Making AR “Lit” Again:

Recreating Environmental Lighting for More Realistic Mobile AR Experiences

(See bottom of document for screenshots!)

Prathik Naidu (prathikn)
Mihir Garimella (mihirg)

We built a system to capture and recreate the lighting within an environment to make mobile AR experiences richer and more realistic. Our motivation was the fact that current frameworks like ARKit and ARCore aren’t able to create experiences that feel truly immersive in part because they lack visual effects like shadows and reflectance that we’re so used to seeing in the real world.

Here’s how our proof-of-concept app works:

1. First, the user opens our app and enters “Calibration” mode. In the background, ARKit begins using visual features coupled with accelerometer and gyroscope measurements to track the phone’s position and orientation. Within the app, we display a feed from their rear camera.

2. To indicate that they’d like us to model a light, the user can tap anywhere within that light in the camera view. (We could just as easily do this automatically, but most environments have many light sources and ARKit can only handle a few at render-time without crashing, so we found that being able to selectively model lights was helpful.)

3. For each tap, we first make sure that the user has tapped on a light (i.e., a bright white spot in the image) and then use OpenCV to perform a floodfill over the luminance (Y) channel of the image to come up with a binary image mask. If the floodfill results in a region that’s large enough to be a light (based on a threshold we determined through experiments), we give the user haptic feedback and store the light mask.

4. We track this light over subsequent frames until it drops out of the camera view. This involves thresholding the luminance channel of the image to obtain a binary mask with all potential light pixels, and then performing a “bitwise and” between the thresholding result and each binary light mask from the previous frame. If we see overlap, we perform the same floodfill described above, but make sure to “link” this light with the corresponding light in the previous frame. With sufficiently small movement between frames (which holds given that we’re running at 30+ fps), this is a simple but effective way to track a light from frame to frame.

5. For both new lights in a frame and lights tracked from previous frames, we track the light’s centroid to determine its position in 3D space. To do this, we randomly sample 50 2D screen-space points from within the light in each frame. We then have ARKit perform a hit test at each of these points based on its underlying (sparse) map of the world to estimate a 3D location for each point. We then average the results of these points to record an estimate for the light’s centroid in 3D.

6. Because the hit test results are *extremely* noisy and often inaccurate, we aggressively prune previous hit test results based on each new frame that we see. To do this, we project each previously estimated 3D centroid into the 2D screen-space coordinate system of each new frame. If the 2D projected centroid is far away from the light’s actual 2D pixel centroid in the new frame, then we “vote against” the corresponding 3D centroid estimate. Any centroid estimate that has more than a threshold number of “votes against” is discarded. After the pruning step, we take a weighted average of the surviving centroid estimates for each light (earlier estimates are weighted exponentially more than later estimates since they’ve survived many more rounds of pruning) to get the light’s final position and visualize it in AR with a green sphere.

7. Also for each frame, we add one equation to a linear system describing the light temperatures. ARKit gives us an estimate of the overall light temperature within each frame, which we express as a linear combination of the (unknown) temperatures of each light weighted by their area, in pixels, in the frame.

8. Once the user is done with this calibration step, they tap “Finished Calibration.” We’ve already been updating its position in real-time for visualization, but we also need to identify the temperature, type, and direction of each light. To do so:
    a. We solve for **temperature** by using an iterative sparse least-squares solver on the system we’ve been building up.
    b. We determine the **type** of light using a simple heuristic – if the light position is above the camera position in the world coordinate space, we assume the light is a directional light (since it’s likely on the ceiling). All other lights are modeled as spotlights.
    c. We get the **direction** of the light depending on the type of light. For directional lights, we use the vector from the light position to the camera when the user finishes calibration and places an object in the scene (we assume here that the user will place an object near the camera and that the light is relatively far away). For spotlights, we assume that the user initially tapped the light head-on and compute the direction vector from the light to the camera at that time.

9. From here, we add light nodes into the scene based on their 3D position in the world coordinate system. For each of these lights, we set the position, direction, temperature, and type attributes based on the results from the calibration step. We also set up shadow mapping for each light in SceneKit, ARKit’s physically-based renderer.

10. Once the lights are placed in the AR scene, we prompt the user prompted to find a flat surface, and visualize plane detection results with a 3D focus square (based on an Apple demo app) to provide real-time feedback to the user.

