users are required to manually generate 3D geometry for the rendering system. This approach is taken by an existing open source project on GitHub by Bjarne Lundgren [3], and can be seen in figure **??**.

Overall, it is clear than any preprocessing step is non-ideal. Using depth sensors directly forgo this problem, and this further motivates our approach of using mobile phone dual cameras. Nevertheless, a combination of depth sensors with scene reconstruction, as used in current AR hardware will most likely be the way to go. This combined approach allows both immediate usage, and increased accuracy over time.

At Apple's World Wide Developer Conference in 2019 (after the development of this project), Apple announced "people occlusion", i.e. occlusion for the specific case of people. These features will be released in the form of their new Reality Kit SDK. This does not yet solve that challenge solved by our application but makes it clear that it is Apple's intention to eventually reach the point where this information is possible. We speculate that since the depth information likely passes through a machine learning pipeline, objects of which we have more data (e.g. faces and bodies) will be the first to have occlusion enabled.

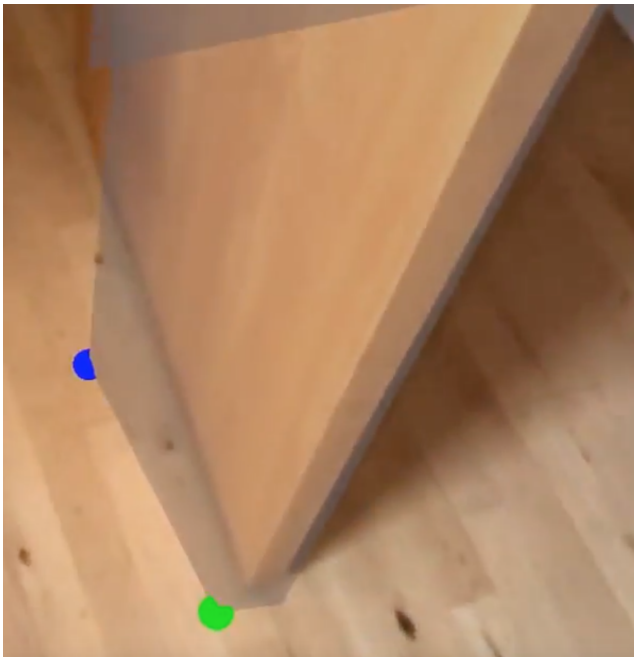

Figure 4. Bjarne Lundgren's app, with the user establishing occluding boxes.

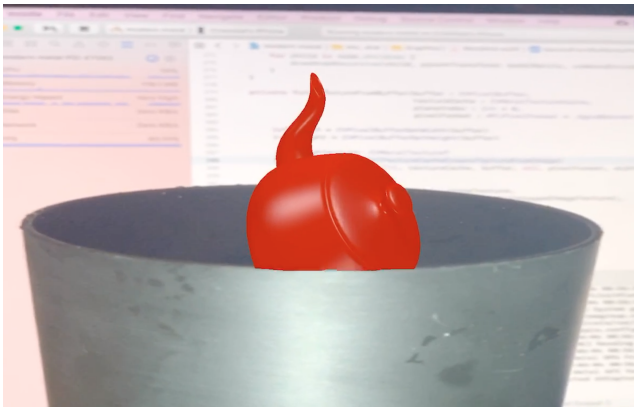

Figure 5. Successful occlusion with out final product.

## 3. Our Work

In this section we elaborate on the final product developed and the process to reach that solution.

### 3.1. Overview

Our final product consists of an iOS game called Teapot Toss, as observable in 5, which allows users to throw virtual teapots into real-world cups. This is done by accessing the camera stream for depth information and conditionally rendering the teapot whether it is occluded by the real-world object. The camera information is extracted from the phone's camera stream using the iOS AVFoundation SDK. We then built a renderer in Metal (Apple's lower-level graphics framework) that describes the 3D scene and renders it with a shader.

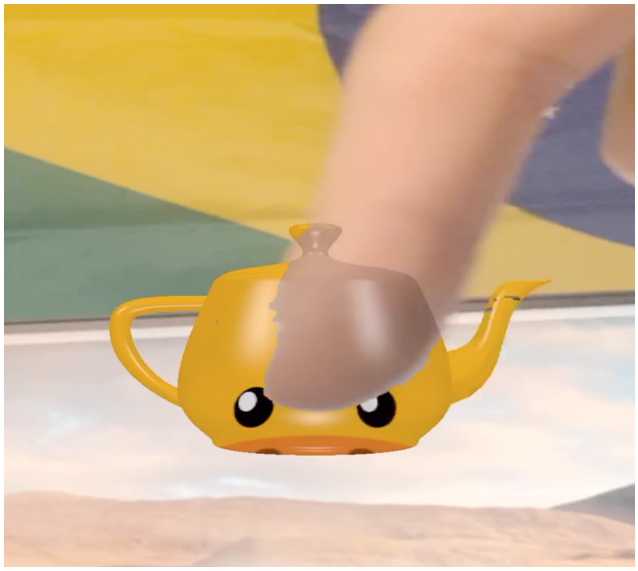

Figure 6. Occlusion shader uses image as base color, but adds Phong shading.

