Sample Code

Generating interactive geometry with RealityKit

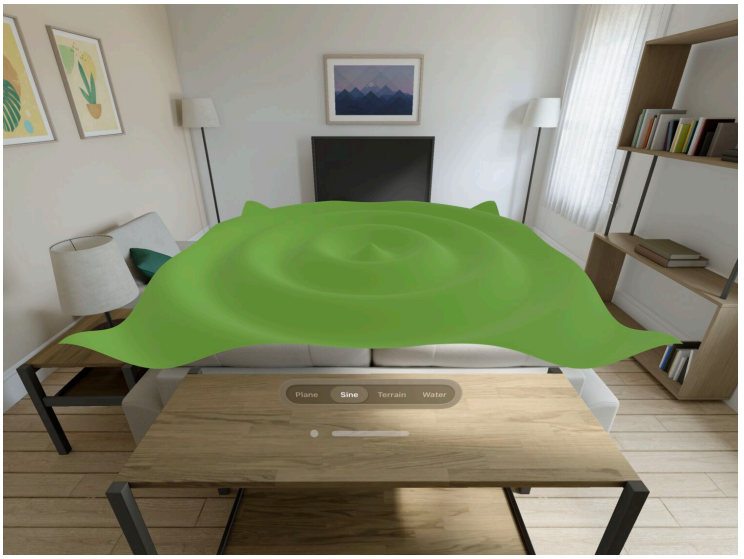
Create an interactive mesh with low-level mesh and low-level texture.

Download

macOS 15.0+ | visionOS 2.0+ | Xcode 16.0+

Overview

This sample app demonstrates how to create geometry that moves and changes shape in real time—to simulate the appearance of fabric, terrain, or water, for example, like in the following images:



To create dynamic, interactive geometry, the app leverages low-level meshes and textures. Using the `LowLevelMesh` and `LowLevelTexture` types, you can customize their data layouts and update their contents directly in real time with Metal compute shaders. You define the compute shaders and the types they operate on using Metal Shading Language (MSL), which is a derivative of C++.

The app starts by generating a plane mesh with `LowLevelMesh`, and it reconfigures the vertices within that mesh by reading data from a low-level texture that stores its height and normal map. As the app runs, it updates both the mesh and texture on the GPU in every frame. The sample demonstrates three unique height map generation functions, including a sine wave that oscillates outward over time, as well as a terrain editor and a water simulation that each respond to a person's input.

Define a custom vertex structure

The sample defines the custom vertex data structure for its plane mesh in an MSL header file:

```
struct PlaneVertex {
    simd_float3 position;
    simd_float3 normal;
};
```

In this example, the vertex has a 3D position and a 3D normal vector, but you can define your app's vertex type with different names and other data.

Create a low-level mesh with a descriptor

The sample creates the plane low-level mesh by defining its vertex attributes, vertex layouts, and the number of vertices and indices it has capacity for:

```
/// The number of vertices in each dimension of the plane mesh.
let dimensions: SIMD2<UInt32>

...

/// Creates a low-level mesh with `PlaneVertex` vertices.
private func createMesh() throws -> LowLevelMesh {
    // Define the vertex attributes of `PlaneVertex`.
    let positionAttributeOffset = MemoryLayout.offset(of: \PlaneVertex.position) ??
    let normalAttributeOffset = MemoryLayout.offset(of: \PlaneVertex.normal) ?? 16

    let positionAttribute = LowLevelMesh.Attribute(semantic: .position, format: .flo
    let normalAttribute = LowLevelMesh.Attribute(semantic: .normal, format: .float3,

    let vertexAttributes = [positionAttribute, normalAttribute]

    // Define the vertex layouts of `PlaneVertex`.
    let vertexLayouts = [LowLevelMesh.Layout(bufferIndex: 0, bufferStride: MemoryLay

    // Derive the vertex and index counts from the dimensions.
    let vertexCount = Int(dimensions.x * dimensions.y)
    let indicesPerTriangle = 3
    let trianglesPerCell = 2
    let cellCount = Int((dimensions.x - 1) * (dimensions.y - 1))
    let indexCount = indicesPerTriangle * trianglesPerCell * cellCount

    // Create a low-level mesh with the necessary `PlaneVertex` capacity.
    let meshDescriptor = LowLevelMesh.Descriptor(vertexCapacity: vertexCount,
                                                 vertexAttributes: vertexAttributes,
                                                 vertexLayouts: vertexLayouts,
                                                 indexCapacity: indexCount)
    return try LowLevelMesh(descriptor: meshDescriptor)
}
```

