Sample Code

Visualizing and interacting with a reconstructed scene

Estimate the shape of the physical environment using a polygonal mesh.

Download

iOS 13.4+ | iPadOS 13.4+ | Xcode 16.0+

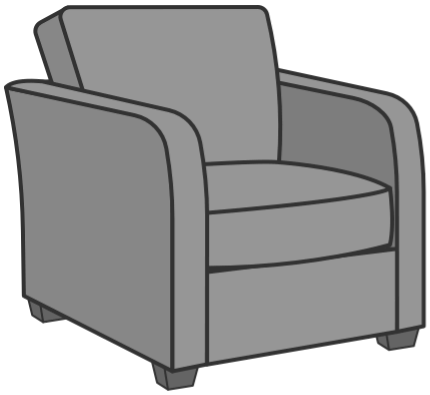
Overview

On a fourth-generation iPad Pro running iPadOS 13.4 or later, ARKit uses the LiDAR Scanner to create a polygonal model of the physical environment. The LiDAR Scanner quickly retrieves depth information in front of the person for ARKit to estimate and reconstruct the shape of the real world. ARKit converts the depth information into a series of vertices that connect to form a mesh. To partition the information, ARKit makes multiple anchors, each assigned a unique portion of the mesh. Collectively, the mesh anchors represent the real-world scene around the person.

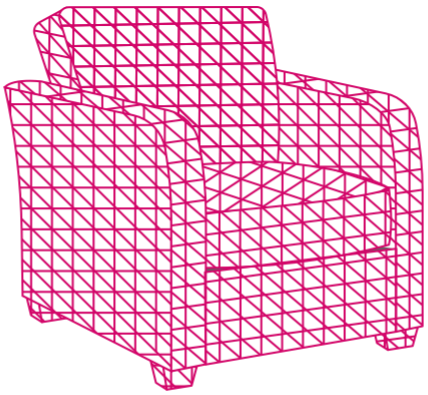
With these meshes, you can:

- Locate points on real-world surfaces more accurately.
- Classify real-world objects that ARKit can recognize.
- Occlude your app's virtual content with real-world objects that are in front of it.
- Enable virtual content to interact with the physical environment realistically. For example, by bouncing a virtual ball off a real-world wall or having the ball follow the laws of physics.
- Render shadows or re-light the real-world scene.

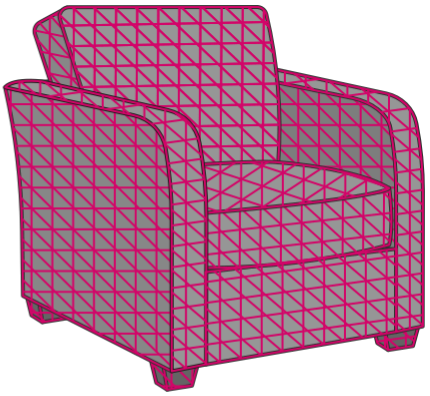
This sample app presents an AR experience using RealityKit. The figure below illustrates how RealityKit leverages real-world information from ARKit, and creates a debug visualization when you run this app and point the device at a chair in the real world.



Real-world chair



Mesh of the chair provided by ARKit



Debug visualization in RealityKit

Visualize the shape of the physical environment

To enable scene meshes, the sample sets a world-configuration's `sceneReconstruction` property to one of the mesh options.

```
arView.automaticallyConfigureSession = false
let configuration = ARWorldTrackingConfiguration()
configuration.sceneReconstruction = .meshWithClassification
```

The sample uses RealityKit's `ARView` to render its graphics. To visualize meshes at runtime, `ARView` offers the `sceneUnderstanding` debugging option.

```
arView.debugOptions.insert(.showSceneUnderstanding)
```

> **Note**
>
> The sample enables mesh visualization only to demonstrate the mesh feature; similarly, you normally enable mesh visualization only for debugging purposes.

To begin the AR experience, the sample configures and runs the session when the app first starts, in the main view controller's `viewDidLoad` callback.

```
arView.session.run(configuration)
```

Add plane detection

When an app enables plane detection with scene reconstruction, ARKit considers that information when making the mesh. Where the LiDAR scanner may produce a slightly uneven mesh on a real-world surface, ARKit flattens the mesh where it detects a plane on that surface.

