Article

Drawing fully immersive content using Metal

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

Overview

If your app draws fully immersive content using Metal, Compositor Services provides a bridge between your SwiftUI code and your Metal rendering engine code. Use this framework to present a fully immersive scene that supports Metal rendering. When you present the scene from your app, Compositor Services provides a layer, which contains the Metal types, textures, and other information you need. The layer also provides timing information to help you manage your app's rendering loop and deliver frames of content in a timely manner.

> **Note**
>
> If you don't use Metal to create a fully immersive experience, you don't need to use Compositor Services.

When creating fully immersive content using Metal, you draw everything the person sees. The result of your drawing efforts is two images, one for each eye, to create a stereosopic effect when viewed on Apple Vision Pro. Use the timing information in the layer to render up to 90 frames a second using a custom rendering loop.

For information about how to draw content with Metal, see Metal.

Add an immersive space for your content

To present your fully immersive experience, configure your app with an `ImmersiveSpace` scene that gets its content from a `CompositorLayer` type. This type provides the `LayerRenderer` type you need to set up and run your app's custom rendering loop. The following example shows how to set up the space and your app's content. In the closure for the `CompositorLayer` type, create a new thread to configure and start your app's render loop.

```
@main
struct MyApp: App {
    var body: some Scene {

        // Display a fully immersive scene that uses Metal for drawing.
        ImmersiveSpace(id: "MyContent") {
            CompositorLayer { layerRenderer in
                // Set up and run the Metal render loop.
                let renderThread = Thread {
                    let engine = myEngineCreate(layerRenderer)
                    myEngineRenderLoop(engine)
                }
                renderThread.name = "Render Thread"
                renderThread.start()
            }
        }

        // Display a 2D window.
        WindowGroup {
            ContentView()
        }
    }
}
```

Don't include any style modifiers on a space that contains a `CompositorLayer` type. The system automatically configures a space with `CompositorLayer` content as fully immersive, and ignores any style modifiers.

Typically, apps don't display an immersive space immediately at launch. Transitioning to a fully immersive experience can be jarring if someone isn't ready for it, so it's preferable to display a window first and let someone enter the space when they're ready. However, if you need to display a space first, add the `UIApplicationPreferredDefaultSceneSessionRole` key to the `UIApplicationSceneManifest` in your app's `Info.plist` file. Set the value of this key to `CPSceneSessionRoleImmersiveSpaceApplication`. When this key is present, the system displays the first space it finds in your app's list of scenes.

Customize the configuration of your layer

If your Metal rendering engine requires specific texture layouts, pixel formats, or rendering options, specify those details when you configure your `CompositorLayer` type. In your scene creation code, pass a type that adopts `CompositorLayerConfiguration` as a parameter to your scene content. The system uses that information to configure the Metal textures your `LayerRenderer` provides. If you don't provide a custom configuration, Compositor Services uses a set of default configuration values.

To specify a set of custom options, define a type that adopts the `CompositorLayerConfiguration` protocol and implement its `makeConfiguration(capabilities:configuration:)` method. In your implementation of that method, update the default values in the `configuration` parameter with your preferred choices. Change only the values you want and leave the other values alone. Configuration options that are available on a device might not be available in Simulator, so use the `capabilities` parameter to validate your choices before making them. The following example changes the pixel format to one that supports HDR values, and configures the texture layout based on the current foveation setting:

```
struct MyContentConfiguration: CompositorLayerConfiguration {
    func makeConfiguration(
        capabilities: LayerRenderer.Capabilities,
        configuration: inout LayerRenderer.Configuration
    ) {
        let supportsFoveation = layerCapabilites.supportsFoveation
        let supportedLayouts = supportedLayouts(options: supportsFoveation ?
                                                [.foveationEnabled] : [])

        configuration.layout = supportedLayouts.contains(.layered) ? .layered : .dec
        configuration.isFoveationEnabled = supportsFoveation

        // HDR support
        configuration.colorFormat = .rgba16Float

    }
```

> **Note**
>
> Apple Vision Pro uses the P3 color space for all pixel color values.

