Sample Code

Generating procedural textures

Display a 3D model that generates procedural textures in a reality view.

Download

visionOS 2.0+ | Xcode 16.0+

Overview

This sample demonstrates how to create and display a 3D torus model that dynamically regenerates a Metal texture at each render frame within a visionOS app.

At launch, the app starts a window group that contains:

- A 3D torus model with a generated texture
- A button displaying the text “generate”

Play

Generate the texture of the model

The sample constructs the `ProceduralTextureGenerator` structure, which contains the `generate()` method to generate and return a `TextureResource` to apply to the torus model.

The app uses `GKARC4RandomSource` to generate a number using the ARC4 algorithm, and assigns the `seed` to the generator to determine the random source's behavior:

```
import SwiftUI
import RealityKit
import GameplayKit

struct ProceduralTextureGenerator {
    static func generate(_ seed: Int) throws -> TextureResource {
        let length = 1024
        let bytesPerPixel = 1
        let bitSize = 8

        /// The generator to randomly generate numbers.
        let generator = GKARC4RandomSource()

        // Assign the generator seed that determines the random source's behavior.
        guard let setSeed = "\(seed)".data(using: .ascii) else {
            throw ProceduralTextureGeneratorError.convertSeedFailed
        }
        generator.seed = setSeed

        // ...
    }
}
```

After creating the generator, the app creates a `CGDataProvider` with pixels that the generator randomly assigns. Then it creates the `CGImage` for the procedural texture:

```
import SwiftUI
import RealityKit
import GameplayKit

struct ProceduralTextureGenerator {
    static func generate(_ seed: Int) -> TextureResource? {
        // ...

        /// The data provider with pixels that the generator randomly assigns.
        guard let provider = CGDataProvider(
            data: Data(bytes: &pixels, count: pixels.count * bytesPerPixel) as CFDat
        ) else {
            throw ProceduralTextureGeneratorError.providerCreationFailed
```

```
        }

        // The source image for the texture.
        guard let image = CGImage(
            width: length,
            height: length,
            bitsPerComponent: bytesPerPixel * bitSize,
            bitsPerPixel: bytesPerPixel * bitSize,
            bytesPerRow: length * bytesPerPixel,
            space: CGColorSpaceCreateDeviceGray(),
            bitmapInfo: CGBitmapInfo(),
            provider: provider,
            decode: nil,
            shouldInterpolate: true,
            intent: .defaultIntent
        ) else {
            throw ProceduralTextureGeneratorError.imageCreationFailed
        }

        return try TextureResource(image: image, options: TextureResource.CreateOpti
    }
}
```

The app attempts to create and return a `TextureResource` with the generated `image`.

Update the texture in real time

The sample defines the custom system `DrawableQueueSystem` to update the model's texture in real time:

```
import RealityKit

struct DrawableQueueSystem: System {
    static let query = EntityQuery(where: .has(DrawableQueueComponent.self))

    init(scene: RealityKit.Scene) { }

    /// Update the results to attach delta time to the component.
    func update(context: SceneUpdateContext) {
        for entity in context.entities(matching: Self.query, updatingSystemWhen: .re
            if var comp = entity.components[DrawableQueueComponent.self] {
                comp.update(deltaTime: context.deltaTime)
```

```
                entity.components[DrawableQueueComponent.self] = comp
            }
        }
    }
}
```

The `update()` method iterates through all matching entities during the rendering phase and updates each entity's `DrawableQueueComponent` by passing the time difference between frames, allowing for smooth real-time updates in the scene.

The initializer registers the `DrawableQueueSystem` within the component, establishing a Metal texture descriptor and a Metal texture. The app attempts to copy the input `TextureResource` texture data into the Metal texture:

```
import SwiftUI
import RealityKit

struct DrawableQueueComponent: Component {
    // ...

    init(texture: TextureResource) throws {
        DrawableQueueSystem.registerSystem()

        // ...

        /// The descriptor to configure new Metal texture objects.
        let desc = MTLTextureDescriptor()
        desc.width = texture.width
        desc.height = texture.height
        desc.usage = [.shaderWrite, .shaderRead]
        desc.pixelFormat = .r16Float

        guard let newTexture = mtlDevice.makeTexture(descriptor: desc) else {
            throw DrawableComponentError.textureCreationFailed
        }
        self.mtlTexture = newTexture

        try texture.copy(to: mtlTexture)

        // ...
    }
}
```

A `TextureResource.DrawableQueue` makes it possible to update a texture resource dynamically. The app creates a descriptor and inserts it into the drawable queue to dynamically replace the texture:

```
import SwiftUI
import RealityKit

struct DrawableQueueComponent: Component {
    // ...

    init(texture: TextureResource) throws {
        // ...

        /// The descriptor for the drawable queue.
        let queueDesc = TextureResource.DrawableQueue.Descriptor(
            pixelFormat: .rgba16Float,
            width: texture.width,
            height: texture.height,
            usage: [.shaderRead, .shaderWrite],
            mipmapsMode: .none
        )

        // Replace the texture with the result of the drawable queue.
        guard let queue = try? TextureResource.DrawableQueue(queueDesc) else {
            throw DrawableComponentError.queueCreationFailed
        }
        texture.replace(withDrawables: queue)

        /// The name of the shader function.
        let shaderFunctionName: String = "textureShader"

        // Assign the configuration with the shader function.
        guard let library = mtlDevice.makeDefaultLibrary(), let function = library.m
            throw DrawableComponentError.shaderFunctionCreationFailed
        }
        self.pipeState = try mtlDevice.makeComputePipelineState(function: function)

        // Assign the Metal command queue.
        guard let newQueue = mtlDevice.makeCommandQueue() else {
            throw DrawableComponentError.commandQueueCreationFailed
        }
        self.mtlCommandQueue = newQueue
```

```
    }
}
```

