Sample Code

Customizing render pass setup

Render into an offscreen texture by creating a custom render pass.

Download

iOS 12.0+ | iPadOS 12.0+ | macOS 10.13+ | tvOS 12.0+ | Xcode 12.3+

Overview

A render pass is a sequence of rendering commands that draw into a set of textures. This sample executes a pair of render passes to render a view's contents. For the first pass, the sample creates a custom render pass to render an image into a texture. This pass is an *offscreen render pass*, because the sample renders to a normal texture, rather than one created by the display subsystem. The second render pass uses a render pass descriptor, provided by the `MTKView` object, to render and display the final image. The sample uses the texture from the offscreen render pass as source data for the drawing command in the second render pass.

Offscreen render passes are fundamental building blocks for larger or more complicated renderers. For example, many lighting and shadow algorithms require an offscreen render pass to render shadow information and a second pass to calculate the final scene lighting. Offscreen render passes are also useful when performing batch processing of data that doesn't need to be displayed onscreen.

Create a texture for the offscreen render pass

An `MTKView` object automatically creates drawable textures to render into. The sample also needs a texture to render into during the offscreen render pass. To create that texture, it first creates an `MTLTextureDescriptor` object and configures its properties.

```
MTLTextureDescriptor *texDescriptor = [MTLTextureDescriptor new];
texDescriptor.textureType = MTLTextureType2D;
texDescriptor.width = 512;
texDescriptor.height = 512;
texDescriptor.pixelFormat = MTLPixelFormatRGBA8Unorm;
texDescriptor.usage = MTLTextureUsageRenderTarget |
                      MTLTextureUsageShaderRead;
```

The sample configures the `usage` property to state exactly how it intends to use the new texture. It needs to render data into the texture in the offscreen render pass and read from it in the second pass. The sample specifies this usage by setting the `renderTarget` and `shaderRead` flags.

Setting usage flags precisely can improve performance, because Metal can configure the texture's underlying data only for the specified uses.

Create the render pipelines

A render pipeline specifies how to execute a drawing command, including the vertex and fragment functions to execute, and the pixel formats of any render targets it acts upon. Later, when the sample creates the custom render pass, it must use the same pixel formats.

This sample creates one render pipeline for each render pass, using the following code for the offscreen render pipeline:

```
pipelineStateDescriptor.label = @"Offscreen Render Pipeline";
pipelineStateDescriptor.sampleCount = 1;
pipelineStateDescriptor.vertexFunction =  [defaultLibrary newFunctionWithName:@"simp
pipelineStateDescriptor.fragmentFunction =  [defaultLibrary newFunctionWithName:@"si
pipelineStateDescriptor.colorAttachments[0].pixelFormat = _renderTargetTexture.pixel
_renderToTextureRenderPipeline = [_device newRenderPipelineStateWithDescriptor:pipel
```

The code to create the pipeline for the drawable render pass is similar to that found in Drawing a triangle with Metal 4. To guarantee that the two pixel formats match, the sample sets the descriptor's pixel format to the view's `colorPixelFormat`. Similarly, when creating the offscreen render pipeline, the sample sets the descriptor's pixel format to the offscreen texture's format.

Set up the offscreen render pass descriptor

To render to the offscreen texture, the sample configures a new render pass descriptor. It creates an `MTLRenderPassDescriptor` object and configures its properties. This sample renders to a single color texture, so it sets `colorAttachment[0].texture` to point to the offscreen texture:

```
_renderToTextureRenderPassDescriptor.colorAttachments[0].texture = _renderTargetText
```

The sample must also configure a *load action* and a *store action* for this render target.

```
_renderToTextureRenderPassDescriptor.colorAttachments[0].loadAction = MTLLoadActionC
_renderToTextureRenderPassDescriptor.colorAttachments[0].clearColor = MTLClearColorM

_renderToTextureRenderPassDescriptor.colorAttachments[0].storeAction = MTLStoreActic
```

A load action determines the initial contents of the texture at the start of the render pass, before the GPU executes any drawing commands. Similarly, a store action runs after the render pass completes, and determines whether the GPU writes the final image back to the texture. The sample configures a load action to erase the render target's contents, and a store action that stores the rendered data back to the texture. It needs to do the latter because the drawing commands in the second render-pass sample this data.

Metal uses load and store actions to optimize how the GPU manages texture data. Large textures consume lots of memory, and working on those textures can consume lots of memory bandwidth. Setting the render target actions correctly can reduce the amount of memory bandwidth the GPU uses to access the texture, improving performance and battery life. See Setting load and store actions for guidance.

A render pass descriptor has other properties not used in this sample that further modify the rendering process. For information on other ways to customize the render pass descriptor, see `MTLRenderPassDescriptor`.

Render to the offscreen texture

The sample has everything it needs to encode both render passes. It's important to understand how Metal schedules commands on the GPU before seeing how the sample encodes the render passes.