To demonstrate the difference that plane detection makes on meshes, this app displays a toggle button. In the button handler, the sample adjusts the plane-detection configuration and restarts the session to effect the change.

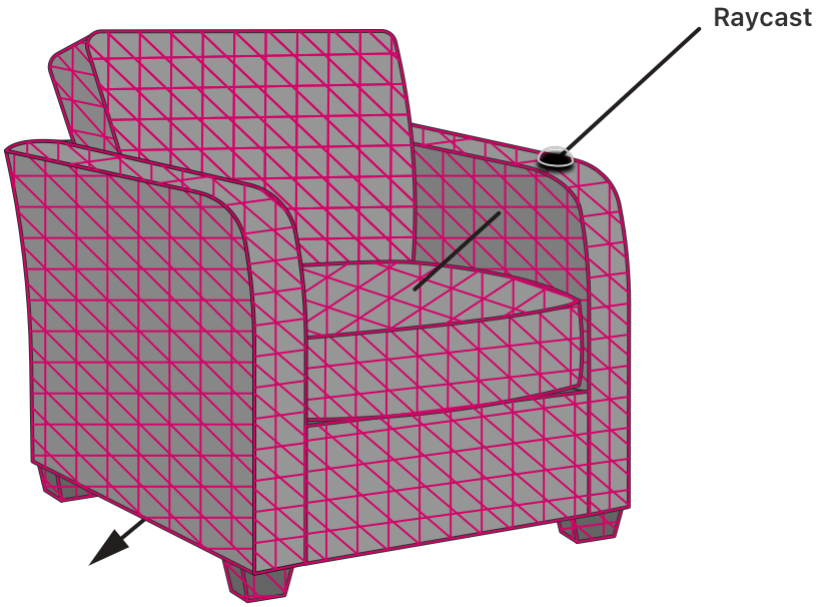
```
@IBAction func togglePlaneDetectionButtonPressed(_ button: UIButton) {
    guard let configuration = arView.session.configuration as? ARWorldTrackingConfig
        return
    }
    if configuration.planeDetection == [] {
        configuration.planeDetection = [.horizontal, .vertical]
        button.setTitle("Stop Plane Detection", for: [])
    } else {
        configuration.planeDetection = []
        button.setTitle("Start Plane Detection", for: [])
    }
    arView.session.run(configuration)
}
```

Locate a Point on an Object's Surface

Apps that retrieve surface locations using meshes can achieve unprecedented accuracy. By considering the mesh, raycasts can intersect with nonplanar surfaces, or surfaces with little or no features, like white walls.

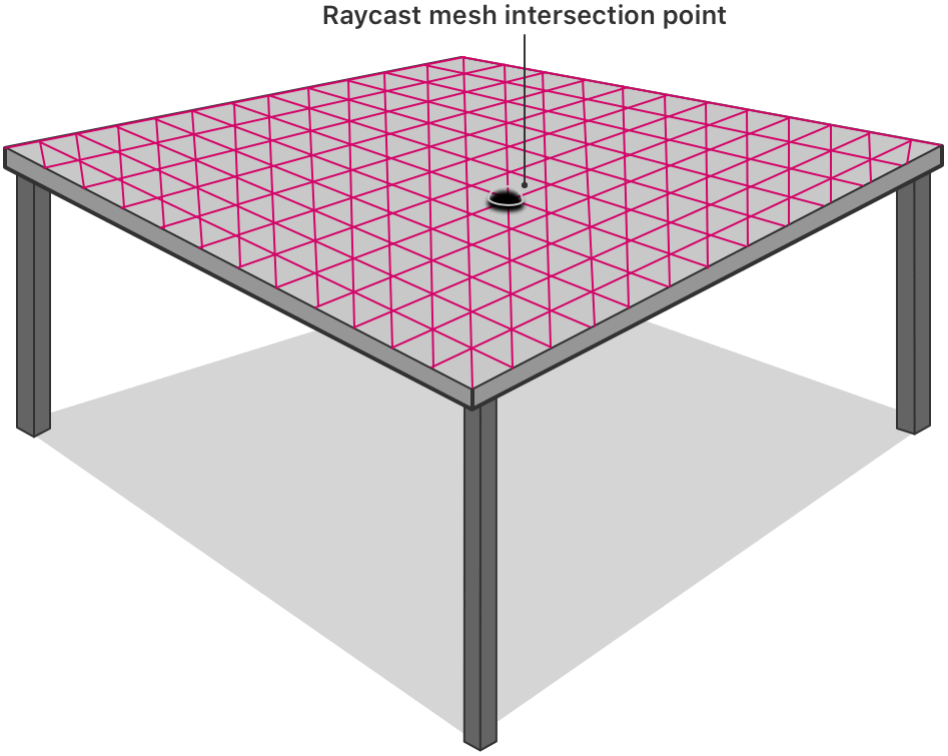
To demonstrate accurate raycast results, this app casts a ray when the user taps the screen. The sample specifies the `ARRaycastQuery.Target.estimatedPlane` allowable-target, and `ARRaycastQuery.TargetAlignment.any` alignment option, as required to retrieve a point on a meshed, real-world object.

```
if let result = arView.raycast(from: tapLocation, allowing: .estimatedPlane, alignme
    // ...
```



