Sample Code

Rendering hover effects in Metal immersive apps

Change the appearance of a rendered onscreen element when a player gazes at it.

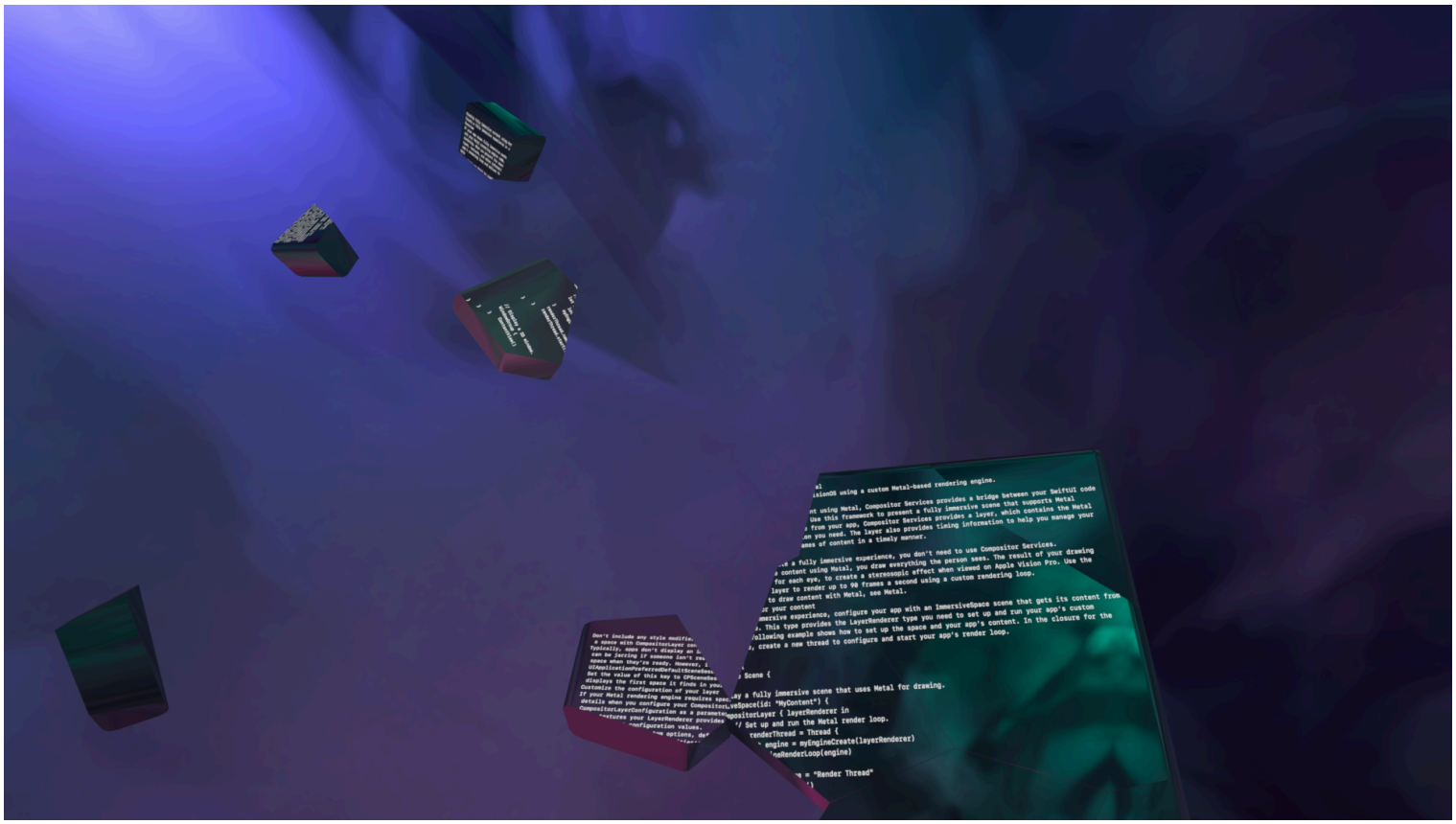
Download

macOS 26.0+ | visionOS 26.0+ | Xcode 26.0+

Overview

In visionOS, both SwiftUI views and RealityKit entities can take advantage of *hover effects*, which change the appearance of a rendered onscreen element when a player gazes at it or highlights it using assistive technologies. In visionOS 26, fully immersive apps that render their own content using Metal can also use hover effects.

This sample code project demonstrates how to pass in uniforms and attributes to your Metal shaders so your app can implement system-provided hover effects in a privacy-preserving way. On launch, the app opens to an immersive virtual space with a large shape that shatters into several pieces. If a player looks at one of the pieces, it highlights, much like a RealityKit entity with a `HoverEffectComponent` does. If the player taps while gazing at the various pieces, they return to their original position, reassembling the original shape.

