Sample Code

# 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.

Download

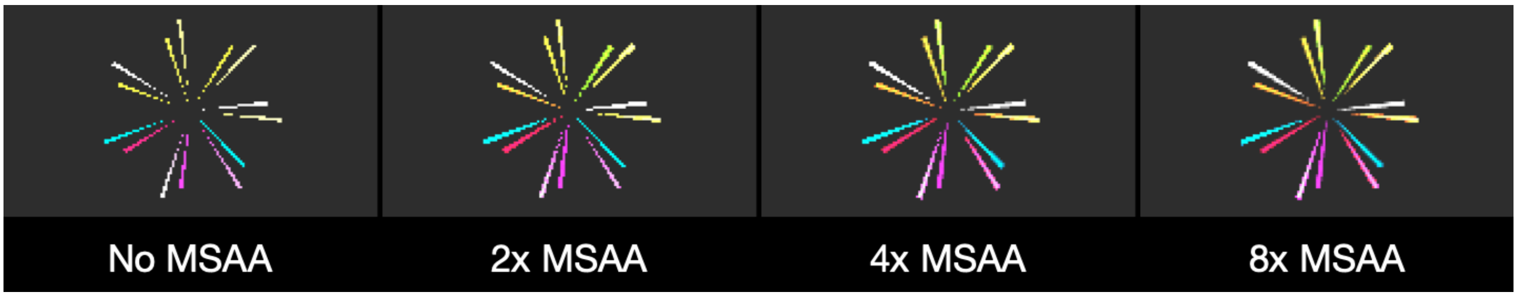
iOS 13.0+ | iPadOS 13.0+ | macOS 11.0+ | tvOS 14.5+ | Xcode 14.0+

## Overview

MSAA improves the rendering of lines and edges by using several color and depth samples per pixel. An MSAA texture normally stores two or four samples per pixel. But some GPU devices support eight samples per pixel. After the app renders a scene to the MSAA texture, it *resolves* it to a normal texture that contains one sample per pixel. The built-in resolve uses a simple average, but sometimes you want to use a custom resolve.

A custom resolve is essential when the built-in resolve isn't sufficient. For example, you may want to tone-map high dynamic range (HDR) samples before averaging them. HDR means that color values exceed the normal color intensity range of zero to one. Tone-mapping means that values are compressed into the zero-to-one range for output to a display device.

This sample shows you how to use a custom resolve with immediate-mode devices and tile-based deferred rendering devices. The immediate-mode custom resolve uses a compute kernel to process the MSAA texture, while a tile-based shader works before the rendering is completed, saving time and memory bandwidth. The following image shows the results of applying MSAA to a colored set of thin shards with one, two, four, and eight samples per pixel.



The sample displays a user interface for you to control the code paths to use MSAA. The app customizes the options presented based on the current device. Some devices support eight samples or the tile-based resolve. You can choose between the built-in resolve or a custom resolve that replicates the built-in choice or applies a tone-mapping operator to the samples. You can choose between two, four, or eight samples per pixel and the immediate-mode path or the tile-based resolve.

There's a checkbox to toggle MSAA on and off. There's also a list of reduced resolutions—ranging from one-sixteenth resolution to full-resolution—that makes it easier to see the difference MSAA makes. Pressing the space bar on a macOS device or tapping the display on an iOS device causes the shards to rotate.

## Configure the sample code project

To run this sample, you need Xcode 12 or later on a macOS, iOS, or tvOS device or simulator. To experience the tile-based custom resolve feature, you need a physical device that supports `MTLGPUFamily.apple4`, such as:

- A Mac with Apple silicon running macOS 11 or later
- An iOS device with an A11 chip or later running iOS 13 or later
- A tvOS device with an A12 chip or later running tvOS 14.5 or later

On a device with an Intel or AMD GPU, or on a simulator, this sample runs with the custom resolve in immediate-mode rendering (IMR) only.

## Check for MSAA support

GPUs on Apple devices can support a variety of sample counts but must support at least a sample count of four. The app checks the device's supported sample count with `supportsTextureSampleCount(_:)` and uses a default sample count of four.

```
NSUInteger sampleCounts[AAPLSampleCountOptionsCount] = { 2, 4, 8 };
for (int i = 0; i < AAPLSampleCountOptionsCount; i++)
{
    sampleCountSupported[i] = [_view.device supportsTextureSampleCount:sampleCounts[
}
```

## Create a multisample texture

The app needs to create a multisample texture before rendering the scene. The following code shows that the texture declaration is the same as a normal texture.

```
id<MTLTexture> _multisampleTexture;
```

The app configures the texture using a texture descriptor. It uses `MTLTextureType.type2DMultisample` as the texture type, and sets the `sampleCount` property with the current count selected in the UI.

```
_multisampleTextureDescriptor.textureType = MTLTextureType2DMultisample;
_multisampleTextureDescriptor.sampleCount = _antialiasingSampleCount;
```