When the user's raycast returns a result, this app gives visual feedback by placing a small sphere at the intersection point.

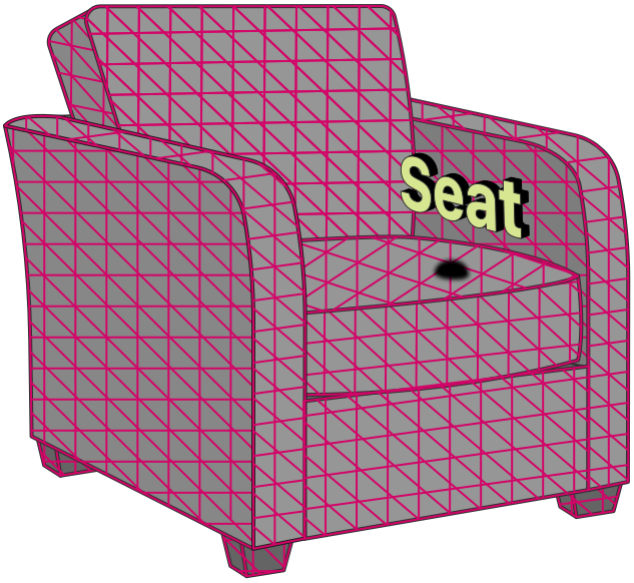
```
let resultAnchor = AnchorEntity(world: result.worldTransform)
resultAnchor.addChild(sphere(radius: 0.01, color: .lightGray))
arView.scene.addAnchor(resultAnchor, removeAfter: 3)
```



Classify Real-World Objects

ARKit has a classification feature that analyzes its meshed model of the world to recognize specific, real-world objects. Within the mesh, ARKit can classify floors, tables, seats, windows, and ceilings. See `ARMeshClassification` for the full list.

If the user taps the screen and the raycast intersects with a meshed, real-world object, this app displays text of the mesh's classification.



When the `automaticallyConfigureSession` property of `ARView` is `true`, RealityKit disables classification by default because it isn't required for occlusion and physics. To enable mesh classification, the sample overrides the default by setting the `sceneReconstruction` property to `meshWithClassification`.

```
arView.automaticallyConfigureSession = false
let configuration = ARWorldTrackingConfiguration()
configuration.sceneReconstruction = .meshWithClassification
```

This app attempts to retrieve a classification for the intersection point from the mesh.

```
nearbyFaceWithClassification(to: result.worldTransform.position) { (centerOfFace, cl
    // ...
```

The mesh consists of triangles, called *faces*. ARKit assigns a classification for each face, and the app searches through the mesh for a face near the intersection point. If the face has a classification, the app displays it onscreen. Because this routine involves extensive processing, the sample processes the mesh faces asynchronously, so the renderer doesn't stall.

```
DispatchQueue.global().async {
    for anchor in meshAnchors {
        for index in 0..<anchor.geometry.faces.count {
            // Get the center of the face so that we can compare it to the given loc
            let geometricCenterOfFace = anchor.geometry.centerOf(faceWithIndex: inde
```

```
            // Convert the face's center to world coordinates.
            var centerLocalTransform = matrix_identity_float4x4
            centerLocalTransform.columns.3 = SIMD4<Float>(geometricCenterOfFace.0, g
            let centerWorldPosition = (anchor.transform * centerLocalTransform).posi

            // We're interested in a classification that is sufficiently close to th
            let distanceToFace = distance(centerWorldPosition, location)
            if distanceToFace <= 0.05 {
                // Get the semantic classification of the face and finish the search
                let classification: ARMeshClassification = anchor.geometry.classific
                completionBlock(centerWorldPosition, classification)
                return
            }
        }
    }
```

