Sample Code

Creating a 3D painting space

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

Download

visionOS 2.0+ | Xcode 16.0+

Overview

This sample demonstrates how to create a painting space so that people can pinch-to-draw in an augmented reality space. To achieve this, the app uses hand-tracking technology that ARKit provides to monitor a person's hand movements. To capture and display each stroke, the app creates collision boxes within the environment, stores the person's drawing points through a pinch gesture, and then updates the mesh at those points to represent the stroke, as the following video shows:

Play

Add a system and component to enable real-time updates

The sample uses a custom system and component to handle updates for entities over time:

```
import SwiftUI
import RealityKit
```

```
struct ClosureComponent: Component {
    /// The closure that takes the time interval since the last update.
    let closure: (TimeInterval) -> Void

    init(closure: @escaping (TimeInterval) -> Void) {
        self.closure = closure
        ClosureSystem.registerSystem()
    }
}
```

The component contains the `closure` variable to track the time. On initialization, it registers `ClosureSystem` into the reality view.

The `ClosureSystem` constructs a query using the `EntityQuery` to retrieve all entities with the `ClosureComponent` from the scene. Then it passes the delta time, which is the elapsed time since the last update, to the `closure` variable for each entity:

```
import SwiftUI
import RealityKit

struct ClosureSystem: System {
    /// The query to find entities that contain `ClosureComponent`.
    static let query = EntityQuery(where: .has(ClosureComponent.self))

    init(scene: RealityKit.Scene) {}

    /// Update entities with `ClosureComponent` at each render frame.
    func update(context: SceneUpdateContext) {
        for entity in context.entities(matching: Self.query, updatingSystemWhen: .re
            guard let comp = entity.components[ClosureComponent.self] else { continu
            comp.closure(context.deltaTime)
        }
    }
}
```

Set up hand tracking for painting action

The sample creates `PaintingHandTracking` to track the person's hand with ARKit and store the latest detected hand anchors in the `latestLeftHand` and `latestRightHand` properties:

```swift
import RealityKit
import ARKit

@MainActor class PaintingHandTracking: ObservableObject {
    /// The ARKit session for hand tracking.
    let arSession = ARKitSession()

    /// The `HandTrackingProvider` for hand tracking.
    let handTracking = HandTrackingProvider()

    /// The current left hand anchor that the app detects.
    @Published var latestLeftHand: HandAnchor?

    /// The current right hand anchor that the app detects.
    @Published var latestRightHand: HandAnchor?

    // ...
}
```

The `startTracking()` method checks device compatibility for hand tracking and starts the `ARKitSession` with the `HandTrackingProvider`, while handling potential errors. It continuously waits for anchor updates and assigns the latest left or right hand based on the chirality value:

```swift
import RealityKit
import ARKit

@MainActor class PaintingHandTracking: ObservableObject {
    // ...

    /// Check whether the device supports hand tracking, and start the ARKit sessior
    func startTracking() async {
        guard HandTrackingProvider.isSupported else {
            print("HandTrackingProvider is not supported on this device.")
            return
        }

        do {
            try await arSession.run([handTracking])
        } catch let error as ARKitSession.Error {
            print("Encountered an error while running providers: \(error.localizedDe
        } catch let error {
            print("Encountered an unexpected error: \(error.localizedDescription)")
        }

        // Assign the left and right hand based on the anchor updates.
        for await anchorUpdate in handTracking.anchorUpdates {
            switch anchorUpdate.anchor.chirality {
            case .left:
                self.latestLeftHand = anchorUpdate.anchor
            case .right:
                self.latestRightHand = anchorUpdate.anchor
            }
        }
    }
}
```

Implement the structural representation of a stroke

The sample creates the `Stroke` structure to represent the current stroke the person creates with the drag gesture input:

```
import SwiftUI
import RealityKit

struct Stroke {
    /// The stroke that represents the stroke.
    var entity = Entity()

    /// The collection of points in 3D space that represent the stroke.
    var points: [SIMD3<Float>] = []

    /// The maximum radius of the stroke.
    let maxRadius: Float = 1E-2

    /// The number of points in each ring of the mesh.
    let pointsPerRing = 8

    // ...
}
```