For Teapot Toss we have a 3D scene constitued of two objects: the teapot and an AR image plane. The former has the ability of being thrown upon tapping. The later hosts an image of the real world captured from the camera.

The renderer receives both the captured image from the phone and the depth map from the iPhone's camera stream and passes them as textures into a shader. For the AR image plane, we place its edges along the border of window space. Consequently, the AR image plane occupies the entirety of the user's screen. The window space is then mapped to UV coordinates which are used to sample the captured image. We thus effectively have a live stream of the phone's camera being displayed on this plane. This can best be visualized by using the color from the image, but still performing specular highlighting on the teapot, per figure 6. Notably, the pot is occluded by the finger, yet we can still see the contour of the object because the normals of the pot are still being used for highlights.

The teapot is then rendered in front of this plane. We get the depth from the depth map corresponding to a fragment of the teapot by projecting the fragment onto window space and sampling the depth map texture. If the world space depth of the teapot is greater than the depth map, we use the same UV coordinates to sample the captured image texture, thus painting the teapot with the video stream. This process is best summarized by a example of Metal shader, available in the appendix.

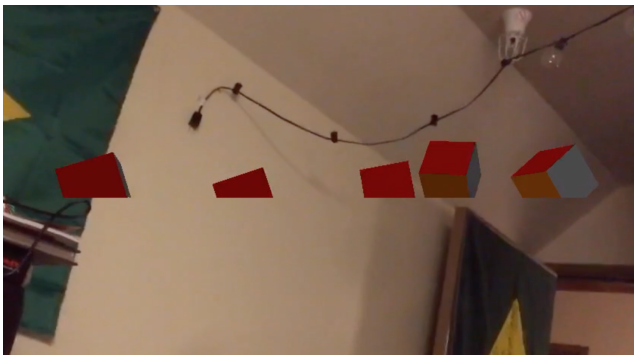

Figure 7. First demo, occluding bottom-half of objects with shader.

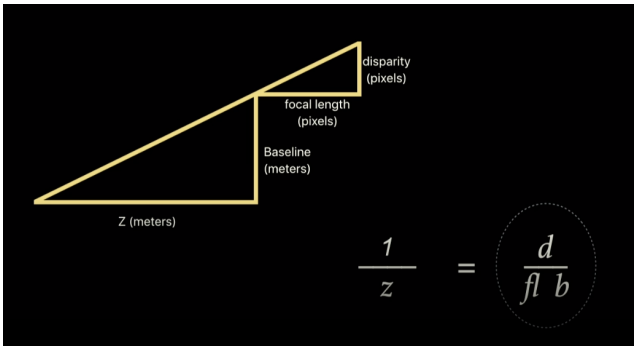


Figure 8. Depth to disparity approximation from WWDC 2017, Session 507.

### 3.2. Occlusion Shader and Extracting Depth

Our initial belief was that this project would be a straightforward implementation. Even though neither of us had experience in iOS development, we knew Apple's ARKit had the reputation of being highly usable and that the iPhone dual-camera was capable of generating depth information. Breno was initially responble for generating a scene and writing the shader that could conditionally render, while Cristobal focused on accessing the camera stream to get depth information, and perform the signal processing necessary for the information to be usable.

On the graphics side, we quickly realized that ARKit built on top of SceneKit, Apple's high-level graphics framework, did not support custom renderers. Shading was done by specifying materials rather than by hand. We thus opted for ARKit built on top of Metal, Apple's lower-level graphics framework comparable to OpenGL. On each render cycle, ARKit provides the camera stream information. This allowed us to write a shader that projects a point onto window space, samples a texture corresponding to the camera stream, and use that value as the color of the fragment. Consequently, we are able to make objects disapper. This functionality can be seen in our occulsion demo in figure 7 which only renders objects if they are in the lower half of a phone screen. Note that the cubes seem to dissapear completely.

For the depth map, Cristobal was able to get the depth information by referencing the code used in a popular Github repo, iOS-Depth-Sampler, which has example code for a variety of depth map related applications in iOS [6].