In this case, determining the exact vertex and index capacities of the mesh is straightforward given its dimensions. For more complex meshes, or meshes you add vertices to or remove vertices from at runtime, you can estimate the maximum vertex capacity and resize the mesh if its vertex count ever exceeds that number.

See Creating a plane with low-level mesh for more information on how the sample creates this mesh, including how it fills the mesh's vertex and index buffers with data, like in the following image:

Create a compute shader dispatch system

The sample defines a custom `ComputeUpdateContext` structure that contains the necessary context for dispatching compute shader functions in every frame:

```
struct ComputeUpdateContext {
    /// The number of seconds elapsed since the last frame.
    let deltaTime: TimeInterval
    /// The command buffer for the current frame.
    let commandBuffer: MTLCommandBuffer
    /// The compute command encoder for the current frame.
    let computeEncoder: MTLComputeCommandEncoder
}
```

The sample dispatches compute shader functions in every frame by passing this structure to each `ComputeSystem` in the app (see Passing Metal command objects around your application).

`HeightMapMesh` is an example of a `ComputeSystem`. It implements the `ComputeSystem` protocol's `update` method to dispatch compute shaders that generate a height map and modify the vertex positions and normals of a mesh in each frame:

```
class HeightMapMesh: ComputeSystem {
    ...

    /// Updates the height map mesh by generating a height map, deriving normals fro
    func update(computeContext: ComputeUpdateContext) {
        ...

        // Generate the height map height values.
        heightMap.generateHeight(computeContext: computeContext, heightMapComputePar
        // Update the height map normal directions.
        heightMap.updateNormals(computeContext: computeContext, heightMapComputePara

        // Update the vertex positions and normals.
        updateVertices(computeContext: computeContext)
    }
}
```

Create a low-level texture height map

The sample creates a height map with `LowLevelTexture` by specifying its pixel format, dimensions, and usage:

```
struct HeightMap {
    ...

    /// The low-level texture that stores the height and normal information of the h
    var heightMapTexture: LowLevelTexture

    init(dimensions: SIMD2<UInt32>) throws {
        // Initialize the texture with an RGBA pixel format where the alpha channel
        // while the red, green, and blue channels store the surface normal directio
        let textureDescriptor = LowLevelTexture.Descriptor(pixelFormat: .rgba32Float
                                                           width: Int(dimensions.x),
                                                           height: Int(dimensions.y)
```

```
                                                        textureUsage: [.shaderRea
        self.heightMapTexture = try LowLevelTexture(descriptor: textureDescriptor)
    }

    ...
}
```

The low-level texture has the same dimensions as the plane low-level mesh, such that each pixel in the texture corresponds to a vertex in the mesh.