11. Once we’ve detected a plane, the user can tap on a “+” button and choose one of a number of objects (Lamp, Cup, Sticky Note, Candle, Chair, Vase, Painting — all demo models from Apple) to place in the scene. They can move the object around the plane by dragging and rotate the object using two fingers.

12. The user will immediately see their object placed in the scene and lit up based on the environmental lighting we’ve set up, with appropriate shadows and specular highlights!

Running our app requires an Apple Developer Program account and an iOS device running iOS 11+. Download our code and then run **`pod install`** in the source directory to download dependencies through CocoaPods. Then, open "ARKitInteraction.xcworkspace" in Xcode and set up app signing with your Apple Developer certificate — choose it by selecting "ARKitInteraction" at the top of the file tree and setting your development team in the "General ➔ Signing" section. Now, plug in your phone and install our app! (Seriously, though, let us know if you'd like to try our app out and we're more than happy to give you a demo/install it on your phone.)

Although we're just building a proof-of-concept here, we're excited about how our work could be extended. For instance, rather than having an explicit pre-rendering calibration step in which we scan the light sources in a room, we could scan the environment in real-time *during* rendering or even constantly monitor the light sources in a room on a background thread before the AR session has started. In addition, we could use this light source modeling during 3D capture as well, to "subtract" the capture-time environmental light from 3D models and then "add" the rendering-time environmental light later on, for even more realistic AR experiences.

