Sample Code

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.

Download

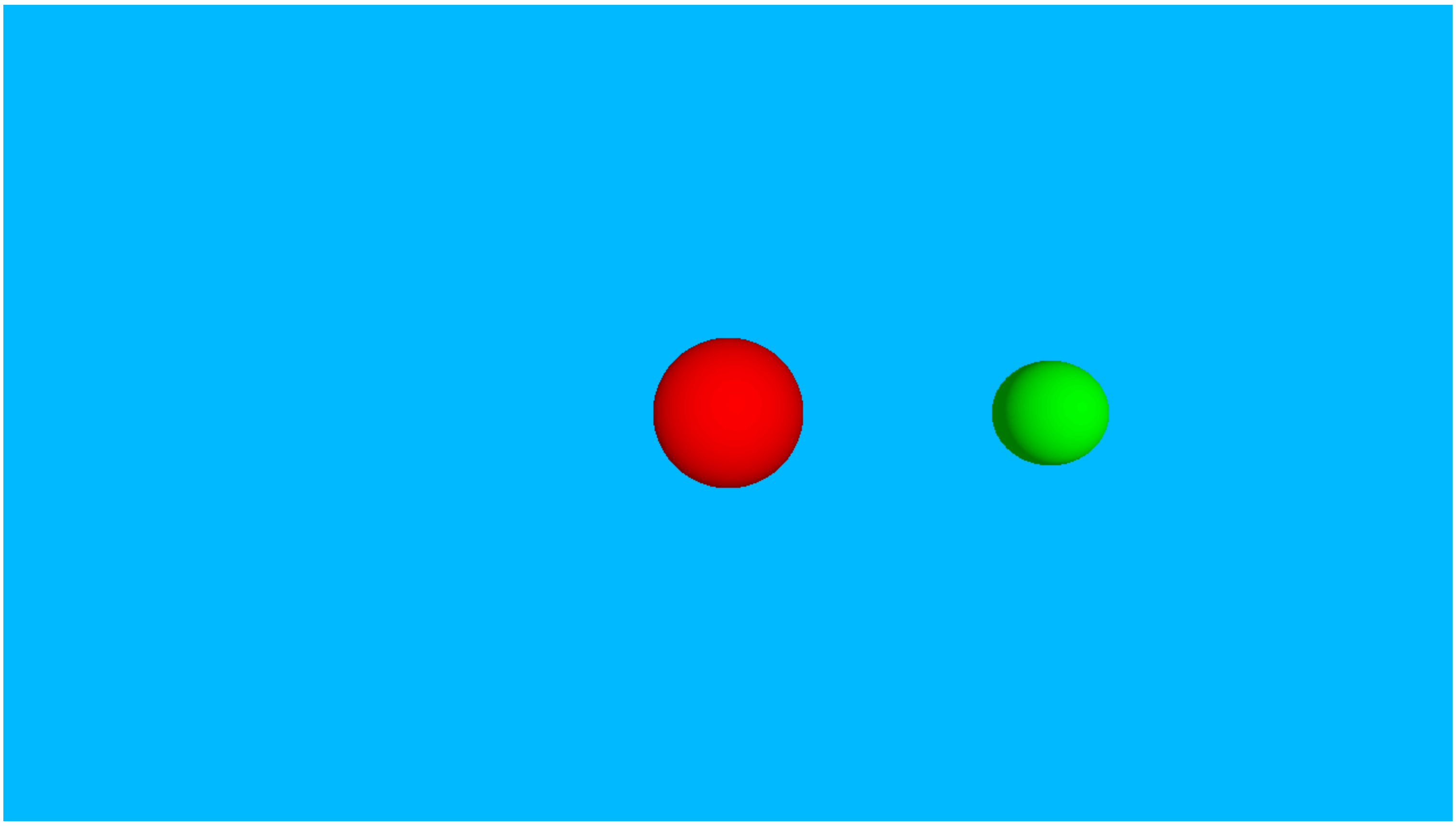
iOS 15.0+ | iPadOS 15.0+ | macOS 11.0+ | tvOS 13.0+ | Xcode 14.0+