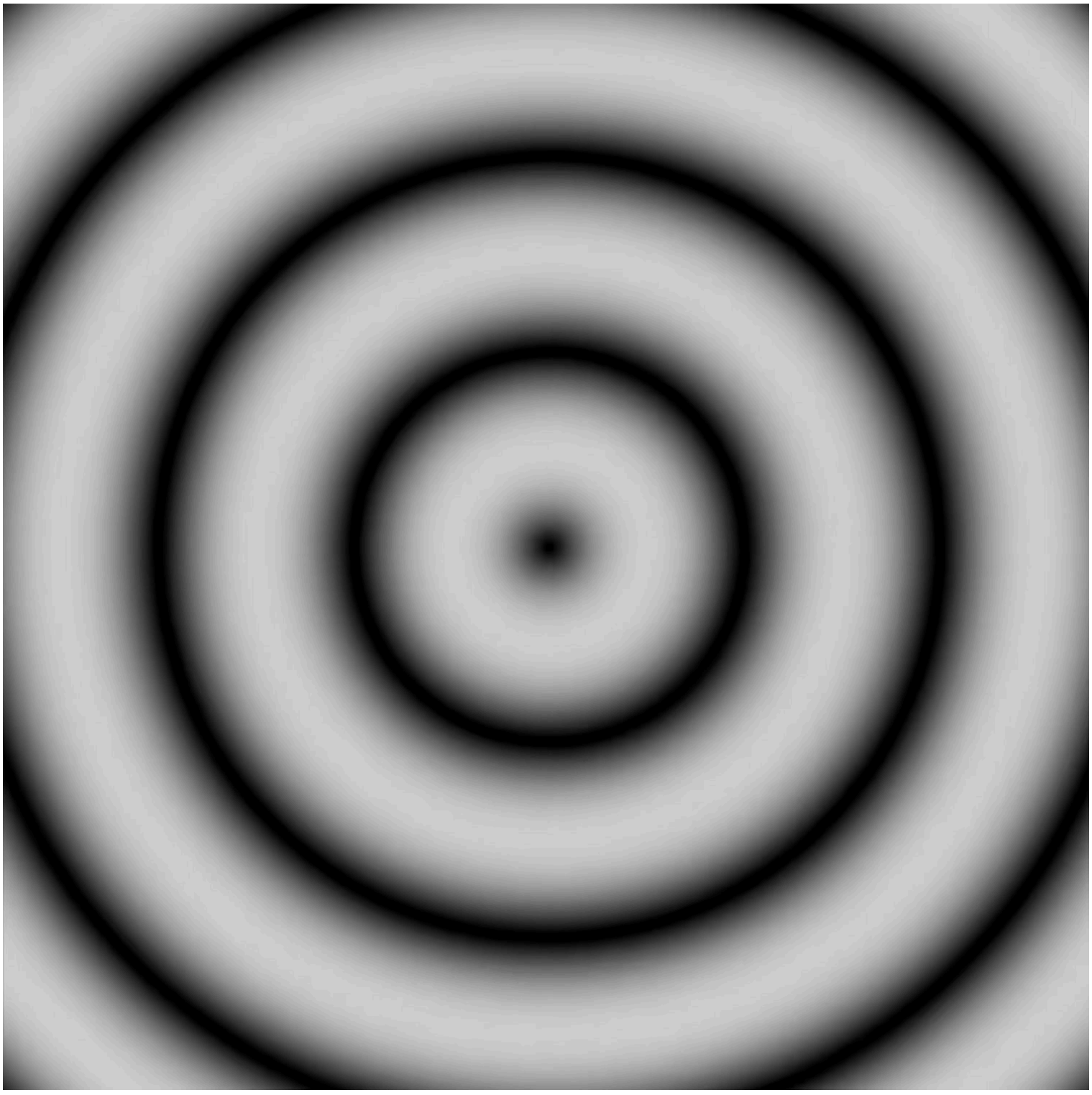
Write height data to the low-level texture

The sample writes height data to the low-level texture in every frame on the GPU by dispatching Metal compute shaders. For example, the compute shader function `generateSineWaveHeightMap` writes height values to the texture in the shape of a sine wave moving outward from the center of the texture over time:

```
[[kernel]]
void generateSineWaveHeightMap(texture2d<float, access::read> heightMapIn [[texture(
                               texture2d<float, access::write> heightMapOut [[textur
                               constant float &time [[buffer(2)]],
                               constant float &amplitude [[buffer(3)]],
                               uint2 pixelCoords [[thread_position_in_grid]]) {
    // Skip out-of-bounds threads.
    // https://developer.apple.com/documentation/metal/compute_passes/calculating_th
    if (pixelCoords.x >= heightMapIn.get_width() || pixelCoords.y >= heightMapIn.get

    // Compute texture coordinates ranging from 0 to 1 along each axis.
    float2 uv = float2(pixelCoords.x / (heightMapIn.get_width() - 1.0),
                       pixelCoords.y / (heightMapIn.get_height() - 1.0));

    // Get the distance to the center of the texture in texture coordinate space.
    float distanceToCenter = length(uv - 0.5);
    // Normalize the distance to a range from 0 to 2π along the horizontal and verti
    float normalizedDistanceToCenter = (distanceToCenter / 0.5) * (2 * M_PI_F);

    // Get sine as a function of the normalized distance to the center of the textur
    // subtracting time to animate it outward over time.
    float waveCount = 3;
    float sine = sin(normalizedDistanceToCenter * waveCount - time);
    // Convert sine to the range 0 to 1.
    float sine01 = (sine + 1) / 2;
```

```
    // Generate height from the sine function.
    float height = amplitude * sine01;

    // Read the current height map data.
    float4 heightMapData = heightMapIn.read(pixelCoords);
    // Update the alpha channel with the new height.
    heightMapData.a = height;
    // Write the updated height data to height map.
    heightMapOut.write(heightMapData, pixelCoords);
}
```