With the classification in-hand, the sample creates 3D text to display it.

```
let textEntity = self.model(for: classification)
```

To prevent the mesh from partially occluding the text, the sample offsets the text slightly to help readability. The sample calculates the offset in the negative direction of the ray, which effectively moves the text slightly toward the camera, which is away from the surface.

```
let rayDirection = normalize(result.worldTransform.position - self.arView.cameraTran
let textPositionInWorldCoordinates = result.worldTransform.position - (rayDirection
```

To make the text always appear the same size on screen, the sample applies a scale based on text's distance from the camera.

```
let raycastDistance = distance(result.worldTransform.position, self.arView.cameraTra
textEntity.scale = .one * raycastDistance
```

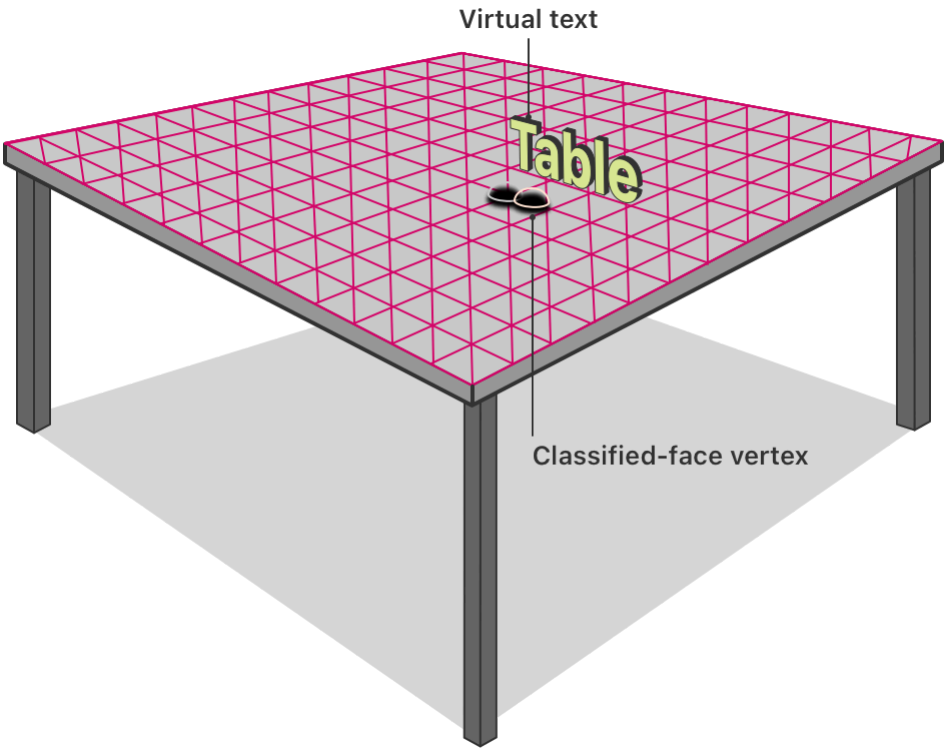
To display the text, the sample puts it in an anchored entity at the adjusted intersection-point, which is oriented to face the camera.

```
var resultWithCameraOrientation = self.arView.cameraTransform
resultWithCameraOrientation.translation = textPositionInWorldCoordinates
let textAnchor = AnchorEntity(world: resultWithCameraOrientation.matrix)
textAnchor.addChild(textEntity)
```

```
self.arView.scene.addAnchor(textAnchor, removeAfter: 3)
```

To visualize the location of the face's vertex from which the classification was retrieved, the sample creates a small sphere at the vertex's real-world position.

```
if let centerOfFace = centerOfFace {
    let faceAnchor = AnchorEntity(world: centerOfFace)
    faceAnchor.addChild(self.sphere(radius: 0.01, color: classification.color))
    self.arView.scene.addAnchor(faceAnchor, removeAfter: 3)
    // ...
}
```

