Sample Code

Creating a fog effect using scene depth

Apply virtual fog to the physical environment.

Download

iOS 15.0+ | iPadOS 15.0+ | Xcode 16.0+

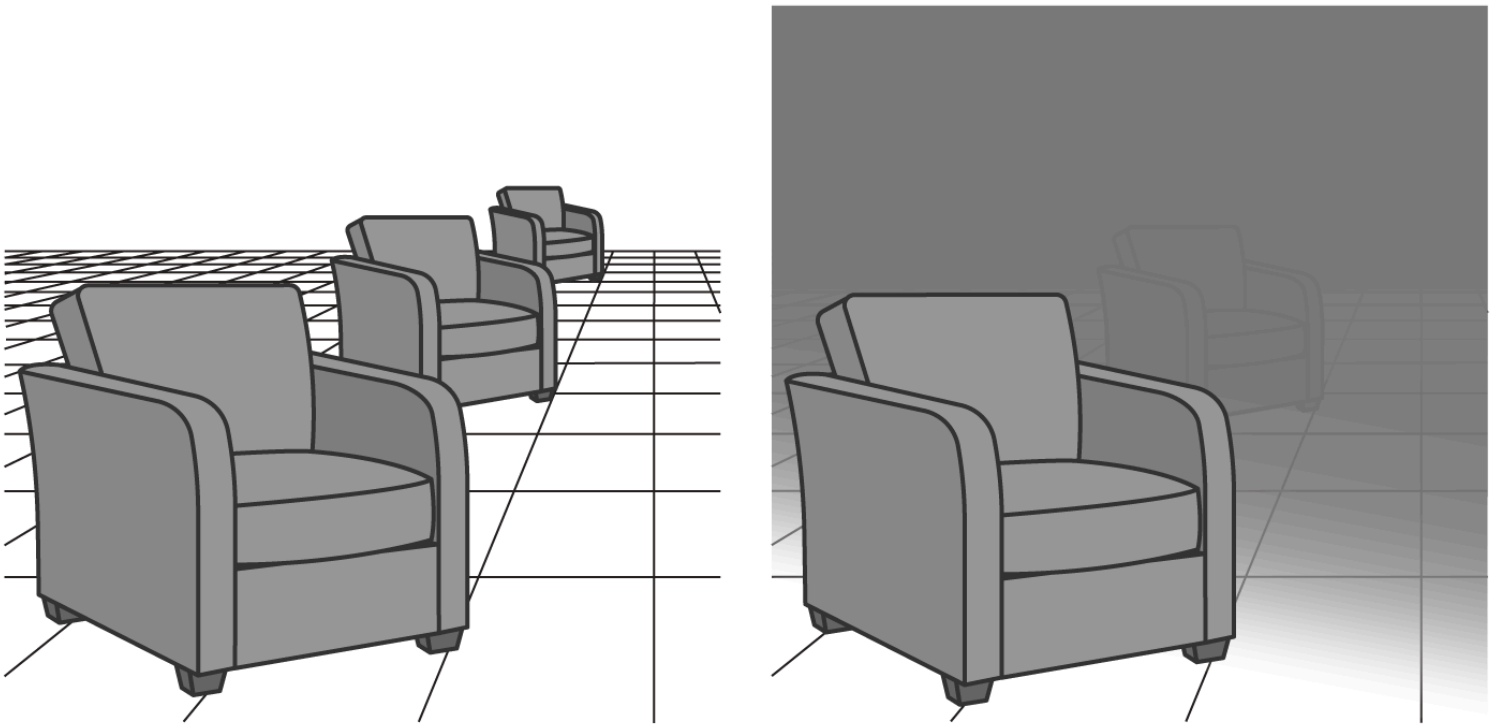
Overview

Devices such as the second-generation iPad Pro 11-inch and fourth-generation iPad Pro 12.9-inch can use the LiDAR Scanner to calculate the distance of real-world objects from the user. In world-tracking experiences on iOS 14, ARKit provides a buffer that describes the objects' distance from the device in meters.

This sample app uses the depth buffer to create a virtual fog effect in real time. To draw its graphics, the sample app uses a small Metal renderer. ARKit provides precise depth values for objects in the camera feed, so the sample app applies a Gaussian blur using Metal Performance Shaders to soften the fog effect. While drawing the camera image to the screen, the renderer checks the depth texture at every pixel, and overlays a fog color based on that pixel's distance from the device. For more information on sampling textures and drawing with Metal, see Creating and sampling textures.

