Sample Code

Applying mesh to real-world surroundings

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

Download

visionOS 2.0+ | Xcode 16.0+

Overview

With ARKit in visionOS, you can create immersive experiences that integrate with real-world environments. This sample demonstrates how to use ARKit's scene-reconstruction capability to construct and display a layer of mesh on real-world objects, like the scene in the following video:

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:

```swift
import SwiftUI
import RealityKit
import ARKit

class MeshAnchorGenerator {
    /// The root entity of the view.
    var root: Entity

    /// The collection of anchors.
    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:

```swift
@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:
            // Retrives the entity from the anchor collection based on the anchor II
            // 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 material for the mesh to update.
            let material = SimpleMaterial(color: .cyan.withAlphaComponent(0.8), isMe
            /// The mesh from an anchor.
            guard let mesh = try? await MeshResource(from: update.anchor) else { ret

            await MainActor.run {
                // Update the entity mesh and apply the 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 {
        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 the `SceneReconstructionProvider`, you must add the `NSWorldSensingUsageDescription` property key list entry to the `info.plist`.

Add scene reconstruction to the view

To track the anchors, the app must start an `ARKitSession`. The app creates the `root` to store all the entities, the `ARKitSession` instance, and the `SceneReconstructionProvider` instance to perform scene reconstruction:

```
import SwiftUI
import RealityKit

struct SceneReconstructionView: View {
    var body: some View {
        RealityView { content in
            /// The root entity.
            let root = Entity()

            /// The `ARKitSession` instance for scene reconstruction.
            let arSession = ARKitSession()

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

            // ...
        }
    }
}
```

When the properties are in place, the app constructs the `MeshAnchorGenerator` class with the root. It initiates the `ARKitSession` while handling any potential errors. Finally, the app proceeds with the generation process by invoking the `run(_:)` method with the `sceneReconstruction` instance:

```
import SwiftUI
import RealityKit

struct SceneReconstructionView: View {
    var body: some View {
        RealityView { content in

            // ...

            Task {
                /// The generator to store the root with unlit materials.
                let generator = MeshAnchorGenerator(root: root)

                // Check if the device supports scene reconstruction.
                guard SceneReconstructionProvider.isSupported else {
                    print("SceneReconstructionProvider is not supported on this devi
                    return
                }

                do {
                    // Start the `ARKitSession` and run the `SceneReconstructionProv
                    try await arSession.run([sceneReconstruction])
                } catch let error as ARKitSession.Error {
                    // Handle any `ARKitSession` errors.
                    print("Encountered an error while running providers: \(error.loc
                } catch let error {
                    // Handle other errors.
                    print("Encountered an unexpected error: \(error.localizedDescrip
                }

                // Start the generator if the session runs successfully.
                await generator.run(sceneReconstruction)
            }

            // Add the root entity to the `RealityView`.
            content.add(root)
        }
    }
}
```

Set up the immersive space

For ARKit and `SceneReconstructionProvider` to work, the app requires an immersive space. The sample creates an immersive scene in the app's entry point:

```
import SwiftUI

@main
struct EntryPoint: App {
    var body: some Scene {
        WindowGroup {
            MainView()
        }
        ImmersiveSpace(id: "SceneReconstruction") {
            SceneReconstructionView()
        }
    }
}
```

The `MainView` launches the immersive space with `openImmersiveSpace` after it appears:

```
import SwiftUI

struct MainView: View {
    @Environment(\.openImmersiveSpace) var openImmersiveSpace

    var body: some View {
        Text("Scene Reconstruction Example")
            .onAppear {
                Task {
                    await openImmersiveSpace(id: "SceneReconstruction")
                }
            }
    }
}
```

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.

{} 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.