Sample Code

Obscuring virtual items in a scene behind real-world items

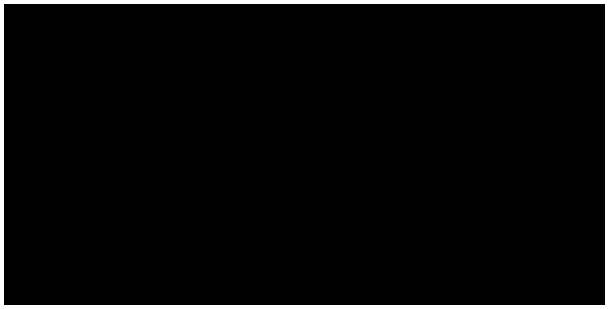
Increase the realism of an immersive experience by adding entities with invisible materials real-world objects.

Download

visionOS 2.0+ | Xcode 16.0+

Overview

You can create layered effects using an `OcclusionMaterial`, which is an invisible material that hides objects that render behind it. This sample project demonstrates how to hide entities behind real-world objects with ARKit by gathering live data about a person's surroundings and applyng the `OcclusionMaterial` to them.



Play

Capture the anchors from the scene

The sample uses the `MeshAnchorGenerator` class to retrieve anchor information from `SceneReconstructionProvider`. In the following code snippet, the generator takes in the root entity

from the reality view to perform actions on the entities and create a dictionary to store the collection of anchors:

```
import SwiftUI
import RealityKit
import ARKit

class MeshAnchorGenerator {
    /// The root to hold the meshes that the app detects.
    var root: Entity?

    /// The collection that comprises the `MeshAnchor.id` and the entity associated
    private var anchors: [UUID: Entity] = [:]

    init(root: Entity) {
        self.root = root
    }

    // ...
}
```

The `run(_:)` method processes all anchor updates asynchronously from the `SceneReconstructionProvider`. When an anchor detects either an `.added` or an `.updated` event, it creates a new entity if one isn't already present, and it updates its mesh, material, and transform properties:

```
@MainActor
func run(_ sceneRec: SceneReconstructionProvider) async {
    // Loop to process all anchor updates that the provider detects.
    for await update in sceneRec.anchorUpdates {
        switch update.event {
        case .added, .updated:
            // Retrieves the entity from the anchor collection based on the anchor I
            // If it doesn't exist, creates and adds a new entity to the collection.
            let entity = anchors[update.anchor.id] ?? {
                let entity = Entity()
                root?.addChild(entity)
                anchors[update.anchor.id] = entity
                return entity
            }()

            /// The occlusion material to apply to the entity.
```

```
            let material = OcclusionMaterial()

            /// The mesh based on the detected anchor.
            guard let mesh = try? await MeshResource(from: update.anchor) else { ret

            await MainActor.run {
                // Update the entity mesh and apply the occlusion material.
                entity.components.set(ModelComponent(mesh: mesh, materials: [materia

                // Set the transform matrix on its position relative to the anchor.
                entity.setTransformMatrix(update.anchor.originFromAnchorTransform, r
            }

            // ...
        }
    }
}
```

When an anchor detects a `.removed` event, the app removes the entity from the `root` and the anchor collection:

```
@MainActor
func run(_ sceneRec: SceneReconstructionProvider) async {
    for await update in sceneRec.anchorUpdates {
        // Handle different types of anchor update events.
        switch update.event {

        // ...

        case .removed:
            // Remove the entity from the root if it exists.
            anchors[update.anchor.id]?.removeFromParent()

            // Remove the anchor entry from the dictionary.
            anchors[update.anchor.id] = nil
        }
    }
}
```

Important

To use `SceneReconstructionProvider`, your app must add the `NSWorldSensingUsageDescription` property key list entry to the `info.plist`.