The following video shows the texture the `generateSineWaveHeightMap` compute function creates:



Play ⊙

The sample dispatches this compute shader function in `SineWaveHeightMapGenerator`, passing in both the height map low-level texture, the time, and the amplitude:

```
class SineWaveHeightMapGenerator: HeightMapGenerator {
    /// Compute pipeline corresponding to the Metal compute shader function `generat
    ///
    /// See `SineWaveComputeShader.metal`.
    private let sineWaveHeightPipeline: MTLComputePipelineState = makeComputePipelir

    /// The number of seconds elapsed since the person reset this generator.
```

```
    private var time: Float = 0

    /// The amplitude of the sine wave this generator generates.
    private var amplitude: Float = 0.05

    ...

    /// Dispatches a Metal compute shader to generate a height map in the shape of a
    func generateHeightMap(computeContext: ComputeUpdateContext,
                           heightMapTexture: LowLevelTexture,
                           heightMapComputeParams: HeightMapComputeParams) {
        // Get deltaTime.
        let deltaTime = Float(computeContext.deltaTime)
        // Get the command buffer and compute encoder.
        let commandBuffer = computeContext.commandBuffer
        let computeEncoder = computeContext.computeEncoder
        // Get the threadgroups.
        let threadgroups = heightMapComputeParams.threadgroups
        let threadsPerThreadgroup = heightMapComputeParams.threadsPerThreadgroup

        // Increment time.
        time += deltaTime

        // Set the compute shader pipeline to `generateSineWaveHeightMap`.
        computeEncoder.setComputePipelineState(sineWaveHeightPipeline)

        // Pass a readable version of the height map texture to the compute shader.
        computeEncoder.setTexture(heightMapTexture.read(), index: 0)
        // Pass a writable version of the height map texture to the compute shader.
        computeEncoder.setTexture(heightMapTexture.replace(using: commandBuffer), in

        // Pass the time to the compute shader.
        computeEncoder.setBytes(&time, length: MemoryLayout<Float>.size, index: 2)
        // Pass the amplitude to the compute shader.
        computeEncoder.setBytes(&amplitude, length: MemoryLayout<Float>.size, index:

        // Dispatch the compute shader.
        computeEncoder.dispatchThreadgroups(threadgroups, threadsPerThreadgroup: thr
    }
}
```

The sample defines the `makeComputePipeline` method as follows:

```
/// The device Metal selects as the default.
let metalDevice: MTLDevice? = MTLCreateSystemDefaultDevice()

...

/// Makes a compute pipeline for the compute function with the given name.
func makeComputePipeline(named name: String) -> MTLComputePipelineState? {
    guard let function = metalDevice?.makeDefaultLibrary()?.makeFunction(name: name)
        return nil
    }
    return try? metalDevice?.makeComputePipelineState(function: function)
}
```

See Creating a dynamic height and normal map with low-level texture for further details, along with a description of how the sample derives surface normal directions from the height map.