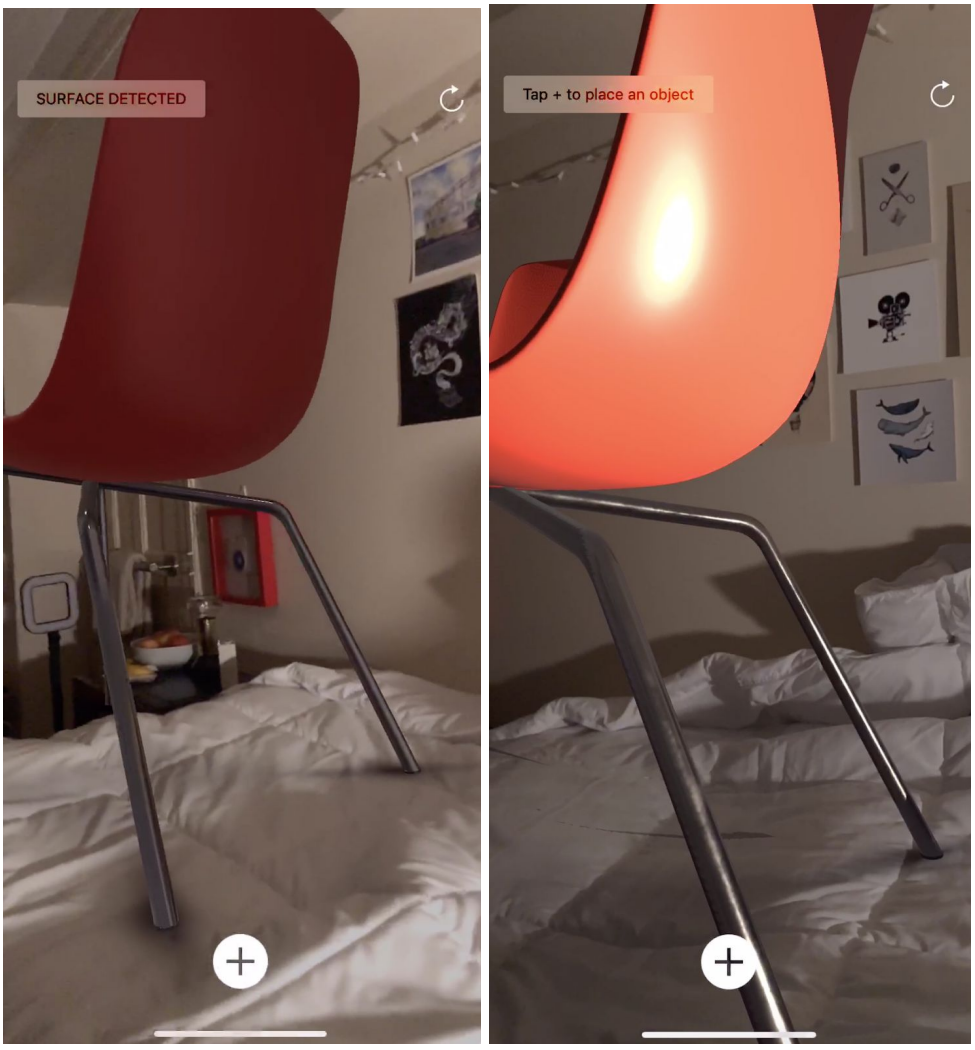

Figure 1: AR scene with a single spotlight in the environment. Notice how ARKit doesn't natively change the specular highlights of a virtual object with different environmental lighting (left image). With our approach (right image), we can model the spotlight placed in a scene to more realistically render the glossy surface.

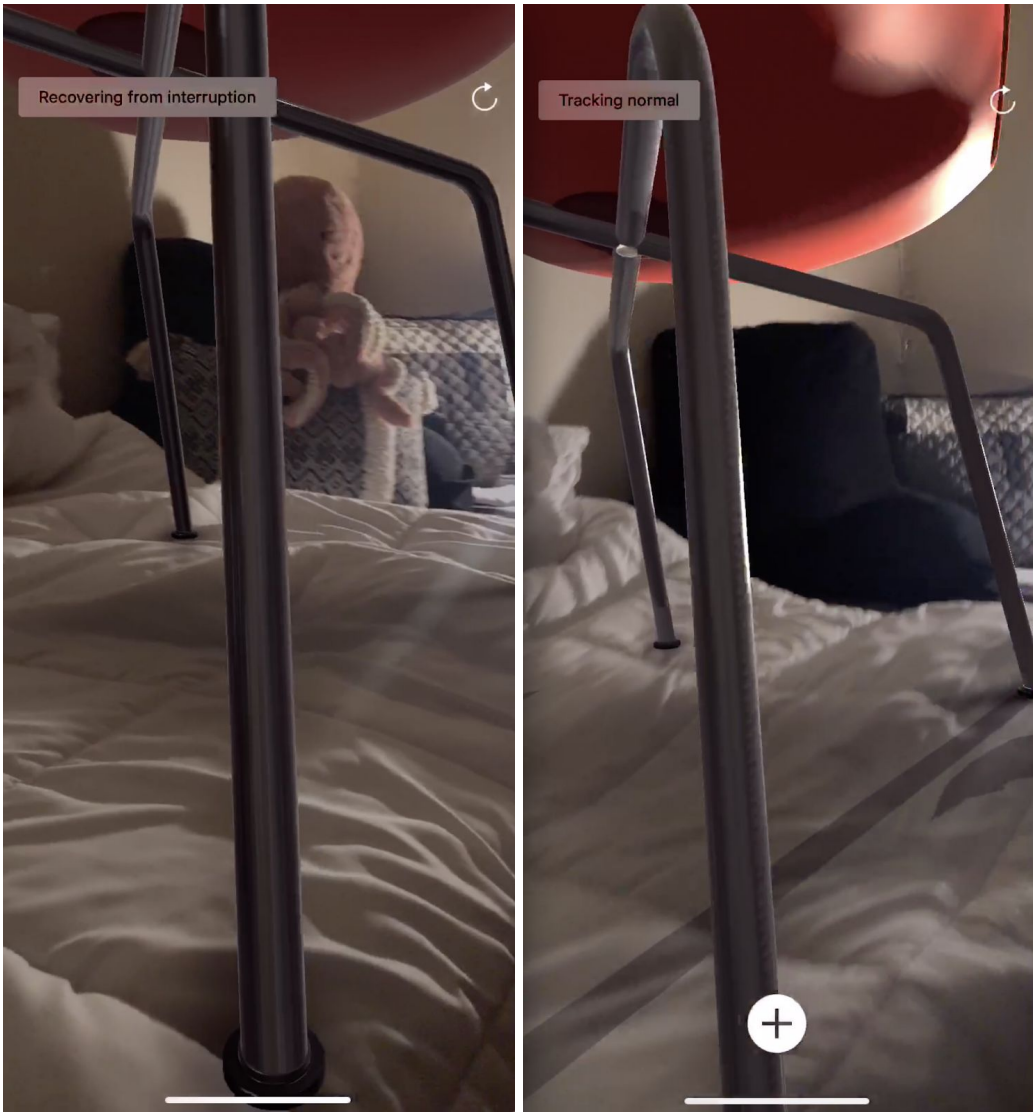

Figure 2: AR scene with a single spotlight in the environment (same scene as Figure 1). ARKit also doesn't model object shadows based on changes in environmental lighting (left image). Our approach accurately determines the position and direction of light sources to add shadows to objects in a scene (right image)