Start scene reconstruction

To track the anchors, the app starts an ARKit session. It uses `runSession(_:)` to initiate the `ARKitSession` with `SceneReconstructionProvider`, to perform scene reconstruction:

```
import RealityKit
import ARKit

func runSession(_ meshAnchors: Entity) {
    /// The ARKit session instance for scene reconstruction.
    let arSession = ARKitSession()

    /// The provider instance for scene reconstruction.
    let sceneReconstruction = SceneReconstructionProvider()

    Task {
        /// The generator to use to replace the mesh on anchors.
        let generator = MeshAnchorGenerator(root: meshAnchors)

        // Check if the device can support `SceneReconstructionProvider`.
        guard SceneReconstructionProvider.isSupported else {
            print("SceneReconstructionProvider is not supported on this device.")
            return
        }

        do {
            // Start the ARKit session to run the `SceneReconstructionProvider`.
            try await arSession.run([sceneReconstruction])
        } catch let error as ARKitSession.Error {
            // Handle any ARKit session errors.
            print("Encountered an error while running providers: \(error.localizedDe
        } catch let error {
            // Handle other unexpected errors.
            print("Encountered an unexpected error: \(error.localizedDescription)")
        }
```

```
        // Run the generator after the ARKit session runs successfully.
        await generator.run(sceneReconstruction)
    }
}
```

The method constructs the `MeshAnchorGenerator` class with the `meshAnchors`. Then it initiates the ARKit session while handling any potential errors. Finally, the app proceeds with the generation process by invoking the `run(_:)` method with the `sceneReconstruction` instance.

Setting up the drag gesture

In the immersive scene, the app loads a chair model that is moveable with drag gestures. The app creates the `translationGesture` to convert the person's gesture movements to the entity's positions, so that a person can target the chair model and move it around the space:

```
var translationGesture: some Gesture {
    DragGesture()
        .targetedToAnyEntity()
        .onChanged({ value in
            /// The entity that the drag gesture targets.
            let targetEntity = value.entity

            // Set `initialPosition` to the position of the entity if it is nil.
            if initialPosition == nil {
                initialPosition = targetEntity.position
            }

            // Convert the movement from the SwiftUI coordinate space to the reality
            let movement = value.convert(value.translation3D, from: .global, to: .sc

            // Apply the entity position to match the drag gesture,
            // and set the movement to stay at the ground level.
            targetEntity.position = (initialPosition ?? .zero) + movement.grounded
        })
        .onEnded({ _ in
            // Reset the `initialPosition` back to `nil` after the gesture ends.
            initialPosition = nil
        })
}
```

Put it all together into a view

The app loads all of the content into a reality view. It creates the `meshAnchors` entity and calls the `runSession(_:)` method with it, which initiates the ARKit session and adds child entities of anchors with occlusion materials to `meshAnchors`. Then the app adds the `meshAnchors` into the reality view:

```
import SwiftUI
import RealityKit

struct WorldOcclusionView: View {
    // ...

    var body: some View {
        RealityView { content in
            /// The entity to hold anchors with occlusion materials.
            let meshAnchors = Entity()

            // Run the ARKit session.
            runSession(meshAnchors)

            // Add the root to the reality view.
            content.add(meshAnchors)

            /// The filename of the chair model.
            let fileName: String = "chair_swan"

            // Load the chair model from the filename asynchronously.
            guard let chair = try? await ModelEntity(named: fileName) else {
                assertionFailure("Failed to load model: \(fileName)")
                return
            }

            // Generate collision shapes to the chair for proper occlusion.
            chair.generateCollisionShapes(recursive: true)

            // Enable inputs for the app to detect the hand gestures.
            chair.components.set(InputTargetComponent())

            // Add the chair entity to the reality view.
            content.add(chair)
        }
        // Set the `translationGesture` to the view.
        .gesture(translationGesture)
```

```
    }
}
```

Finally, the app creates a chair model and sets the chair to accept user input with `InputTargetComponent`. Then it sets the chair with `TargetModelComponent`, enabling the chair to work with `translationGesture`. The app then adds the chair model to the reality view.

See Also

Integrating ARKit

{} Creating a 3D painting space

Implement a painting canvas entity, and update its mesh to represent a stroke.

{} Tracking and visualizing hand movement

Use hand-tracking anchors to display a visual representation of hand transforms in visionOS.

{} Displaying an entity that follows a person's view

Create an entity that tracks and follows head movement in an immersive scene.

{} Applying mesh to real-world surroundings

Add a layer of mesh to objects in the real world, using scene reconstruction in ARKit.