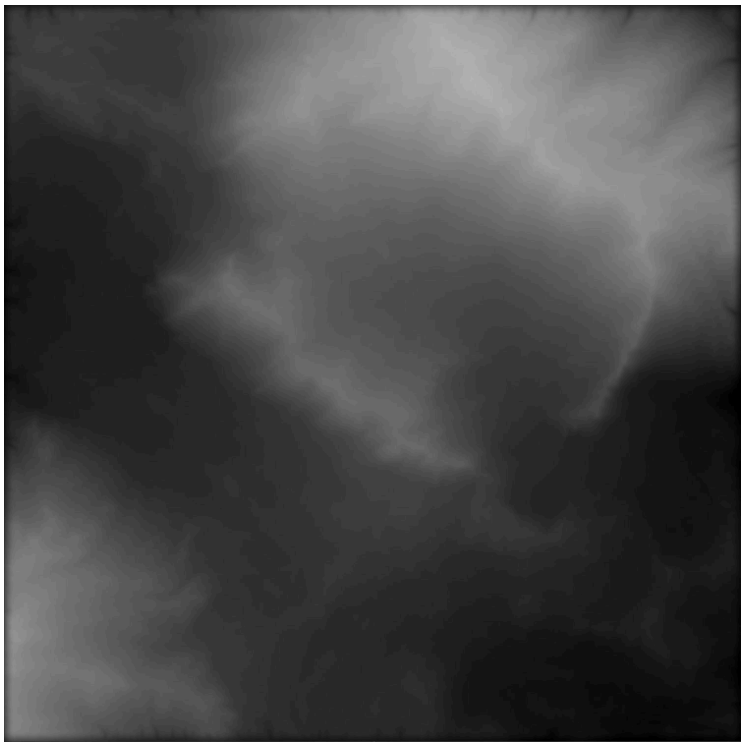
Generate additional height maps

The sample abstracts the generation of height maps by defining a `HeightMapGenerator` protocol:

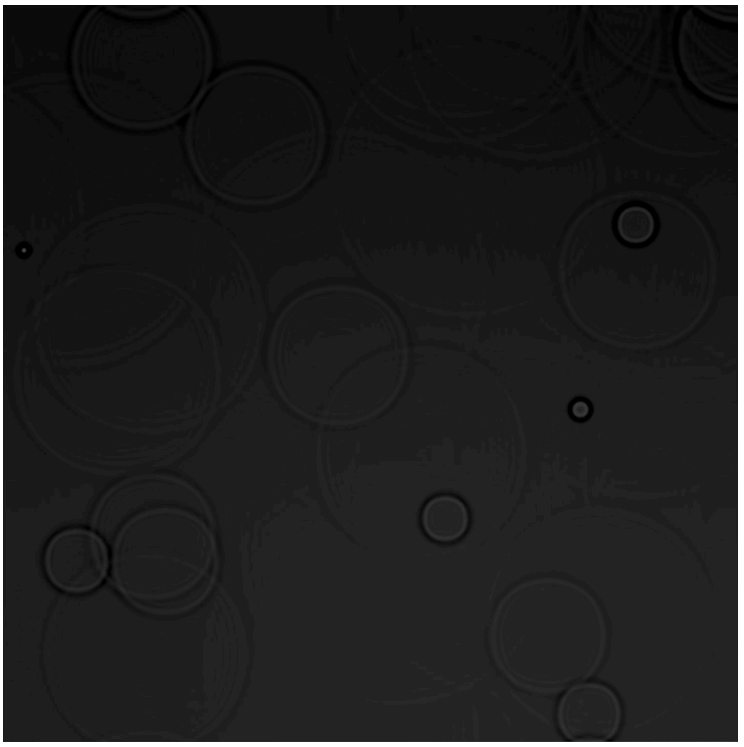
```
protocol HeightMapGenerator {
    /// Resets the height map.
    func reset()

    /// Generates the height map.
    func generateHeightMap(computeContext: ComputeUpdateContext,
                           heightMapTexture: LowLevelTexture,
                           heightMapComputeParams: HeightMapComputeParams)
}
```

You can adopt this protocol to create custom height map generators. These height map generators can generate custom height maps by dispatching compute shader functions, with the given `computeContext`, that write height data to the alpha channel of the given `heightMapTexture`.