If the app is running on a device that supports tile shaders, then the usage and storage mode enable the renderer to take advantage of memoryless storage. That means the device discards the texture when it finishes the tile-based resolve. In comparison, the immediate-mode custom resolve requires the texture to stay in memory, and the renderer uses a separate compute pass to resolve the MSAA texture. The app sets the shader-read flag to allow the compute kernel to access the texture. The following code shows how to set the `usage` and `storageMode` properties in the descriptor.

```
if (_resolvingOnTileShaders)
{
    _multisampleTextureDescriptor.usage = MTLTextureUsageRenderTarget;
    _multisampleTextureDescriptor.storageMode = MTLStorageModeMemoryless;
}
else
{
    _multisampleTextureDescriptor.usage = MTLTextureUsageRenderTarget | MTLTextureUs
    _multisampleTextureDescriptor.storageMode = MTLStorageModePrivate;
}
```

The renderer also creates a resolve texture to store the final image that'll be copied to the drawable texture. The following code shows how it uses the same width and height for both textures. The `_resolveTextureDescriptor` uses a single sample for the resolve texture.

```
_multisampleTextureDescriptor.width = _viewportSize.x;
_multisampleTextureDescriptor.height = _viewportSize.y;

_multisampleTexture = [_device newTextureWithDescriptor:_multisampleTextureDescripto

_multisampleTexture.label = @"Multisampled Texture";

_resolveTextureDescriptor.width = _viewportSize.x;
_resolveTextureDescriptor.height = _viewportSize.y;

_resolveResultTexture = [_device newTextureWithDescriptor:_resolveTextureDescriptor]

_resolveResultTexture.label = @"Resolved Texture";
```

## Configure the render pipeline

The renderer ensures the render pipeline state object (PSO) has the current sample count as the texture to use as a render target. It also decides the correct fragment function depending on whether HDR is enabled or not. The app uses a separate custom resolve step if HDR is enabled, so it doesn't apply tone-mapping. But if antialiasing is disabled, then it uses a fragment function that applies tone-mapping. If the HDR resolve option is chosen in the UI, then toggling MSAA on and off shows the result of using tone-mapping. The following code shows how the renderer customizes the render pipeline descriptor before it creates the PSO.

```
if (_antialiasingEnabled)
{
    _renderPipelineDescriptor.sampleCount = _antialiasingSampleCount;
    _renderPipelineDescriptor.fragmentFunction = _fragmentFunctionNonHDR;
}
else
{
    _renderPipelineDescriptor.sampleCount = 1;
    _renderPipelineDescriptor.fragmentFunction = _usesHDR ? _fragmentFunctionHDR : _
}
_renderPipelineState = [_device newRenderPipelineStateWithDescriptor:_renderPipeline
```

After the the renderer sets up the PSO, it also sets up the render pass descriptor to use the multisample texture as a color attachment. If MSAA isn't enabled, then it sets up the `_resolveResultTexture` as the main color attachment. But if MSAA is enabled, then the following code shows how the renderer chooses the store action. If tile shaders are available or you choose the built-in resolve, then Metal resolves the texture without needing an extra compute pass. The code also shows how to prepare the render pass for a resolve using tile shaders that the README discusses later.

```
if (_resolvingOnTileShaders)
{
    renderPassDescriptor.tileWidth = AAPLTileWidth;
    renderPassDescriptor.tileHeight = AAPLTileHeight;
    renderPassDescriptor.imageblockSampleLength = 32;
}

MTLStoreAction storeAction = shouldResolve ? MTLStoreActionMultisampleResolve : MTLS
renderPassDescriptor.colorAttachments[0].storeAction = storeAction;
renderPassDescriptor.colorAttachments[0].texture = _multisampleTexture;
renderPassDescriptor.colorAttachments[0].resolveTexture = shouldResolve ? _resolveRe
```

## Use a custom resolve in IMR

The built-in resolve averages the color samples in the texture together to create the final pixel color in the resolve texture. But developers can use a custom resolve by using a compute kernel. Devices that don’t support tile-based rendering need to use a compute kernel. The compute kernel reads the multisample texture and writes a normal texture. The following code shows a compute kernel that accumulates the samples and calculates their average.

```
/// A custom resolve kernel that averages color at all sample points.
kernel void
averageResolveKernel(texture2d_ms<float, access::read> multisampledTexture [[texture
                     texture2d<float, access::write> resolvedTexture [[texture(1)]],
                     uint2 gid [[thread_position_in_grid]])
{
    const uint count = multisampledTexture.get_num_samples();

    float4 resolved_color = 0;

    for (uint i = 0; i < count; ++i)
    {
        resolved_color += multisampledTexture.read(gid, i);
    }

    resolved_color /= count;

    resolvedTexture.write(resolved_color, gid);
}
```