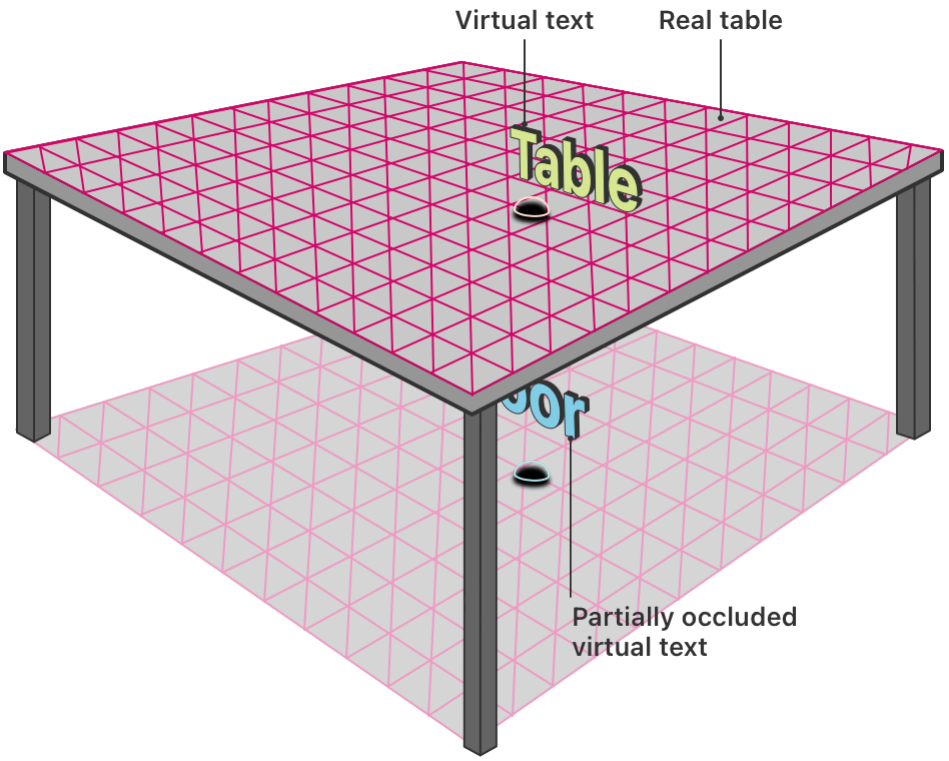


Occlude virtual content with a mesh

Occlusion is a feature where parts of the real world cover an app's virtual content, from the camera's perspective. To achieve this illusion, RealityKit checks for any meshes in front of virtual content, viewed by the user, and omits drawing any part of the virtual content obscured by those meshes. The sample enables occlusion by adding the `ARView.Environment.SceneUnderstanding.Options` option to the environment's `sceneUnderstanding` property.

```
arView.environment.sceneUnderstanding.options.insert(.occlusion)
```

At runtime, this app omits drawing portions of the virtual text that are behind any part of the meshed, real world.



Interact with real-world objects using physics

With scene meshes, virtual content can interact with the physical environment realistically because the meshes give RealityKit's physics engine an accurate model of the world. The sample enables physics by adding the `physics` option to the environment's `sceneUnderstanding` property.

```
arView.environment.sceneUnderstanding.options.insert(.physics)
```

To detect when virtual content comes in contact with a meshed, real-world object, the sample defines the text's proportions using a collision shape in the `addAnchor(_:,removeAfter:)` `Scene` extension.

```
if model.collision == nil {
    model.generateCollisionShapes(recursive: true)
    model.physicsBody = .init()
}
```

When this app classifies an object and displays some text, it waits three seconds before dropping the virtual text. When the sample sets the text's doc://com.apple.documentation/documentation/realitykit/modelentity/physicsBody's `mode` to `PhysicsBodyMode.dynamic`, the text reacts to gravity by falling.

```
Timer.scheduledTimer(withTimeInterval: seconds, repeats: false) { (timer) in
    model.physicsBody?.mode = .dynamic
}
```

As the text falls, it reacts when colliding with a meshed, real-world object, such as landing on the floor.

See Also

Occlusion

{} Occluding virtual content with people

Cover your app's virtual content with people that ARKit perceives in the camera feed.

{} Effecting People Occlusion in Custom Renderers

Occlude your app's virtual content where ARKit recognizes people in the camera feed by using matte generator.

`class ARMatteGenerator`

An object that creates matte textures you use to occlude your app's virtual content with people, that ARKit recognizes in the camera feed.