One realization was the iOS only provides "True Depth," i.e. accurate in real-world information, for the front-facing camera. We had, however, the need to use the back-facing camera for both our TeapotToss application and for possible future applications such as extending the Ikea Place app with occlusion. The back-facing camera provides disparity information that is generated by comparing the stereo pair of images. From disparity, depth can be approximated if we know the focal length and relative positioning of the cameras. The derivation is further elucidated in session 517 from the 2017 WWDC (8), and is abstracted out by Apple's API. Therefore, while we could not access "True Depth" information, we could still get depth information, albeit less accurate. One small note is that the depth information is of lower resolution than the image size, however when converting to a texture, the information is interpolated and normalized to $[0, 1]^2$, automatically fixing the problem.

To clean up the depth information, both a Gaussian blur of radius 3 pixels and gamma adjustment of power 0.5 were performed. The former cleared up unrealiable high frequency information, whereas the later spread out the information so that it was easier to separate out different objects. This depth information was provided in meters. However, when generating a texture with the pixel buffer, the values are scaled to the range $[0, 1]$. Consquently, despite the filtering, some outlier spikes led the depth information to be concentrated in the lower half of the $[0, 1]$ range. This is shown clearly in figure 9.

In order to preserve all the whole range, we could pass the maximum depth value into the shader, and reconstruct the complete depth information (save floating points errors). At this point, however, our processing becomes more ad-hoc for the particular application of teapot tossing desired. Rather than compare the teapot's real world distance to the depth map real world distance, we map the values to the $[0, 1]$ range, optimizing the mapping for the occlusion effect.

We map the depth map's $[0, \frac{1}{3}]$ subregion to $[0, 1]$ and then discretize the values into 10 buckets. This allows us get more disparate regions of depth, and leads to a very accurate depth illusion in the app. The teapot's depth is similarly scaled to the range $[0, 1]$. We also noticed that higher contrast in the region being measured leads to better depth maps, and thus usually demo with a light directly illuminating the occluder.

Figure 9. Pre-filtered depth information showing a hand. Information is concentrated the range $[0.2, 0.3]$.

### 3.3. Integration and transition to bare Metal

One key challenge that arose was the discovery that Apple's camera stream can only have one consumer at a given time. Access to the stream is done via a AVCaptureSession abstraction. This is used both by the depth map extraction code and ARKit's ARSession API. Therefore, when trying to connect Cristobal's depth map effort to Breno's occlusion prototype, we were blocked by the operating system. Either we could get depth information from the camera, or we could have an ARSession running. Interestingly, the ARSession could, in theory, provide a depth map since it consumes the dual camera. This, however, is not done in practice by Apple's API. Therefore we were forced to forgo the ARSession and implement a rough equivalent of ARKit in Metal. We are indebted to Warren Moore's *Metal by Example* for providing the foundational code for this [4].

As described above in the overview, this is done by essentially pasting the captured image onto a plane within a 3D scene that is rendered by Metal. This is in essence the core of AR rendering: using a camera image as a texture within a 3D scene. This proved succesful and is the approach used in the final product.

The final important detail is the implmentation of animation of our virtual objects. This is no longer abstracted away by ARKit and thus must be done from scratch. In our renderer, we create a tree of objects with associated transforms. These transforms are passed into shaders as object transforms which are then used to find the object's correct position. By updating the object transform on each render, we are able to animate our teapot.

### 3.4. Code Overview

Here we provide a overview of the important files and operations done in our iOS app. The entire source code is available at `https://github.com/tobyshooters/TeapotToss`.

- *ViewController.swift*: this file contains all logic for initializing a session with the camera and extracting video and depth information. The depth information is processed in this file, applying both the Gaussian blur and the gamma adjustment. It is the class that orchestrates the rendering process. and is consequently where the Renderer instance is initialized. Finally, ViewController also handles taps and thus is the trigger for the teapot's animation.

- *Renderer.swift*: the Renderer class is responsible for all the logic for teapot rendering and animation. It receives camera information as a parameter. This is where the rendering pipeline is determined, scene objects are described, and camera information is transformed from a pixel buffer into a texture.

- *Shaders.metal*: contains the shaders for both pasting the camera image onto the 3D plane, as well as lighting for the virtual depth map. The shader scales the depth map data to range of data and discretizes it. It then assigns the fragment color to either it's described material or the video information.

- *Scene.swift*: abstractions for describing a scene, provided by the Metal by Example tutorial [4].

If one were to analyze the application in terms of data flow it is best explained as: Camera information is accessed via $AVCaptureSession$ in the View Controller class, and then forwarded to the Renderer defined in $Renderer.swift$. The Renderer, using $Scene.swift$'s abstractions then creates a scene to be rendered by the Shader.

### 3.5. Running from Source

In order to run the code, you need both a Apple computer and an iPhone with dual cameras. Then one must simply clone the source code, connect the phone to the computer, and run the XCode project using the phone as the target.

