Sample Code

Creating screen annotations for objects in an AR experience

Annotate an AR experience with virtual sticky notes that you display onscreen over real and virtual objects.

Download

iOS 13.0+ | iPadOS 13.0+ | Xcode 16.0+

Overview

At times, the user may want to annotate real or virtual objects in your AR experience. For example, they might want to place a virtual name plate on paintings at a museum. By fixing annotations to the screen, you enable the user to annotate their AR experience in *screen space*. To demonstrate screen-space annotations, this sample app enables the capability to tap the screen to place one or more virtual sticky notes with text in the real world.

Text displayed in screen space remains readable at all viewing angles and distances. The sample app implements sticky notes using a `UITextView` that's flush with the screen, which allows the user to quickly define the note's text using regular touch input. Using UIKit to annotate an AR experience also provides the benefits of localization and accessibility.

> **Note**
>
> This sample uses RealityKit to anchor virtual content in the real world. RealityKit requires iOS 13. ARKit is not available in iOS Simulator.

To display text that's anchored in world space instead, see Recognizing and Labeling Arbitrary Objects.

Resolve the User’s Tap to a 3D Location

To annotate an object in an AR experience, you first determine where it is in the physical environment. This sample app enables the user to tap the screen to place a sticky note by first adding a tap gesture recognizer to the view.

```
func arViewGestureSetup() {
    let tapGesture = UITapGestureRecognizer(target: self, action: #selector(tappedOn
    arView.addGestureRecognizer(tapGesture)

    let swipeGesture = UISwipeGestureRecognizer(target: self, action: #selector(swip
    swipeGesture.direction = .down
    arView.addGestureRecognizer(swipeGesture)
}
```

When the input handler is called, you read the tap screen coordinates by calling `location(in:)`.

```
let touchLocation = sender.location(in: arView)
```

To get a 3D world position that corresponds to the tap location, cast a ray from the camera’s origin through the touch location to check for intersection with any real-world surfaces along that ray.

```
guard let raycastResult = arView.raycast(from: touchLocation, allowing: .estimatedPl
    messageLabel.displayMessage("No surface detected, try getting closer.", duration
    return
}
```

If ARKit finds a planar surface where the user tapped, the ray-cast result provides you the 3D intersection point in `worldTransform`.

Anchor a sticky note in the environment

To keep track of a real-world location, you create an anchor positioned there. RealityKit implements an anchor as an `Entity` conforming to `HasAnchoring`. Thus, you implement those protocols when designing a sticky note in RealityKit.

```
class StickyNoteEntity: Entity, HasAnchoring, HasScreenSpaceView {
    // ...
```

Create the entity by calling its initializer and passing in the ray-cast result's `worldTransform`.

```
let note = StickyNoteEntity(frame: frame, worldTransform: raycastResult.worldTransfo
```

In the sticky note entity's `init` function, position the entity at the tap location by setting its transformation matrix to the argument `worldTransform`.

```
init(frame: CGRect, worldTransform: simd_float4x4) {
    super.init()
    self.transform.matrix = worldTransform
    // ...
```

Let RealityKit know about your entity by adding it to the scene hierarchy. RealityKit then registers an `ARAnchor` for your entity with ARKit.

```
// Add the sticky note to the scene's entity hierarchy.
arView.scene.addAnchor(note)
```

Display the sticky note's annotation

For the purposes of this sample app, the sticky note entity has no geometry and thus, no appearance. Its anchor provides a 3D location only, and it's the sticky note's screen-space annotation that has an appearance. To display it, you define the sticky note's annotation. Following RealityKit's entity-component model, design a component that houses the annotation, which in this case is a view. See `ScreenSpaceComponent.swift`.

```
struct ScreenSpaceComponent: Component {
    var view: StickyNoteView?
    //...
```

As a prepackaged UI element that renders text for you, `UITextView` is useful as a screen-space annotation.

```
class StickyNoteView: UIView {
    var textView: UITextView!
    // ...
```

Expose the screen-space component in its own protocol.

```
protocol HasScreenSpaceView: Entity {
    var screenSpaceComponent: ScreenSpaceComponent { get set }
}
```

Implement the protocol in your entity; see `StickyNoteEntity.swift`.

```
class StickyNoteEntity: Entity, HasAnchoring, HasScreenSpaceView {
    // ...
```

To display the entity's annotation, add the sticky-note view to the view hierarchy.

```
// Add the sticky note's view to the view hierarchy.
guard let stickyView = note.view else { return }
arView.insertSubview(stickyView, belowSubview: trashZone)
```

To put the annotation in the right place on the screen, ask the `ARView` to convert its entity's world location to a 2D screen point.

```
guard let projectedPoint = arView.project(note.position) else { return }
```

To enhance visual accuracy, center the note's view around the anchor's projected world location.

```
setPositionCenter(projectedPoint)
```

To do that, calculate the midpoint and set the view's origin.

```
view.frame.origin = CGPoint(x: centerPoint.x, y: centerPoint.y)
```

Update the annotation's position

Because users move their device during an AR experience, the annotation's screen position quickly becomes out of sync with its anchor's world position. To keep the annotation's screen position accurate, call `ARView`'s `project` function every frame, updating the annotation's position with the result.

```
// Updates the screen position of the note based on its visibility
note.projection = Projection(projectedPoint: projectedPoint, isVisible: isVisible)
note.updateScreenPosition()
```

Handle user interaction

A benefit of using `UIView` types for screen annotations is that they simplify user interaction. The sample implements sticky notes using `UITextView`, which enables users to more easily edit their text. The sample implements minimal gesture recognizer code to manage sticky notes.