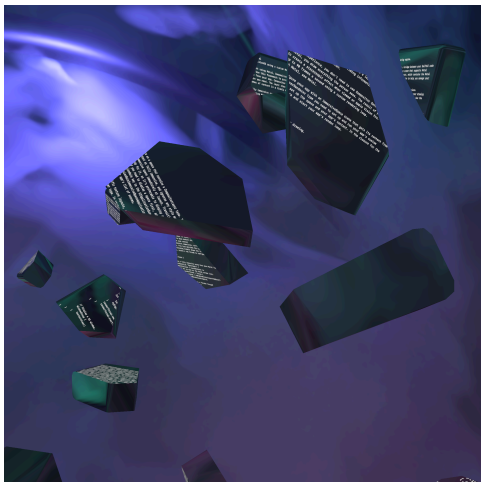


Play

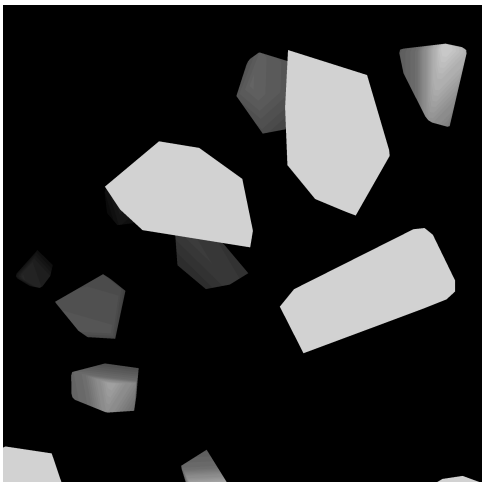
Understand the flow

To protect privacy, Metal shaders can't get information about where a person using an Apple Vision Pro device is currently looking. RealityKit apps can add gaze highlighting using a `HoverEffectComponent`. For Full Space Metal apps, that's not an option. Instead, Compositor Services provides a privacy-preserving mechanism for highlighting the virtual objects that Metal renders. The system does the gaze testing and highlight rendering, not the app.

Compositor Services provides your app with an `Integer` frame buffer for passing in the indices of the draw calls your app needs the system to highlight. The system then does gaze hit-testing on those items, and when a player looks at one of them, it renders a highlight over it. It does this out of process, and your app can't access the actual gaze data, but can respond to spatial gestures that a player generates by tapping those items.



Color



Depth

Tracking Areas

Declare the immersive space

In `CompositorServicesHoverEffectApp`, the app declares a `WindowGroup` for a small SwiftUI window that shows at launch, and an `ImmersiveSpace` that puts the app in Full Space mode when a player activates it. The app declares the space to `ImmersionStyle` with `full`. The body of the `ImmersiveSpace` displays a `CompositorLayer` that the `makeCompositorLayer()` function creates, configures, and returns.

```
ImmersiveSpace(id: AppModel.immersiveSpaceId) {
    makeCompositorLayer()
}
.immersionStyle(selection: .constant(.full), in: .full)
```

Set up the compositor layer

The `makeCompositorLayer()` function first sets up the depth and color buffers by specifying their format.

```
func makeCompositorLayer() -> CompositorLayer {
    CompositorLayer(configuration: { capabilities, configuration in

        // Set the buffer formats for the depth and color buffer.
        configuration.depthFormat = .depth32Float
        configuration.colorFormat = .bgra8Unorm_srgb
```

Then it enables foveation if the device supports it. For more information on using foveation in your Metal apps, see Drawing fully immersive content using Metal.

```
        // If the device supports foveation, enable or disable it according to the u
        if capabilities.supportsFoveation {
            configuration.isFoveationEnabled = appModel.foveation
        }
```

Set up tracking areas

Then `makeCompositorLayer()` checks whether the app is running in visionOS 26 or later. If it is, it enables hover effects by specifying a `trackingAreasFormat` value of `MTLPixelFormat.r8Uint`, which tells the system to enable gaze tracking and highlighting using unsigned integer values to represent different draw calls.

> **Important**
>
> Using an 8-bit integer buffer, your app is limited to 255 tracking areas per frame. If your app needs more, you can also use 16-bit integers, which allows up to 65,534 tracking areas, but also increases your app's memory consumption.

```
        // Set up features requiring visionOS 26 or later.
        if #available(visionOS 26.0, *), appModel.withHover {
            // Enable the tracking area buffer for Metal.
            configuration.trackingAreasFormat = .r8Uint
```

After that, `makeCompositorLayer()` specifies the type of access its shaders need to the tracking-area data. Because the app offers multisample antialiasing (MSAA) as an option, it specifies different access depending on whether MSAA is available on the device and enabled in the app's preferences. Metal handles MSAA for color buffers automatically, but not for integer buffers. When using both hover effects and MSAA, the app configures a usage of ".shaderWrite" instead of ".renderTarget" on the tracking area texture, because the texture will be the output of a custom tile resolver.