The following code shows how to resolve an MSAA texture with the compute kernel. It sets the two input textures for the kernel and dispatches the work.

```
// Resolve the multisample texture with the chosen custom filter.
id<MTLComputeCommandEncoder> computeEncoder = [commandBuffer computeCommandEncoder];

computeEncoder.label = @"Resolve on Compute";

[computeEncoder setComputePipelineState:_resolveComputePipelineState];

[computeEncoder setTexture:_multisampleTexture atIndex:0];
[computeEncoder setTexture:_resolveResultTexture atIndex:1];

[computeEncoder dispatchThreadgroups:_threadgroupsInGrid
               threadsPerThreadgroup:_intrinsicThreadgroupSize];

[computeEncoder endEncoding];
```

## Use a custom resolve to tone-map an HDR image

Images with HDR need a tone-mapping operator before they're output to a computer display. The sample shows how to use a custom resolve to prepare the image. The following code shows the simple tone-mapping operator that uses luminance to scale the input color.

```
/// Tone-maps an input color by calculating its Rec. 709 luminance and applying a si
half3 tonemapByLuminance(half3 inColor)
{
    const half3 kRec709Luma(0.2126h, 0.7152h, 0.0722h);

    const half luminance = dot(inColor, kRec709Luma);

    return inColor / (1 + luminance);
}
```

The sample uses a different kernel function, `hdrResolveKernel`, that tone-maps the samples before it takes the average. Developers may choose to use a similar approach for applications where two mathematical functions don't commute. In this case, tone-mapping the average of four samples isn't necessarily the same numerical result as averaging four tone-mapped samples.

```
for (uint i = 0; i < count; ++i)
{
    const half4 sampleColor = multisampledTexture.read(gid, i);

    const half3 tonemappedColor = tonemapByLuminance(sampleColor.xyz);

    resolved_color += half4(tonemappedColor, 1);
}
```

## Use a tile-based resolve on an Apple silicon device

On Apple silicon devices, this app can resolve a multisample texture using a tile-based resolve, allowing the device to perform the resolve before the device saves the tile to memory. The app gains two advantages: The multisample texture doesn't need to be stored, saving memory space, and it doesn't require a second render or compute pass, saving compute time and memory bandwidth.

Not all devices support tile shaders, so the app uses the following code to check whether the device supports the feature.

```
/// Checks whether the device supports tile shaders that were introduced with Apple
- (BOOL)supportsTileShaders
{
    return [_device supportsFamily:MTLGPUFamilyApple4];
}
```

If the device supports the feature, then the renderer creates a new PSO that uses the `_averageResolveTileKernelFunction`. The following code shows how to set the kernel function, pixel format, sample count, and whether the kernel function expects the tile size to match the thread group size.

```
_resolveTileRenderPipelineDescriptor = [MTLTileRenderPipelineDescriptor new];

_resolveTileRenderPipelineDescriptor.label = @"CustomResolvePipeline";
_resolveTileRenderPipelineDescriptor.tileFunction = _averageResolveTileKernelFunctio
_resolveTileRenderPipelineDescriptor.threadgroupSizeMatchesTileSize = YES;
_resolveTileRenderPipelineDescriptor.colorAttachments[0].pixelFormat = _renderTarget

_resolveTileRenderPipelineDescriptor.rasterSampleCount = _antialiasingSampleCount;

_resolveTileRenderPipelineState = [_device newRenderPipelineStateWithTileDescriptor:
```

```
                                                                              options:
                                                                           reflection:
                                                                                error:
NSAssert(_resolveTileRenderPipelineState, @"Failed aquiring pipeline state: %@", er
```

After it creates the PSO, the renderer can dispatch the work before it finishes encoding the work. The following code shows how the app avoids this step when it uses the built-in resolve.

```
if (_antialiasingEnabled && _resolveOption != AAPLResolveOptionBuiltin)
{
    // Resolve MSAA with a custom resolve filter.
    [renderEncoder setRenderPipelineState:_resolveTileRenderPipelineState];

    [renderEncoder dispatchThreadsPerTile:MTLSizeMake(16, 16, 1)];
}

[renderEncoder endEncoding];
```

Finally, the following code shows a custom kernel function that resolves the MSAA texture when the input texture has HDR samples. The other custom kernel function in the app is similar, but it only averages the sample values, like the built-in resolve.

```
/// A tile kernel for a custom MSAA resolve that applies tone-mapping to HDR color s
kernel void
hdrResolveTileKernel(imageblock<FragData> img_blk_colors,
                     ushort2 tid [[thread_position_in_threadgroup]])
{

    const ushort pixelColorCount = img_blk_colors.get_num_colors(tid);

    half4 resolved_color = half4(0);

    for (ushort i = 0; i < pixelColorCount; ++i)
    {
        const half4 color = img_blk_colors.read(tid, i, imageblock_data_rate::color)

        const ushort sampleColorCount = popcount(img_blk_colors.get_color_coverage_m

        const half3 tonemappedColor = tonemapByLuminance(color.xyz);

        resolved_color += half4(tonemappedColor, 1) * sampleColorCount;
    }
    resolved_color /= img_blk_colors.get_num_samples();

    const ushort output_sample_mask = 0xF;

    img_blk_colors.write(FragData{resolved_color}, tid, output_sample_mask);
}
```

## 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.

{} Customizing render pass setup

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

{} 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.

{} 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.