The app attempts to load the shader function. Then it creates a compute pipeline state, which allows the GPU to execute the shader function. Additionally, the app creates a command queue to enable communication between the app and the GPU.

The `update(deltaTime:)` method starts by incrementing the value of time, to track and allow contents to change over time:

```
import SwiftUI
import RealityKit

struct DrawableQueueComponent: Component {
    // ...

    mutating func update(deltaTime: TimeInterval) {
        // Increment the value of time with amount of time.
        time += deltaTime

        // ...

        let threadGroupCount = MTLSizeMake(8, 8, 1)
        let threadGroups = MTLSizeMake(
            texture.width / threadGroupCount.width,
            texture.height / threadGroupCount.height,
            1
        )

        encoder.dispatchThreadgroups(threadGroups, threadsPerThreadgroup: threadGrou
        encoder.endEncoding()

        commandBuffer.commit()
        drawable.present()
    }
}
```

The method sets up the encoder, then commits to the command buffer to submit the command buffer to run on the GPU. Lastly, it presents the updated texture to the renderer with the `present()` method.

Set up the torus model

The `EntityView` attempts to create the torus model by calling the `makeEntity()` method:

```
struct EntityView: View {
    /// The value that controls which iteration of the texture the generator uses
    var seed: Int

    // ...

    var body: some View {
        RealityView { content in
            do {
                let entity = try makeEntity()
                content.add(entity)
            } catch {
                print("Error creating entity: \(error)")
            }
        } update: { content in
            for entity in content.entities {
                if let modelEntitty = entity as? ModelEntity {
                    try? updateEntity(modelEntitty)
                } else {
                    print("Entity is not a ModelEntity.")
                }
            }
        }
    }
}
```

The `update` closure loops through all the entities in the reality view, and attempts to update the entity in scene updates.

The `makeEntity()` method marks the `MainActor` to execute in the main dispatch queue:

```
@MainActor
func makeEntity() throws -> ModelEntity {
    let fileName: String = "Torus"

    /// The 3D torus model.
    let entity = try Entity.loadModel(named: fileName)

    // Set the scale of the entity with the bounding radius of the model's visible b
    entity.scale /= entity.visualBounds(relativeTo: nil).boundingRadius
```

```
    let scale: Float = 0.2

    // Apply the scale factor with the value of the scale variable.
    entity.scale *= scale

    // Rotate the entity along the y-axis so that the entity faces the user.
    entity.transform.rotation *= simd_quatf(from: SIMD3<Float>(0, 1, 0), to: SIMD3<F

    try updateEntity(entity)

    return entity
}
```

The `loadModel(named:in:)` method synchronously loads a 3D torus model, scales it to the proper size, and applies the `updateEntity()` method.

The `updateEntity()` method generates a new texture by passing the seed value into the `generate()` method, and creates a material sampler for the `PhysicallyBasedMaterial`:

```
@MainActor
func updateEntity(_ entity: ModelEntity) throws {
    /// The texture based on the seed values.
    guard let texture = ProceduralTextureGenerator.generate(seed) else {
        return
    }

    /// The texture of the sampler, to the nearest pixel.
    var sampler = PhysicallyBasedMaterial.Texture.Sampler()
    sampler.modify { $0.magFilter = .nearest }

    /// The material that simulates the appearance of real-world objects.
    var material = PhysicallyBasedMaterial()

    /// The texture for the material.
    let textureAndSampler = PhysicallyBasedMaterial.Texture(
        texture,
        sampler: sampler
    )
    material.baseColor = PhysicallyBasedMaterial.BaseColor(texture: textureAndSample

    // Apply the material to the model of the entity.
    entity.model?.materials = [material]
```

```
    entity.components[DrawableQueueComponent.self] = try DrawableQueueComponent(text
}
```

The method applies the material to the model of the entity, and attempts to set the `DrawableQueueComponent` to initalize the texture.

Set up the parent window

The `DrawableView` creates a `State` `seed` property to control the randomization of the texture generator:

```
import SwiftUI

struct DrawableView: View {
    @State var seed: Int = 0

    var body: some View {
        HStack {
            EntityView(seed: seed)

            Divider()

            Button("Regenerate") { seed += 1 }
        }.padding()
    }
}
```

Using the `HStack` makes it possible to call the `EntityView` that accepts `seed` and a button that increases the value of `seed` on each press.

See Also

Building materials

{} Implementing adjustable material

Update the adjustable parameters of a 3D model in visionOS.

{} Displaying a stereoscopic image

Build a stereoscopic image by applying textures to the left and right eye in a shader graph material.