To use your configuration options for rendering, pass your custom `CompositorLayerConfiguration` type to your `CompositorLayer` at initialization time. The following example

modifies the previous scene's setup code to include custom configuration data. Compositor Services integrates your configuration details into the `LayerRenderer` type it creates.

```
@main
struct MyApp: App {
    var body: some Scene {

        ImmersiveSpace(id: "MyContent") {
            CompositorLayer (configuration: MyContentConfiguration()) { layerRendere
                // Set up and run the Metal render loop.
                let renderThread = Thread {
                    let engine = myEngineCreate(layerRenderer)
                    myEngineRenderLoop(engine)
                }
                renderThread.name = "Render Thread"
                renderThread.start()
            }
        }
        // Other scenes...
    }
}
```

Configure your app's rendering loop

When your app displays a space with a `CompositorLayer`, the system runs the code you provide. Use that code to configure your Metal rendering engine and spawn a thread for your rendering loop, but don't start rendering your content immediately. Instead, check the state of the `LayerRenderer` type to see if the scene is actually running. The system might leave a scene in the `LayerRenderer.State.paused` state while it confirms the person wants to enter the fully immersive experience. The system changes the state to `LayerRenderer.State.running` only when it's ready to display your scene's content.

The following example shows the logic you might use to check the state of your loop each time through your rendering loop. While the layer is paused, the code pauses the render loop thread and waits until the layer starts running again. When the system or a person dismisses the scene, the layer moves to the `LayerRenderer.State.invalidated` state to let you know it's time to stop your rendering loop.

```
void myEngineRenderLoop(my_engine *engine) {
    my_engine_setup_render_pipeline(engine);
```

```
    bool is_rendering = true;
    while (is_rendering) @autoreleasepool {
        switch (cp_layer_renderer_get_state(engine->layer_renderer)) {
            case cp_layer_renderer_state_paused:
                // Wait until the scene appears.
                cp_layer_renderer_wait_until_running(engine->layer_renderer);
                break;

            case cp_layer_renderer_state_running:
                // Render the next frame.
                my_engine_render_new_frame(engine);
                break;

            case cp_layer_renderer_state_invalidated:
                // Exit the render loop.
                is_rendering = false;
                break;
        }
    }

    my_engine_invalidate(engine);
}
```

Important

Always run your render loop code in a dedicated thread, and not in your app's main thread.

Creating Metal pipeline state information is potentially expensive, so use the setup phase of your render loop to configure as much of your Metal code as possible. Start loading textures and shader code, and set up the render and compute descriptors you need for your content. You can also use your setup code to configure the ARKit code you need to fetch device anchor information.

Until your scene is visible, you can't fetch new frames from the `LayerRenderer` and use them to configure your rendering code. If you need information about the configuration of textures, create a `LayerRenderer.Properties` type using the same `CompositorLayerConfiguration` information you used to configure your scene. The `LayerRenderer.Properties` type contains the number of views to draw and the organization of textures for each frame.

Update and encode a single frame of content

While your layer is in the `LayerRenderer.State.running` state, fetch a new frame and fill it with content each time through your render loop. The layer manages a finite number of frames, so

render only one frame at a time and submit it. Compositor Services provides timing information with each frame to help you start work on the frame at the appropriate time and submit your changes before the system needs them.

The following sequence shows the steps to create a single frame of content. Perform these steps each time through your app's render loop.

1. Call `queryNextFrame()` to fetch the next frame to use for drawing.
2. Call `predictTiming()` (or `cp_frame_predict_timing`) to get the predicted render deadlines for your code. You use this information later to pause your thread until the optimal rendering time.
3. Call `startUpdate()` to mark the start of the update phase.
4. Apply user interactions to your content and update any app-specific data.
5. Perform any rendering-related work that doesn't rely on the device anchor information.
6. Call `endUpdate()` to mark the end of the update phase.
7. Call `wait(until:tolerance:)` (or `cp_time_wait_until`) to pause your render loop until the optimal rendering time.
8. Call `startSubmission()` to mark the start of submission phase.
9. Fetch the predicted device anchor from ARKit using the `frameTiming` information, and apply that anchor to your frame.
0. Encode any drawing commands that depend on the device position or orientation.
1. Call `encodePresent(commandBuffer:)` to encode a presentation event into your command buffer.
2. Commit your command buffer.
3. Call `endSubmission()` to mark the end of your GPU submission.

The system uses data from the `startUpdate()`, `endUpdate()`, `startSubmission()`, and `endSubmission()` functions to improve the timing information for subsequent frames. Call these functions to ensure your app has accurate timing information, and to help the system manage CPU and GPU resources efficiently.