Without fog effect

With fog



Enable scene depth and run a session

In order to avoid running an unsupported configuration, the sample app first checks whether the device supports scene depth.

```
if !ARWorldTrackingConfiguration.supportsFrameSemantics(.sceneDepth) ||
    !ARWorldTrackingConfiguration.supportsFrameSemantics(.smoothedSceneDepth) {
    // Ensure that the device supports scene depth and present
    //  an error-message view controller, if not.
    let storyboard = UIStoryboard(name: "Main", bundle: nil)
    window?.rootViewController = storyboard.instantiateViewController(withIdentifier
}
```

If the device running the app doesn't support scene depth, the sample project will stop. Optionally, the app could present the user with an error message and continue the experience without scene depth.

If the device supports scene depth, the sample app creates a world-tracking configuration and enables the `smoothedSceneDepth` option on the `ARConfiguration.FrameSemanticsuct`` property.

```
configuration.frameSemantics = .smoothedSceneDepth
```

Then, the sample project begins the AR experience by running the session.

```
session.run(configuration)
```

Access the scene's depth

ARKit exposes the depth buffer documentation/arkit/ardepthdata/depthmap) as a CVPixelBuffer on the current frame's `sceneDepth` or `smoothedSceneDepth` property, depending on the enabled frame semantics. This sample app visualizes `smoothedSceneDepth` by default. The raw depth values in `sceneDepth` can create a flicker whereas smoothing the depth differences across frames visualizes a more realistic fog effect. For debug purposes, the sample allows switching between `smoothedSceneDepth` and `sceneDepth` with an onscreen toggle.

```
guard let sceneDepth = frame.smoothedSceneDepth ?? frame.sceneDepth else {
    print("Failed to acquire scene depth.")
    return
}
var pixelBuffer: CVPixelBuffer!
pixelBuffer = sceneDepth.depthMap
```

Every pixel in the depth buffer maps to a region of the visible scene, which defines that region's distance from the device in meters. Because the sample project draws to the screen using Metal, it converts the pixel buffer to a Metal texture as required to transfer the depth data to the GPU for rendering.

```
var texturePixelFormat: MTLPixelFormat!
setMTLPixelFormat(&texturePixelFormat, basedOn: pixelBuffer)
depthTexture = createTexture(fromPixelBuffer: pixelBuffer, pixelFormat: texturePixel
```

To set the depth texture's Metal pixel format, the sample project calls `CVPixelBufferGetPixelFormatType(_:)` with the `depthMap` and chooses an appropriate mapping based on the result.

```
fileprivate func setMTLPixelFormat(_ texturePixelFormat: inout MTLPixelFormat?, base
    if CVPixelBufferGetPixelFormatType(pixelBuffer) == kCVPixelFormatType_DepthFloat
        texturePixelFormat = .r32Float
    } else if CVPixelBufferGetPixelFormatType(pixelBuffer) == kCVPixelFormatType_One
        texturePixelFormat = .r8Uint
    } else {
        fatalError("Unsupported ARDepthData pixel-buffer format.")
    }
}
```

Apply a blur to the depth buffer

As a benefit of rendering its graphics with Metal, this app has at its disposal the display conveniences of MPS. The sample project uses the MPS Gaussian Blur filter to make realistic fog. When instantiating the filter, the sample project passes a `sigma` of 5 to specify a 5-pixel radius blur.

```
blurFilter = MPSImageGaussianBlur(device: device, sigma: 5)
```

> **Note**
>
> To gain performance at the cost of precision, the app can add `MPSKernelOptionsAllowReducedPrecision` to the blur filter's `options`, which reduces computation time by using `half` instead of `float`.

MPS requires input and output images that define the source and destination pixel data for the filter operation.

```
let inputImage = MPSImage(texture: depthTexture, featureChannels: 1)
let outputImage = MPSImage(texture: filteredDepthTexture, featureChannels: 1)
```

The sample app passes the input and output images to the blur's `encode` function, which schedules the blur to happen on the GPU.

```
blur.encode(commandBuffer: commandBuffer, sourceImage: inputImage, destinationImage:
```

> **Note**
>
> In-place MPS operations can save time, memory, and power. Since in-place MPS requires fallback code for devices that don't support it, this sample project doesn't use it. For more information on in-place operations, see Image Filters.

Visualize the blurred depth to create fog

Metal renders by providing to the GPU a fragment shader that draws the app's graphics. Since the sample project renders a camera image, it packages up the camera image for the fragment shader by calling `setFragmentTexture`.

```
renderEncoder.setFragmentTexture(CVMetalTextureGetTexture(cameraImageY), index: 0)
renderEncoder.setFragmentTexture(CVMetalTextureGetTexture(cameraImageCbCr), index: 1
```

Next, the sample app packages up the filtered depth texture.

```
renderEncoder.setFragmentTexture(filteredDepthTexture, index: 2)
```