The following videos show the custom height map textures the `TerrainHeightMapGenerator` and `WaterSurfaceHeightMapGenerator` create:



Play ⊙



Play ⊙

The `HeightMap` structure calls the `generateHeightMap` method of its active height map generator, passing in the height map low-level texture and the compute update context necessary to dispatch compute shader functions with the texture:

```
struct HeightMap {
    ...

    /// The generator that generates the height values of the height map.
    var heightMapGenerator: HeightMapGenerator = SineWaveHeightMapGenerator()

    /// The low-level texture that stores the height and normal information of the h
    var heightMapTexture: LowLevelTexture

    ...

    /// Generates the height values in the alpha channel of the height map using the
    func generateHeight(computeContext: ComputeUpdateContext, heightMapComputeParams
        heightMapGenerator.generateHeightMap(computeContext: computeContext,
                                             heightMapTexture: heightMapTexture,
                                             heightMapComputeParams: heightMapComput
    }

    ...
}
```

Update the mesh vertex data on the GPU

The sample defines a structure containing the information necessary to update the mesh's vertex data on the GPU:

```
struct MeshParams {
    simd_uint2 dimensions;
    simd_float2 size;
    float maxVertexDepth;
};
```

These parameters are suitable for updating the vertices of the plane low-level mesh on the GPU, but you can define parameters that conform to your custom mesh and vertex format. Declare this structure in an MSL header file so that you can send it to the GPU (see Passing Structured Data to a Metal Compute Function).

Next, the sample updates the position and normal of each vertex in the low-level mesh with a compute shader function that reads this information from the height map low-level texture:

```
[[kernel]]
void setVertexData(constant MeshParams &params [[buffer(0)]],
                   device PlaneVertex *vertices [[buffer(1)]],
                   texture2d<float, access::read> heightMap [[texture(2)]],
                   uint2 vertexCoords [[thread_position_in_grid]]) {
    // Skip out-of-bounds threads.
    // https://developer.apple.com/documentation/metal/compute_passes/calculating_th
    if (any(vertexCoords >= params.dimensions)) { return; }

    // Calculate the 1D vertex buffer index given its 2D x, y coordinates.
    uint vertexIndex = vertexCoords.x + params.dimensions.x * vertexCoords.y;
    // Get the current vertex.
    device PlaneVertex &vert = vertices[vertexIndex];

    // Sample the height map pixel corresponding to this vertex.
    float4 heightMapData = heightMap.read(vertexCoords);
    // Extract the normal direction and the height.
    float3 normal = heightMapData.rgb;
    float height = heightMapData.a;

    // Convert the x and y vertex coordinates to the range [0, 1].
    float2 vertexCoords01 = float2(vertexCoords) / float2(params.dimensions - 1);
```

```
    // Get the x and y position from the size.
    float2 xyPosition = params.size * vertexCoords01 - params.size / 2;
    // Get the z position from the height, clamping it within
    // the bounds of the mesh that `maxVertexDepth` defines.
    float zPosition = clamp(height, 0., params.maxVertexDepth);

    // Update the vertex position and normal.
    vert.position = float3(xyPosition, zPosition);
    vert.normal = normal;
}
```

Here, the `maxVertexDepth` parameter defines the maximum z offset position for vertices, so that they remain within the bounds of the mesh (see Creating a plane with low-level mesh). You can take your own approach to ensuring your vertices remain within bounds.

The sample passes the mesh parameters, vertex buffer, and height map to the compute function before dispatching it:

```
private func updateVertices(computeContext: ComputeUpdateContext) {
    // Set the compute shader pipeline to `setVertexData`.
    computeContext.computeEncoder.setComputePipelineState(setVerticesPipeline)

    // Pass the mesh parameters to the compute shader.
    computeContext.computeEncoder.setBytes(&meshParams, length: MemoryLayout<MeshPa
    // Pass the vertex buffer to the compute shader.
    let vertexBuffer = planeMesh.mesh.replace(bufferIndex: 0, using: computeContext.
    computeContext.computeEncoder.setBuffer(vertexBuffer, offset: 0, index: 1)
    // Pass the height map to the compute shader.
    computeContext.computeEncoder.setTexture(heightMap.heightMapTexture.read(), inde

    // Dispatch the compute shader.
    computeContext.computeEncoder.dispatchThreadgroups(threadgroups, threadsPerThrea
}
```