To update the mesh of the stroke, use `updateMesh()` in the `Stroke` structure. This method uses the initial point as the center and initializes the `positions`, `normal`, and `triangle` indices from `generateMeshData()`, which iterates through the `points` array to construct a path for the mesh:

```swift
func updateMesh() {
    // The starting point where the stroke mesh begins.
    guard let center = points.first else { return }

    /// The position, normals, and triangle indices that the points generate.
    let (positions, normals, triangles) = generateMeshData()

    // ...
}
```

The method creates the MeshResource.Contents to represent the content of the mesh that the stroke updates:

```swift
func updateMesh() {
    // ...

    /// The `MeshResource.Contents` instance.
    var contents = MeshResource.Contents()

    // Create and assign an instance to `contents`.
    contents.instances = [MeshResource.Instance(id: "main", model: "model")]

    // Create the part for the model, and set the vertex positions, triangle indices
    var part = MeshResource.Part(id: "part", materialIndex: 0)
    part.positions = MeshBuffer(positions)
    part.triangleIndices = MeshBuffer(triangles)
    part.normals = MeshBuffer(normals)

    // Create and assign a model that consists of the `part`.
    contents.models = [MeshResource.Model(id: "model", parts: [part])]

    // ...
}
```

The method creates a MeshResource.Instance, then define a MeshResource.Part for the model, where MeshBuffer assigns vertex positions, triangle indices, and normals. Finally, it creates a MeshResource.Model with `part` and assigns to `contents`.

The method either updates an existing mesh component on the entity, or creates a new mesh. If a mesh component already exists, it updates it with the `contents`; otherwise, it generates a new mesh with `contents` and assigns it to the entity:

```swift
func updateMesh() {
    // ...

    // Replace the mesh with `contents` if there is a mesh component on the entity.
    if let mesh = entity.model?.mesh {
        do {
            try mesh.replace(with: contents)
        } catch {
            print("Error replacing mesh: \(error.localizedDescription)")
        }
    } else {
        /// The new mesh that generates with `content`.
        guard let mesh = try? MeshResource.generate(from: contents) else {
        print("Error generating mesh")
            return
        }

        // Set the model component to the new mesh and assign a simple material.
        entity.components.set(ModelComponent(
            mesh: mesh,
            materials: [SimpleMaterial(color: .white, roughness: 1.0, isMetallic: fa
        ))

        // Set the entity's transform and position.
        entity.setTransformMatrix(.identity, relativeTo: nil)
        entity.setPosition(center, relativeTo: nil)
    }
}
```

Create the painting canvas to store strokes

To create the 3D painting environment, the sample uses `PaintingCanvas` to set up `root`, which represents the painting canvas, and `currentStroke`, which represents the stroke the person creates:

```swift
import SwiftUI
import RealityKit
```

```swift
class PaintingCanvas {
    /// The main root entity for the painting canvas.
    let root = Entity()

    /// The stroke the person creates.
    var currentStroke: Stroke?

    /// The distance for the box that extends in the positive direction.
    let big: Float = 1E2

    /// The distance for the box that extends in the negative direction.
    let small: Float = 1E-2

    init() {
        root.addChild(addBox(size: [big, big, small], position: [0, 0, -0.5 * big])
        root.addChild(addBox(size: [big, big, small], position: [0, 0, +0.5 * big])

        // ...
    }

    /// Create a collision box that takes in user input with the drag gesture.
    private func addBox(size: SIMD3<Float>, position: SIMD3<Float>) -> Entity {
        let box = Entity()
        box.components.set(InputTargetComponent())
        box.components.set(CollisionComponent(shapes: [.generateBox(size: size)], is
        box.position = position
        return box
    }

    // ...
}
```

After the initialization of the class, the app sets up the root by adding six boxes to represent the canvas. User input can target these boxes, and the boxes can interact with other collision entities, stacking along the x, y, and z-axes.