Rendered scenes look better with MSAA, so the app offers it as a configurable option. To work around the fact that some pixels don't reference the correct draw call identifier, it implements a compute shader to ensure the draw call for any specific pixel is correct. To do that, the app's shaders need `.shaderWrite` access when MSAA is enabled. If MSAA isn't enabled, the configuration only needs write access to the render target, and uses the provided draw call identifier values.

```
            // Specify how to use the data.
            if appModel.withHover && appModel.useMSAA {
```

```
                configuration.trackingAreasUsage = [.shaderWrite, .shaderRead]
            } else {
                configuration.trackingAreasUsage = [.renderTarget, .shaderRead]
            }
```

Override maximum render quality

If the player requests a specific render quality, the function passes that value into the configuration.

```
            // Override the render-quality resolution, if requested.
            if appModel.overrideResolution {
                configuration.maxRenderQuality = .init(appModel.resolution)
            }
        }
```

> **Note**
>
> If you plan to let people override the maximum render quality, make sure you thoroughly test performance. Increasing the render quality greatly increases the amount of processing power used by your app.

Then, in the trailing closure, it calls the `render(_:)` function.

```
}) { renderer in
        render(renderer)
}
```

Set up rendering

The `render(_:)` function creates a high-priority asynchronous task so that the rendering work doesn't occur on the main thread. Then it creates a `RenderData` object, which is an `Actor` object that holds all of the app's render-related objects. Using an `actor` ensures that all code affecting the rendering data runs in the same global concurrency thread pool.

The function starts by setting up world tracking, loading assets, and implementing the render pipelines for its shaders. For more information about world tracking, see Tracking specific points in world space.

```
Task(priority: .high) {
    let renderData = RenderData(renderer, theAppModel: appModel)
    await renderData.setUpWorldTracking()
    await renderData.loadAssets()
    await renderData.setUpTileResolvePipeline()
    await renderData.setUpShaderPipeline()
```

Listen for spatial events

Next, the sample app checks whether it's running in visionOS 26 or later again. If it is, it registers a closure that the renderer calls whenever that renderer generates a spatial event, such as a gesture the player makes while gazing at content with hover effects on.

The system calls the app's closure when a tap gesture starts, as well as when it ends. The app checks specifically for the `.ended` event so that a piece only returns to its original position after the gesture is over. It gets the tracking-area identifier from the event and stores it in the render data to inform the shaders. It also begins another task that sleeps for a period of time and then unhides the tapped item.

```
if #available(visionOS 26.0, *) {
    renderer.onSpatialEvent = { events in
        for event in events {
            logger.log(level: .info, "Received spatial event:\(String(describing: ev
            let id = event.trackingAreaIdentifier.rawValue
            let phase = event.phase
            if id != 0 && phase == .ended {
                Task(priority: .userInitiated) {
                    await renderData.tap(on: id)
                }
            }
        }
    }
}
```

Start the render loop

The last thing the `render(_:renderData:)` function does is call the `renderLoop()` function on the render data actor. By putting the render loop on the actor, it ensures all attempts to read and write render data happen in the same concurrent context.

```
    await renderData.renderLoop()
```

The `renderLoop()` function enters a `while` loop until the system invalidates the renderer. Each time through the loop, it gets the next frame from the renderer.

```
/// Performs the rendering loop.
func renderLoop() async {
    while renderer.state != .invalidated {
        guard let frame = renderer.queryNextFrame() else { continue }
        frame.startUpdate()
        frame.endUpdate()
```

Then it calcuates the optimal time to wait before submitting data for the next frame, and waits for that length of time.

```
        guard let timing = frame.predictTiming() else { continue }
        LayerRenderer.Clock().wait(until: timing.optimalInputTime)
```

After that, the app retrieves the drawables from the current frame object. When running in visionOS 26 or later, it uses `LayerRenderer/Frame/queryDrawables()` to retrieve all available `LayerRenderer.Drawable` objects. In earlier visionOS versions, it retrieves a single `Drawable` and puts it in an array so the output of the calls are the same type, regardless of the OS version.

```
        frame.startSubmission()

        buffer = queue.makeCommandBuffer()!
        let drawables = {
            if #available(visionOS 26.0, *) {
                return frame.queryDrawables()
            } else {
                return frame.queryDrawable().map { [$0] } ?? []
            }
        }()
```