Display the mesh with an entity

The sample creates a custom `HeightMapMeshEntity` class with a `ModelComponent` to display the `HeightMapMesh`:

```
class HeightMapMeshEntity: Entity, HasModel {
    /// The height map mesh this entity renders.
```

```
    var heightMapMesh: HeightMapMesh?

    /// Sets up the entity by creating a `HeightMapMesh` and adding the necessary co
    private func setup(size: SIMD2<Float>, dimensions: SIMD2<UInt32>, maxVertexDepth
        // Try to create a `HeightMapMesh` and get its low-level mesh.
        guard let heightMapMesh = try? HeightMapMesh(size: size, dimensions: dimensi
              let planeMesh = try? MeshResource(from: heightMapMesh.planeMesh.mesh)
            assertionFailure("Failed to create height map mesh and get its low-level
            return
        }
        self.heightMapMesh = heightMapMesh

        // Add a compute system component with the height map mesh as its compute sy
        self.components.set(ComputeSystemComponent(computeSystem: heightMapMesh))

        // Add a model component with the plane mesh.
        self.components.set(ModelComponent(mesh: planeMesh, materials: [SimpleMateri

        // Make this entity capable of receiving gestures by giving it an input targ
        self.components.set(InputTargetComponent())
        let collisionBoxDepth: Float = 0.025
        let collisionBox = ShapeResource.generateBox(width: size.x, height: size.y,
            .offsetBy(translation: [0, 0, -collisionBoxDepth / 2])
        self.components.set(CollisionComponent(shapes: [collisionBox]))
    }

    /// The custom initializer.
    ///
    /// Sets up the `heightMapMesh` with given size, dimensions, and maximum vertex
    init(size: SIMD2<Float>, dimensions: SIMD2<UInt32>, maxVertexDepth: Float) {
        super.init()
        setup(size: size, dimensions: dimensions, maxVertexDepth: maxVertexDepth)
    }

    /// The default initializer.
    required init() {
        super.init()
        setup(size: [1, 1], dimensions: [512, 512], maxVertexDepth: 1)
    }
}
```

The following video shows a `HeightMapMeshEntity` displaying the mesh its `HeightMapMesh` generates with the `SineWaveHeightMapGenerator`.

Play ⊙

Make the mesh and texture interactive

The sample allows the mesh to respond to a person's interactions by passing interaction data to the GPU, which uses it to modify the height map, which in turn updates the vertices of the mesh.

To start, the sample captures the person's interaction position and state with a `DragGesture`, passing that information to `HeightMapMesh`:

```
.gesture(
    DragGesture()
        .targetedToEntity(heightMapMeshEntity)
        .onChanged({ value in
            let interactionPosition = value.convert(value.location3D,
                                                    from: .local,
                                                    to: heightMapMeshEntity)
            heightMapMeshEntity.heightMapMesh?.interactionPosition = interactionPosi
            heightMapMeshEntity.heightMapMesh?.isInteractionHappening = true
```

```
        })
        .onEnded({ value in
            heightMapMeshEntity.heightMapMesh?.isInteractionHappening = false
        })
)
```

`HeightMapMesh` passes this interaction information to the active `HeightMapGenerator`, which can use it to generate its height map.

For example, `WaterSurfaceHeightMapGenerator` takes the interaction position and passes it to a compute shader with a custom structure that the sample defines in an MSL header file:

```
struct WaterParams {
    float deltaTime;
    float waterSpeed;
    simd_float2 disturbancePosition;
    float disturbanceRadius;
    float disturbanceAmount;
    simd_uint2 dimensions;
    simd_float2 size;
    simd_float2 cellSize;
};
```