Use `addPoint(_:)` to add the points that the person targets into the `currentStroke`, and update the stroke's mesh to display the stroke on the painting canvas:

```swift
class PaintingCanvas {
    // ...
    func addPoint(_ position: SIMD3<Float>) {
        /// The maximum distance between two points before requiring a new point.
        let threshold: Float = 1E-9

        // Start a new stroke if no stroke exists.
        if currentStroke == nil {
            currentStroke = Stroke()

            // Add the stroke to the root.
            root.addChild(currentStroke!.entity)
        }

        // Check if the length between the current hand position and the previous po
        if let previousPoint = currentStroke?.points.last, length(position - previou
            return
        }

        // Add the current position to the stroke.
        currentStroke?.points.append(position)

        // Update the current stroke mesh.
        currentStroke?.updateMesh()
    }

    func finishStroke() {
        if let stroke = currentStroke {
            // Trigger the update mesh operation.
            stroke.updateMesh()

            // Clear the current stroke.
            currentStroke = nil
        }
    }
}
```

When the person ends a painting action, `finishStroke()` updates the stroke's mesh and sets the `currentStroke` to nil, to mark the end of the stroke.

Set up the painting space

The sample creates a `PaintingView` to combine the hand tracking, painting canvas, and drag gesture detection. It creates the instance of the `PaintingHandTracking` class, the instance of the PaintingCanvas class, and the lastIndexPose to store the last position of the index finger:

```swift
import SwiftUI
import RealityKit
import ARKit

struct PaintingView: View {
    /// The `PaintingHandTracking` class instance.
    var paintingHandTracking = PaintingHandTracking()

    /// The instance of the `PaintingCanvas` class to handle painting operations.
    @State var canvas = PaintingCanvas()

    /// The last position of the index finger.
    @State var lastIndexPose: SIMD3<Float>?

    // ...
}
```

Within the main body of the view, the app attaches a ClosureComponent to the root entity, allowing the hand anchors to update over time:

```swift
var body: some View {
    RealityView { content in
        /// The root entity from the painting canvas.
        let root = canvas.root
        content.add(root)

        root.components.set(ClosureComponent(closure: { deltaTime in
            /// The collection of `HandAnchor` instances.
            var anchors = [HandAnchor]()

            if let latestLeftHand = paintingHandTracking.latestLeftHand {
                anchors.append(latestLeftHand)
            }
            if let latestRightHand = paintingHandTracking.latestRightHand {
                anchors.append(latestRightHand)
            }

            // Loop through each anchor that the app detects.
            for anchor in anchors {
                guard let handSkeleton = anchor.handSkeleton else {
                    continue
                }

                let thumbPos = (anchor.originFromAnchorTransform * handSkeleton.joir
                let indexPos = (anchor.originFromAnchorTransform * handSkeleton.joir

                /// The threshold to check whether the index and thumb are close.
                let pinchThreshold: Float = 0.05
                if length(thumbPos - indexPos) < pinchThreshold {
                    lastIndexPose = indexPos
                }
            }
        }))
    }

    // ...
}
```

The `ClosureComponent` iterates through the collection of hand anchors and attempts to retrieve the hand skeleton with each anchor. Then it calculates the distance between the tip of the thumb and the tip of the index finger, using the data from the hand skeleton. If this distance is less than the defined threshold for a pinch gesture, it updates the last known index finger position.

The app attaches the main body view to the `.gesture()` modifier, to recognize user inputs. When the `DragGesture` recognizes an `.onChanged` action, it interprets this as the person initiating a painting action, adding a point to the canvas. When the `DragGesture` recognizes an `.onEnded` action, it interprets this as the person finishing a painting action, ending the stroke on the canvas:

```
.gesture(
    DragGesture(minimumDistance: 0)
    // Enable the gesture to target an entity.
    .targetedToAnyEntity()
    .onChanged({ _ in
        // Get the current position.
        if let pos = lastIndexPose {
            // Add a point at the current position.
            canvas.addPoint(pos)
        }
    })
    .onEnded({ _ in
        // End the current stroke when the drag gesture ends.
        canvas.finishStroke()
    })
)
.task {
    // Enable hand tracking when the view starts.
    await paintingHandTracking.startTracking()
}
```

The app starts the hand-tracking session within the `.task()` modifier, which allows the app to enable hand tracking asynchronously before the view appears.

Related samples

{} Creating a spatial drawing app with RealityKit

Use low-level mesh and texture APIs to achieve fast updates to a person's brush strokes by integrating RealityKit with ARKit and SwiftUI.

{} 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.