When an app commits a buffer of commands to a command queue, by default, Metal must act as if it executes commands sequentially. To increase performance and to better utilize the GPU, Metal can run commands concurrently, as long as doing so doesn't generate results inconsistent with sequential execution. To accomplish this, when a pass writes to a resource and a subsequent pass reads from it, as in this sample, Metal detects the dependency and automatically delays execution of the later pass until the first one completes. So, unlike Synchronizing CPU and GPU work, where the CPU and GPU needed to be explicitly synchronized, the sample doesn't need to do anything special. It simply encodes the two passes sequentially, and Metal ensures they run in that order.

The sample encodes both render passes into one command buffer, starting with the offscreen render pass. It creates a render command encoder using the offscreen render pass descriptor it previously created.

```
id<MTLRenderCommandEncoder> renderEncoder =
    [commandBuffer renderCommandEncoderWithDescriptor:_renderToTextureRenderPassDesc
renderEncoder.label = @"Offscreen Render Pass";
[renderEncoder setRenderPipelineState:_renderToTextureRenderPipeline];
```

Everything else in the render pass is similar to Drawing a triangle with Metal 4. It configures the pipeline and any necessary arguments, then encodes the drawing command. After encoding the command, it calls `endEncoding()` to finish the encoding process.

```
[renderEncoder endEncoding];
```

Multiple passes must be encoded sequentially into a command buffer, so the sample must finish encoding the first render pass before starting the next one.

Render to the drawable texture

The second render pass needs renders the final image. The drawable render pipeline's fragment shader samples data from a texture and returns that sample as the final color:

```
// Fragment shader that samples a texture and outputs the sampled color.
fragment float4 textureFragmentShader(TexturePipelineRasterizerData in      [[stage_
                                      texture2d<float>             texture [[textur
{
    sampler simpleSampler;

    // Sample data from the texture.
    float4 colorSample = texture.sample(simpleSampler, in.texcoord);

    // Return the color sample as the final color.
    return colorSample;
}
```

The code uses the view's render pass descriptor to create the second render pass, and encodes a drawing command to render a textured quad. It specifies the offscreen texture as the texture argument for the command.

```
id<MTLRenderCommandEncoder> renderEncoder =
    [commandBuffer renderCommandEncoderWithDescriptor:drawableRenderPassDescriptor];
renderEncoder.label = @"Drawable Render Pass";
```

```
[renderEncoder setRenderPipelineState:_drawableRenderPipeline];

[renderEncoder setVertexBytes:&quadVertices
                       length:sizeof(quadVertices)
                      atIndex:AAPLVertexInputIndexVertices];

[renderEncoder setVertexBytes:&_aspectRatio
                       length:sizeof(_aspectRatio)
                      atIndex:AAPLVertexInputIndexAspectRatio];

// Set the offscreen texture as the source texture.
```

When the sample commits the command buffer, Metal executes the two render passes sequentially. In this case, Metal detects that the first render pass writes to the offscreen texture and the second pass reads from it. When Metal detects such a dependency, it prevents the subsequent pass from executing until the GPU finishes executing the first pass.

See Also

Render workflows

Using Metal to draw a view’s contents

Create a MetalKit view and a render pass to draw the view’s contents.

Drawing a triangle with Metal 4

Render a colorful, rotating 2D triangle by running draw commands with a render pipeline on a GPU.

Selecting device objects for graphics rendering

Switch dynamically between multiple GPUs to efficiently render to a display.

Creating a custom Metal view

Implement a lightweight view for Metal rendering that’s customized to your app’s needs.

Calculating primitive visibility using depth testing

Determine which pixels are visible in a scene by using a depth texture.

Encoding indirect command buffers on the CPU

Reduce CPU overhead and simplify your command execution by reusing commands.

{} Implementing order-independent transparency with image blocks

Draw overlapping, transparent surfaces in any order by using tile shaders and image blocks.

{} Loading textures and models using Metal fast resource loading

Stream texture and buffer data directly from disk into Metal resources using fast resource loading.

{} Adjusting the level of detail using Metal mesh shaders

Choose and render meshes with several levels of detail using object and mesh shaders.

{} Creating a 3D application with hydra rendering

Build a 3D application that integrates with Hydra and USD.

{} Culling occluded geometry using the visibility result buffer

Draw a scene without rendering hidden geometry by checking whether each object in the scene is visible.

{} Improving edge-rendering quality with multisample antialiasing (MSAA)

Apply MSAA to enhance the rendering of edges with custom resolve options and immediate and tile-based resolve paths.

{} Achieving smooth frame rates with a Metal display link

Pace rendering with minimal input latency while providing essential information to the operating system for power-efficient rendering, thermal mitigation, and the scheduling of sustainable workloads.