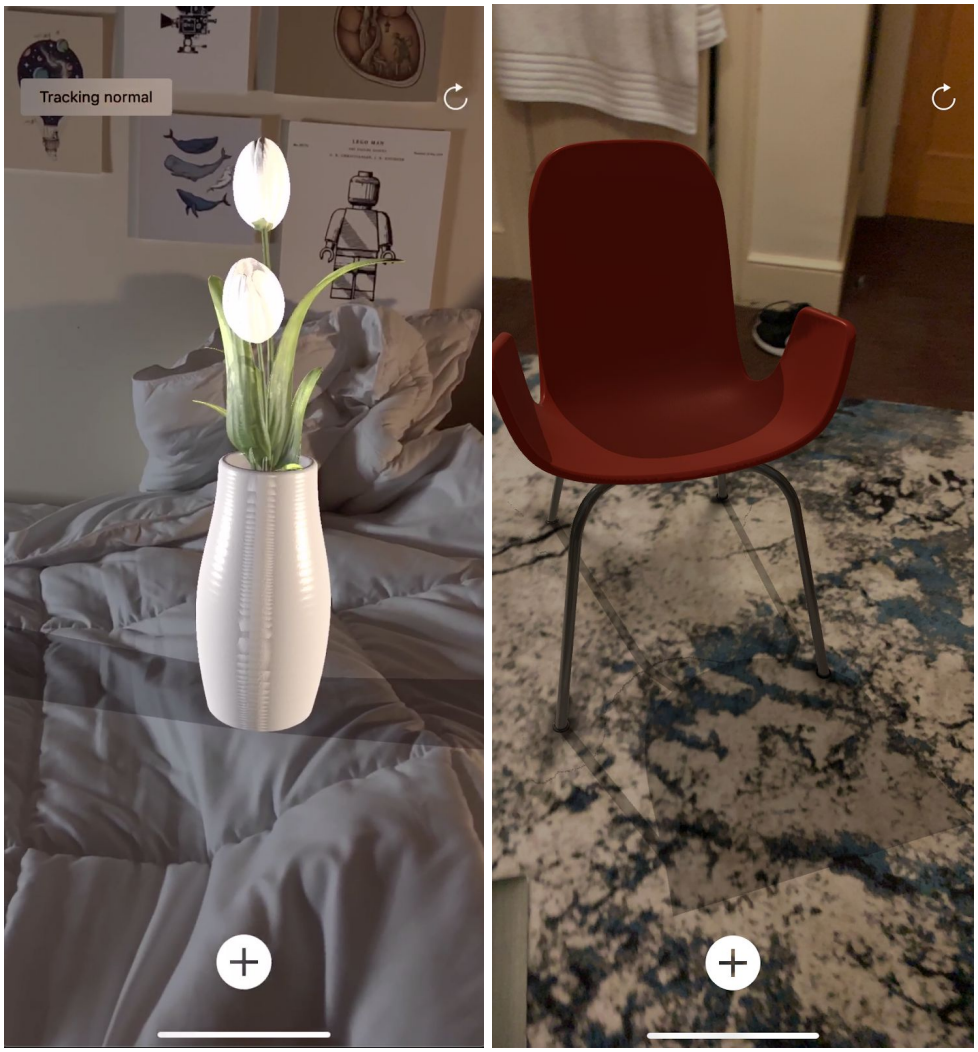

Figure 3: Additional features of our approach. Apart from single spotlights, our method also handles multiple light sources with different locations, directions, and temperatures — notice how the left half of the vase is lit with cooler light than the right half, and that the shadows point in both directions due to spotlights on either side (left image). We can also determine different types of lights, including directional light sources (ceiling lights) and render objects with accurate shadows (right image).