The following example shows the structure of the drawing code for rendering one frame of content. The custom `my_engine_gather_inputs`, `my_engine_update_frame`, and `my_engine_draw_and_submit_frame` functions perform custom tasks the app needs to update its data structures and render the content of the frame. The code also fetches the current device anchor from ARKit using the custom `my_engine_get_ar_device_anchor` function and associates that information with the frame.

```
void my_engine_render_new_frame(my_engine *engine) {
    // Get the next frame.
    cp_frame_t frame = cp_layer_renderer_query_next_frame(engine->layer_renderer);
    if (frame == nullptr) { return; }

    // Fetch the predicted timing information.
    cp_frame_timing_t timing = cp_frame_predict_timing(frame);
    if (timing == nullptr) { return; }

    // Update the frame...
    cp_frame_start_update(frame);

    // Update any position- or orientation-independent information.
    my_input_state input_state = my_engine_gather_inputs(engine, timing);
    my_engine_update_frame(engine, timing, input_state);

    cp_frame_end_update(frame);

    // Wait until the optimal time for querying the input.
    cp_time_wait_until(cp_frame_timing_get_optimal_input_time(timing));

    // Submit the frame...
    cp_frame_start_submission(frame);
    cp_drawable_t drawable = cp_frame_query_drawable(frame);
    if (drawable == nullptr) { return; }

    cp_frame_timing_t timing = cp_drawable_get_frame_timing(frame);
    ar_device_anchor_t anchor = my_engine_get_ar_device_anchor(engine, timing);
    cp_drawable_set_ar_device_anchor(drawable, anchor);

    my_engine_draw_and_submit_frame(engine, frame, drawable);

    cp_frame_end_submission(frame);
}
```

For information about how to set up Metal command buffers and command encoders, see Setting up a command structure.

Configure the render pass descriptor for the frame

During drawing, add the textures from your frame's `LayerRenderer.Drawable` to your render pass descriptor. The render pass descriptor tells Metal where to deliver the output of your

rendering commands. Because each frame of content relies on different textures, you must create and configure a render pass descriptor with the current frame's textures each time through your render loop.

The following example shows a function that creates a new render pass descriptor and configures its texture information. The `LayerRenderer.Drawable` in the example uses the `LayerRenderer.Layout.layered` layout, which uses a single texture of type `MTLTextureType.type2DArray`. You could use similar code to set up the render pass descriptor for the `LayerRenderer.Layout.shared` layout.

```
MTLRenderPassDescriptor* my_renderer_create_render_descriptor(my_renderer *renderer,
                                                              cp_drawable_t *drawabl
    MTLRenderPassDescriptor *pass_descriptor = [[MTLRenderPassDescriptor alloc] init

    pass_descriptor.colorAttachments[0].texture = cp_drawable_get_color_texture(draw
    pass_descriptor.colorAttachments[0].storeAction = MTLStoreActionStore;

    pass_descriptor.depthAttachment.texture = cp_drawable_get_depth_texture(drawable
    pass_descriptor.depthAttachment.storeAction = MTLStoreActionStore;

    pass_descriptor.renderTargetArrayLength = cp_drawable_get_view_count(drawable);

    // Foveation support
    pass_descriptor.rasterizationRateMap = cp_drawable_get_rasterization_rate_map(dr

    return pass_descriptor;
}
```

For a `LayerRenderer.Layout.dedicated` layout, you must perform two render passes on your content and create a separate render pass descriptor for each one. Configure each render pass descriptor with the texture at a different index in the arrays of the `LayerRenderer.Drawable` type.

Retrieve device anchor information and attach it to the frame

To prevent the person viewing your content from experiencing disorientation or physical discomfort, it's essential to match the position of the camera in your scene to the location of the person's head. Matching the person's head movements ensures that what they see doesn't conflict with the input their body receives from the real world.

Because you render your app's content in advance, you also need to know the position and orientation of the device in advance. To retrieve this information, use ARKit to call `ar_world_tracking_provider_query_device_anchor_at_timestamp` during the encoding process for your frame. ARKit provides this function to deliver the expected device anchor at the time you specify. Use this information to configure any camera positions during rendering.

The following example shows how to retrieve the predicted device anchor using ARKit. Use the timing information from the `LayerRenderer.Drawable` to get the most accurate presentation time for the frame. Return the device anchor upon success or return `nil` if an error occurs.

```
ar_device_anchor_t my_engine_get_ar_device_anchor(my_engine *engine, cp_frame_timing
    ar_device_anchor_t anchor = ar_device_anchor_create();
    ar_world_tracking_provider_t provider = engine->my_world_tracking_provider;

    // Fetch the device anchor from ARKit.
    auto p_time = cp_time_to_cf_time_interval(cp_frame_timing_get_presentation_time(
    auto anchor_status = ar_world_tracking_provider_query_device_anchor_at_timestamp
    if (anchor_status == ar_device_anchor_query_status_success) {
        return anchor;
    }
    return nil;
}
```

When it displays your frame, the system checks for a discrepancy between the predicted device anchor you provided for your frame and the actual device anchor the hardware reports. If there's a difference, the system automatically adjusts the rendered frame to compensate for the movement. If you don't want the system to make this adjustment, don't specify a device anchor using the `cp_drawable_set_device_anchor` function.