The following code enables the capability to create a note by tapping the screen.

```
@objc
func tappedOnARView(_ sender: UITapGestureRecognizer) {

    // Ignore the tap if the user is editing a sticky note.
    for note in stickyNotes where note.isEditing { return }

    // Create a new sticky note at the tap location.
    insertNewSticky(sender)
}
```

By implementing its own tap gesture recognizer to control editing, `UITextView` enables the user to tap an existing note to edit its text. To be notified when the user edits a note, override `UITextView`'s `textViewDidBeginEditing(_ textView:)` delegate callback.

```
extension ViewController: UITextViewDelegate {

    // - Tag: TextViewDidBeginEditing
    func textViewDidBeginEditing(_ textView: UITextView) {


        // Get the main view for this sticky note.
        guard let stickyView = textView.firstSuperViewOfType(StickyNoteView.self) el
        // ...

```

The following code enables the capability to move a note by panning the screen.

```
@objc
func panOnStickyView(_ sender: UIPanGestureRecognizer) {

    guard let stickyView = sender.view as? StickyNoteView else { return }

    let panLocation = sender.location(in: arView)

    // Ignore the pan if any StickyViews are being edited.
    for note in stickyNotes where note.isEditing { return }

    panStickyNote(sender, stickyView, panLocation)
}
```

When the user pans to reposition a sticky note, you convert the screen touch location to a 3D world position using `raycast(from:allowing:alignment:)`. The user can then reposition the sticky note's anchor in the real world versus simply moving the annotation to a new arbitrary screen location. If a ray cast from the final screen location in the pan gesture doesn't produce an intersection with a 3D world location, don't move the sticky note there.

```
fileprivate func attemptRepositioning(_ stickyView: StickyNoteView) {
    // Conducts a ray-cast for feature points using the panned position of the Stick
    let point = CGPoint(x: stickyView.frame.midX, y: stickyView.frame.midY)
    if let result = arView.raycast(from: point, allowing: .estimatedPlane, alignment
        stickyView.stickyNote.transform.matrix = result.worldTransform
    } else {
        messageLabel.displayMessage("No surface detected, unable to reposition note.
        stickyView.stickyNote.shouldAnimate = true
    }
}
```

The following portion of the pan gesture handler enables the capability to remove a sticky note when the user drags it to the text that says "delete" at the top of the screen.

```
if stickyView.isInTrashZone {
    deleteStickyNote(stickyView.stickyNote)
    // ...
```

Enhance the experience with animation

Keeping screen-space annotations to a minimum will maximize the user's immersion in the AR experience. The sample app makes sticky notes small when the user isn't editing text, minimizing distractions so they can focus on the real-world environment. But for similar reasons, you should enlarge a sticky note when the user is editing text. To create a seamless transition between editing and nonediting states, animate the sticky note’s size instead of changing it abruptly. See the `animateStickyViewToEditingFrame` function.

```
func animateStickyViewToEditingFrame(_ stickyView: StickyNoteView, keyboardHeight: D
    let safeFrame = view.safeAreaLayoutGuide.layoutFrame
    let height = safeFrame.height - keyboardHeight
    let inset = height * 0.05
    let editingFrame = CGRect(origin: safeFrame.origin, size: CGSize(width: safeFram
    UIViewPropertyAnimator(duration: 0.2, curve: .easeIn) {
        stickyView.frame = editingFrame
        //...
```

Bring even more focus to the editing experience by dimming the background and by lighting the sticky note that the user is editing.

```
stickyView.blurView.effect = UIBlurEffect(style: .light)
```

To prevent the user from losing track of a sticky note’s real-world location, animate the note smoothly from one position to the next. For example, if an annotation fails to reposition, animate the sticky note back to its original screen position. This increases the user’s ability to track the annotation if they want to try moving it again.

```
if shouldAnimate {
    animateTo(projectedPoint)
    // ...
```

To animate the note’s movement, you continually set its location using a `UIViewPropertyAnimator`.

```
func animateTo(_ point: CGPoint) {

    let animator = UIViewPropertyAnimator(duration: 0.3, curve: .linear) {
        self.setPositionCenter(point)
    }
    // ...
```

See Also

Text Annotations

{} Recognizing and Labeling Arbitrary Objects

Create anchors that track objects you recognize in the camera feed, using a custom optical-recognition algorithm.