Next, it iterates through the array of `LayerRenderer.Drawable` objects and passes each `Drawable`, along with its offset index, to `handleDrawable(_:_:)`, which handles processing a single drawable.

```
        if drawables.isEmpty { break }
        for pair in drawables.enumerated() {
            let drawable = pair.element
            let offset = pair.offset
            await handleDrawable(drawable, offset)
        }

        buffer?.commit()
        frame.endSubmission()
    }
}
```

Handle draw calls and MSAA

Before processing a draw call, `handleDrawable(_:_:)` retrieves all the objects it needs, including the draw calls, device anchor, renderer pass descriptor, encoder, and viewports. It also calls the `setUpMSAA(drawable:offset:)` function to cache the textures the system needs when resolving the correct object ID for pixels that system anti-aliasing impacts in a compute shader. The function adds the color, depth, and tracking to the drawable's texture cache because the compute shader needs access to that information.

```
func setUpMSAA(drawable: LayerRenderer.Drawable,
               offset: Int) async {

    if appModel.useMSAA {
        if colorTextureCache.perDrawable[offset] == nil {
            colorTextureCache.perDrawable[offset] = memorylessTexture(from: drawable
        }

        if #available(visionOS 26.0, *), appModel.withHover {
            if indexTextureCache.perDrawable[offset] == nil {
                indexTextureCache.perDrawable[offset] = memorylessTexture(from: draw
            }
        }

        if depthTextureCache.perDrawable[offset] == nil {
            depthTextureCache.perDrawable[offset] = memorylessTexture(from: drawable
        }
    }
```

Add a hover effect to the drawable's tracking area

The sample app then iterates through each of the drawable's draw calls using `handleDrawCall(encoder:drawable:drawCall:id:)`. A critical step for gaze tracking happens here. When running in visionOS 26 or later, the system creates a tracking area for the drawable and then calls `addHoverEffect()` on it, passing the raw value from the tracking area into a *uniform*, which is a constant value that the app passes to its shaders.

```
if #available(visionOS 26.0, *), appModel.withHover, drawCall.hasHover {
    let trackingArea = drawable.addTrackingArea(identifier: .init(UInt64(id)))
    trackingArea.addHoverEffect(.automatic)
    uniforms.hoverIndex = trackingArea.renderValue.rawValue
}
```

Return hover indices from your fragment shader

In addition to returning the fragment color, for hover effects to work, the app's fragment shader also needs to return the object index of the fragment's corresponding draw call. Without the object index, the system can't perform gaze testing or highlighting.

The sample does this by declaring a Metal `struct` in `shaders.metal`. This lets the fragment shader return both the color and the object index.

```
struct FragmentOut {
    float4 color [[color(0)]];
    uint16_t index [[color(1)]];
};
```

Then, in the fragment shader, after doing any other fragment processing the app needs to render its content, the sample returns the calculated color for the fragment, and its object index passes in from the spatial event closure.

```
return FragmentOut {
    float4(outColor, 1.0),
    uniformsArray.hoverIndex
};
```

Implement a custom resolver for MSAA support

At this point, hover effects work, however, gaze tracking isn't taking system-provided anti-aliasing into account. Because Metal's MSAA resolve ignores integer buffers, and MSAA needs to compute a value from all the MSAA samples, the app uses a compute shader to resolve the MSAA samples of the tracking area textures.

```
kernel void block_resolve(imageblock<FragmentOut> block,
                          ushort2 tid [[thread_position_in_threadgroup]],
                          uint2 gid [[thread_position_in_grid]],
                          ushort array_index [[render_target_array_index]],
                          texture2d_array<uint16_t, access::write> resolvedTextureAr
                          texture2d<uint16_t, access::write> resolvedTexture [[textu
{

    const ushort pixelCount = block.get_num_colors(tid);
    ushort index = 0;

    for (ushort i = 0; i < pixelCount; ++i) {
        const FragmentOut color = block.read(tid, i, imageblock_data_rate::color);
        index = max(index, color.index);
    }

    if (use_texture_array) {
        resolvedTextureArray.write(index, gid, array_index);
    } else {
        resolvedTexture.write(index, gid);
    }
}
```

The system calls this compute function for every pixel. The function loops through the fragments and selects the highest object index of any of MSAA samples. The function then writes the value to the texture.

See Also

App integration

- Drawing fully immersive content using Metal

  Create a fully immersive experience in visionOS using a custom Metal-based rendering engine.

{} Interacting with virtual content blended with passthrough

Present a mixed immersion style space to draw content in a person's surroundings, and choose how upper limbs appear with respect to rendered content.

`struct CompositorLayer`

A type that you use with an immersive space to display fully immersive content using Metal.

`protocol CompositorLayerConfiguration`

An interface for specifying the texture configurations and rendering behaviors to use with your Metal rendering engine.

`struct DefaultCompositorLayerConfiguration`

A type that configures the layer with the default texture configurations and rendering behaviors for the current device.