The sample project's GPU-side code fields the texture arguments in the order of the `index` argument. For example, the fragment shader fields the texture with index `0` above as the argument containing the suffix `texture(0)`, as shown in the example below.

```
fragment half4 fogFragmentShader(FogColorInOut in [[ stage_in ]],
texture2d<float, access::sample> cameraImageTextureY [[ texture(0) ]],
texture2d<float, access::sample> cameraImageTextureCbCr [[ texture(1) ]],
depth2d<float, access::sample> arDepthTexture [[ texture(2) ]],
```

To output a rendering, Metal calls the fragment shader once for every pixel it draws to the destination. The sample project's fragment shader begins by reading the RGB value of the current pixel in the camera image. The object "`s`" is a `sampler`, which enables the shader to inspect a texture at a specific location. The value `in.texCoordCamera` refers to this pixel's relative location within the camera image.

```
constexpr sampler s(address::clamp_to_edge, filter::linear);

// Sample this pixel's camera image color.
float4 rgb = ycbcrToRGBTransform(
    cameraImageTextureY.sample(s, in.texCoordCamera),
    cameraImageTextureCbCr.sample(s, in.texCoordCamera)
);
half4 cameraColor = half4(rgb);
```

By sampling the depth texture at `in.texCoordCamera`, the shader queries for depth at the same relative location that it did for the camera image, and obtains the current pixel's distance in meters from the device.

```
float depth = arDepthTexture.sample(s, in.texCoordCamera);
```

To determine the amount of fog that covers this pixel, the sample app calculates a fraction using the current pixel's distance divided by the distance at which the fog effect fully saturates the

scene.

```
float fogPercentage = depth / fogMax;
```

The `mix` function mixes two colors based on a percentage. The sample project passes in the RGB values, fog color, and fog percentage to create the right amount of fog for the current pixel.

```
half4 foggedColor = mix(cameraColor, fogColor, fogPercentage);
```

After Metal calls the fragment shader for every pixel, the view presents the final, fogged image of the physical environment to the screen.

Visualize confidence data

ARKit provides the `confidenceMap` property within `ARDepthData` to measure the accuracy of the corresponding depth data `depthMap`. Although this sample project doesn't factor depth confidence into its fog effect, confidence data could filter out lower-accuracy depth values if the app's algorithm required it.

To provide a sense for depth confidence, this sample app visualizes confidence data at runtime using the `confidenceDebugVisualizationEnabled` in the `Shaders.metal` file.

```
// Set to `true` to visualize confidence.
bool confidenceDebugVisualizationEnabled = false;
```

When the renderer accesses the current frame's scene depth, the sample project creates a Metal texture of the `confidenceMap` to draw it on the GPU.

```
pixelBuffer = sceneDepth.confidenceMap
setMTLPixelFormat(&texturePixelFormat, basedOn: pixelBuffer)
confidenceTexture = createTexture(fromPixelBuffer: pixelBuffer, pixelFormat: texture
```

While the renderer schedules its drawing, the sample project packages up the confidence texture for the GPU by calling `setFragmentTexture`.

```
renderEncoder.setFragmentTexture(CVMetalTextureGetTexture(confidenceTexture), index:
```

The GPU-side code fields confidence data as the fragment shader's third texture argument.

```
texture2d<uint> arDepthConfidence [[ texture(3) ]])
```

To access the confidence value of the current pixel's depth, the fragment shader samples the confidence texture at `in.texCoordCamera`. Each confidence value in this texture is a `uint` equivalent of its corresponding case in the `ARConfidenceLevel` enum.

```
uint confidence = arDepthConfidence.sample(s, in.texCoordCamera).x;
```

Based on the confidence value at the current pixel, the fragment shader creates a normalized percentage of the confidence color to overlay.

```
float confidencePercentage = (float)confidence / (float)maxConfidence;
```

The sample project calls the `mix` function to blend the confidence color into the processed pixel based on the confidence percentage.

```
return mix(confidenceColor, foggedColor, confidencePercentage);
```

After Metal calls the fragment shader for every pixel, the view presents the camera image augmented with the confidence visualization.

This sample uses the color red to identify parts of the scene in which depth confidence is less than `ARConfidenceLevel.high`. At low confidence depth values with a normalized percentage of `0`, the visualization renders solid red (`confidenceColor`). For high confidence depth values with a value of one, the `mix` call returns the unfiltered, fogged camera-image color (`foggedColor`). At medium-confidence areas of the scene, the `mix` call returns a blend of both colors that applies a reddish tint to the fogged camera-image.

See Also

Video Frame Analysis

{} Displaying a point cloud using scene depth

Present a visualization of the physical environment by placing points based a scene's depth data.

`class ARFrame`

A video image captured as part of a session with position-tracking information.

`class` `ARPointCloud`

A collection of points in the world coordinate space of the AR session.

`class` `ARDepthData`

An object that describes the distance to regions of the real world from the plane of the camera.