It dispatches a compute shader to disturb the height of the water at the `interactionPosition` whenever an interaction is happening, by storing the interaction position in this structure's `disturbancePosition` property:

```
class WaterSurfaceHeightMapGenerator: HeightMapGenerator {
    ...

    // Disturbs the water surface by dispatching a compute shader that increases/dec
    // of the water around the disturbance position.
    func disturbWaterSurface(computeContext: ComputeUpdateContext,
                             heightMapTexture: LowLevelTexture,
                             heightMapComputeParams: HeightMapComputeParams,
                             waterParams: inout WaterParams) {
        // Dispatch the disturbance compute function.
        computeContext.computeEncoder.setComputePipelineState(disturbWaterSurfacePip
        computeContext.computeEncoder.setBytes(&waterParams, length: MemoryLayout<Wa
        computeContext.computeEncoder.setTexture(heightMapTexture.read(), index: 1)
        computeContext.computeEncoder.setTexture(heightMapTexture.replace(using: com
        computeContext.computeEncoder.dispatchThreadgroups(heightMapComputeParams.th
```

```
                                                        threadsPerThreadgroup: he
    }

    ...

    func generateHeightMap(computeContext: ComputeUpdateContext,
                           heightMapTexture: LowLevelTexture,
                           heightMapComputeParams: HeightMapComputeParams) {
        ...

        // Disturb the water surface downward at the position the person is interact
        // if an interaction is happening.
        if heightMapComputeParams.isInteractionHappening {
            waterParams.disturbancePosition = simd_make_float2(heightMapComputeParam
            waterParams.disturbanceRadius = 7 * waterParams.cellSize.x
            waterParams.disturbanceAmount = 250 * waterParams.cellSize.x * waterPara
            disturbWaterSurface(computeContext: computeContext,
                                heightMapTexture: heightMapTexture,
                                heightMapComputeParams: heightMapComputeParams,
                                waterParams: &waterParams)
        }

        ...
    }
}
```

The `disturbWaterSurface` compute shader function subtracts height from the height map around the disturbance position, simulating the effect of the person's interaction pushing the water downward:

```
[[kernel]]
void disturbWaterSurface(constant WaterParams &params [[buffer(0)]],
                         texture2d<float, access::read> heightMapIn [[texture(1)]],
                         texture2d<float, access::write> heightMapOut [[texture(2)]]
                         uint2 pixelCoords [[thread_position_in_grid]]) {
    // Skip out-of-bounds threads.
    // https://developer.apple.com/documentation/metal/compute_passes/calculating_th
    if (any(pixelCoords >= params.dimensions)) { return; }

    // Get the current state of the height map.
    float4 heightMapData = heightMapIn.read(pixelCoords);

    // Convert the position of the current pixel to the same coordinate space as the
```

```
    float2 currentPosition = float2(remap(pixelCoords.x, float2(0, params.dimensions
                                    remap(pixelCoords.y, float2(0, params.dimensions
    // Disturb the height of the water closer to the disturbance position.
    float distance = length(currentPosition-params.disturbancePosition);
    if (distance <= params.disturbanceRadius) {
        heightMapData.a -= params.disturbanceAmount * pow((params.disturbanceRadius-
    }

    // Write modified height map data back to the height map.
    heightMapOut.write(heightMapData, pixelCoords);
}
```

The following video shows the result of using the person's interaction position to dynamically alter the height map:

Play ⊙

The app uses a similar technique to allow the person to edit the terrain height map, like in the following video:

Play ⊙

See Also

Scene content

{} Hello World

Use windows, volumes, and immersive spaces to teach people about the Earth.

{} Enabling video reflections in an immersive environment

Create a more immersive experience by adding video reflections in a custom environment.

{} Creating a spatial drawing app with RealityKit

Use low-level mesh and texture APIs to achieve fast updates to a person's brush strokes by integrating RealityKit with ARKit and SwiftUI.

{} Combining 2D and 3D views in an immersive app

Use attachments to place 2D content relative to 3D content in your visionOS app.

{} Transforming RealityKit entities using gestures

Build a RealityKit component to support standard visionOS gestures on any entity.

{} Responding to gestures on an entity

Respond to gestures performed on RealityKit entities using input target and collision components.

Models and meshes

Display virtual objects in your scene with mesh-based models.

Materials, textures, and shaders

Apply textures to the surface of your scene's 3D objects to give each object a unique appearance.

Anchors

Lock virtual content to the real world.

Lights and cameras

Control the lighting and point of view for a scene.

Content synchronization

Synchronize the contents of entities locally or across the network.

Audio

Create personalized and realistic spatial audio experiences.

Videos

Present videos in your RealityKit experiences.

Images

Present images and spatial scenes in your RealityKit experiences.