## 4. Discussion and Future Work

### 4.1. Discussion

It is clear that the weakest point of our application is the depth map processing. While we can in fact extract this information, it is scaled to suit the range needed for our application: teapot tossing. This restricts the applications usage to trash cans around 1 foot from the camera. Furthermore, by forgoing ARKit, we are no longer tracking the world and thus the teapot is in a fixed position relative to the phone's position. While this could have been done in iOS by querying gyro and accelerometer information, it was out of the scope of our project.

Nevertheless, depsite this arbitrarily scaled depth information, we believe that the visual effect we have produced is quite good. We are able to both toss objects into cans

and have them disapper, as well as wave our hand through the object, having it disapper and reappear in sync with the hands movement.

Our experience with iOS was very mixed. It is great for developing applications that Apple wants you to make, but is an incredible burden for anything that is outside of that scope. The fact that ARKit's ARSession does not provide depth information seems to be a deliberate move by Apple to prohibit our kind of application from being developed. Our hypothesis is that, since the depth information is not of the highest quality, Apple engineers are not comfortable exposing the information as it will lead to poor user experiences. This is in accordance to Apple's design philosphy, but is certainly detrimental to engineering tinkering and the advancement of consumer-grade AR apps.

### 4.2. Future Work

Our developed app could be bettered by an more investment in the processing of the depth map. With sufficient time, we would have liked to have cleaned up outliers in the depth map. This would allow us to have richer depth information as it wouldn't be compressed to a small region. Furthermore, it would be better to be comparing real-world depths rather than massaged depths in the $[0, 1]$ range.

We would also like to make some comments on AR platform integration. A key aspect of software, its modularity, is currently being ignored. What we observe are several proprietary hardware/software solutions that are tightly integrated, such as Apple's iPhone with ARKit. We believe that by formalizing the hardware/software interface into a agreed-upon universal interface, many advances can be shared across the industry.

For example, if a depth map is outputed by an arbitrary hardware system, it can be massaged by an integrating software layer to be compatible with the universal interface. Therefore, any software expecting a depth map would be able to run on that hardware. This would allow software development to advance independently of hardware, and would allow for quicker iteration of protoytpes. In that world, our developments here would be fruitful for the entire graphics industry, rather than just the iOS developers interested in occlusion.

## 5. Conclusion

Overall, we believe that our product achieves our original goal of creating a paper toss game similar to the origina iPhone app that uses a real-world trash can and a virtual paper ball.

We started off with no iOS experience and were eventually able to coerce Apple's graphics stack to do what we desired, despite a series of barriers established by the iOS and ARKit APIs. By implmenting an AR pipeline in Metal, rather than relying on an existing abstraction, we had access rendering pipeline of our application and thus able to get the desired effect. Nevertheless, this kind of hacking is far from accessible to the average iOS developer, and requires more foundational graphics knowledge, ranging from the math involved in perspective transforms and signal processing, to the systems knowledge of how Apple's highly abstracted IDE actually works.

Hopefully this project can encourage more developers to start playing around with occlusion in iOS, and push Apple to be more open towards developers who want to work with their hardware, but are unable to due to the software.

## References


[1] J. Cutting and P. Vishton. *Perceiving layout and knowing distances: The interaction, relative potency, and contextual use of different information about depth*, volume 5, pages 69–177. 01 1995.

[2] GSMA. The mobile economy report, 2019.

[3] B. Lundgren. Arkit occlusion demo. `https://github.com/bjarnel/arkit-occlusion`, 2017.

[4] W. Moore. Metal by example. `https://github.com/metal-by-example/modern-metal`, 2018.

[5] Selerio. Selerio sdk. `https://www.selerio.io/`. Accessed: 2019-06-07.

[6] S. Tsutsumi. ios depth sampler. `https://github.com/shu223/iOS-Depth-Sampler`, 2019.

## 6. Appendix

### 6.1. Shader Example

```
fragment float4 fragment_main(
        constant FragmentUniforms &uniforms [[buffer(0)]],
        texture2d<float, access::sample> imageTex [[texture(0)]],
        texture2d<float, access::sample> depthTex [[texture(1)]],
        texture2d<float, access::sample> objectTex [[texture(2)]],
        sampler texSampler [[sampler(0)]])
{
    float worldDepth = depthTex.sample(texSampler, fragmentIn.camCoords).r;

    float3 color;
    if (worldDepth > fragmentIn.position.z) {
        color = objectTex.sample(texSampler, fragmentIn.camCoords).rgb;
    } else {
        color = imageTex.sample(texSampler, fragmentIn.camCoords).rgb;
    }

    return float4(color, 1.0);
}
```