For more information about how to track the device anchor, see ARKit.

Render each view with the correct perspective

The goal of your Metal rendering engine is to produce 2D textures to display to the viewer. When your content is 3D, you need to map points in your scene to the 2D texture in a way that makes the content look realistically 3D to someone viewing it. This process requires you to create a projection matrix that maps points in your 3D content to points on the texture for each view. For stereoscopic rendering, you also have to account for the positional differences between the device anchor and the position of the person's eyes.

During rendering, the rendering engine calls the method in the following example once for each view in the frame. It uses the device anchor it assigned to the frame earlier to create a transform to compensate for any differences between the device position and the view's position. It also

creates a projection matrix using the view's tangents information and the distances to the near and far projection planes. The `makeProjectiveTransformFromTangents` function assembles the actual matrix values in the same way as `init(leftTangent:rightTangent:topTangent:bottomTangent:nearZ:farZ:reverseZ:)`.

```
typedef struct{
    simd_float4x4 projectionMatrix;
    simd_float4x4 viewMatrix;
} Uniforms;

static const NSUInteger MaxBuffersInFlight = 3;
@implementation Renderer (UniformsExtension)
   id <MTLBuffer> _uniformBufferAddress[MaxBuffersInFlight];

- (void)updateUniformsForRenderer:(Renderer*)renderer
                      withDrawable:(cp_drawable_t)drawable
                           atIndex:(size_t)index {

    Uniforms *uniforms = (Uniforms*)_uniformBufferAddress;

    // Get the current device anchor value.
    ar_device_anchor_t device_anchor = cp_drawable_get_device_anchor(drawable);
    simd_float4x4 head_position = ar_anchor_get_origin_from_anchor_transform(device_

    cp_view_t view = cp_drawable_get_view(drawable, index);
    simd_float4 tangents = cp_view_get_tangents(view);
    simd_float2 depth_range = cp_drawable_get_depth_range(drawable);
    simd_float4x4 transform = makeProjectiveTransformFromTangents(tangents[0], /* le
                                                                   tangents[1], /* ri
                                                                   tangents[2], /* to
                                                                   tangents[3], /* bo
                                                                   depth_range[1], /*
                                                                   depth_range[0], /*
                                                                   true); /* reverseZ

    uniforms[index].projectionMatrix = transform;

    // Adjust the camera transform for the current eye position.
    simd_float4x4 camera_transform = simd_mul(head_position, cp_view_get_transform(v
    uniforms[index].viewMatrix = simd_inverse(camera_transform);
}

@end
```

Respond to interactions with your custom content

When your scene is visible, you're responsible for managing all interactions with your custom content. Because you render everything yourself using Metal, you can't rely on view-based events or gesture recognizers for input. Instead, use one of the following techniques:

- Add an `.onSpatialEvent` callback to your layer and map touch events to your content.
- Use ARKit hand tracking to manage input yourself.

When the system detects any direct or indirect touch events, it reports them to the `.onSpatialEvent` callback of the `LayerRenderer`. Use this callback to handle any interactions with your custom content. The system executes your callback on the main thread each time a new touch occurs or an active touch changes, so keep your callback code brief. The following example shows how to add this callback to your layer:

```
@main
struct MyApp: App {
    var body: some Scene {

        // Create a fully immersive scene.
        ImmersiveSpace(id: "MyContent") {
            CompositorLayer (configuration: MyContentConfiguration()) { layerRendere
                // Set up and run the Metal render loop.
                let renderThread = Thread {
                    let engine = myEngineCreate(layerRenderer)
                    myEngineRenderLoop(engine)
                }
                renderThread.name = "Render Thread"
                renderThread.start()

                // Handle any events in the scene.
                layerRenderer.onSpatialEvent = { eventCollection in
                    var events = eventCollection.map { mySpatialEvent($0) }
                    myEnginePushSpatialEvents(engine, &events, events.count)
                }
            }
        }
         // Other scenes...
    }
}
```

> **Important**
>
> To prevent issues when reading or writing event data, use locks or another synchronization mechanism to access event data. The system delivers events on the app's main thread, but your rendering loop handles those events on a different thread. Synchronization is therefore necessary to prevent errors, undefined behavior, or crashes.

For information about ARKit hand tracking, see ARKit.

See Also

App integration

{} 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.

{} Rendering hover effects in Metal immersive apps

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

`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.