Overview

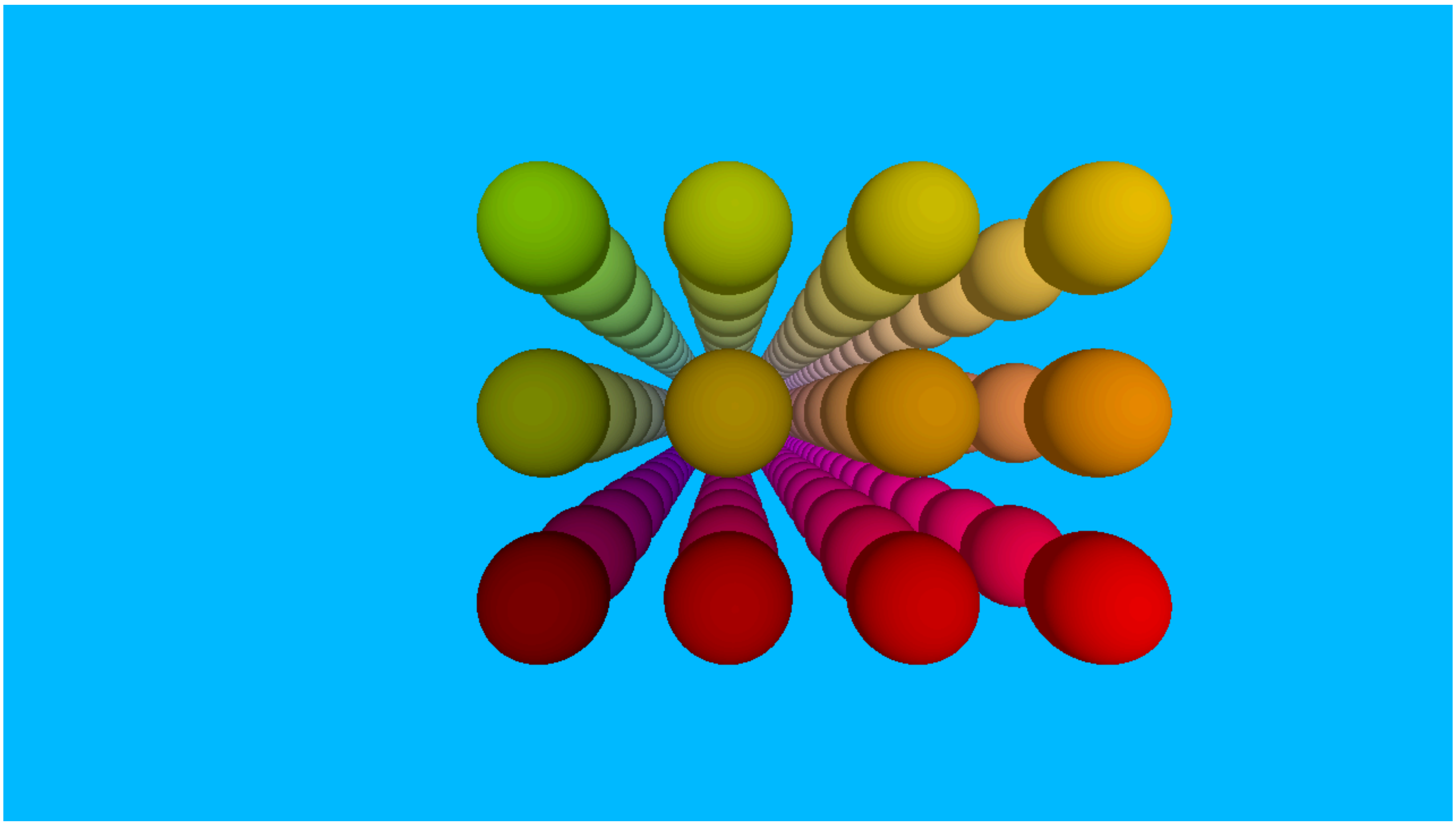
This sample demonstrates how to use a visibility result buffer to perform *fragment counting* and *occlusion culling*. The visibility result buffer stores an array of integers that contain the number of times a fragment passes the depth and stencil tests. Before a draw call, the app sets the location of the 64-bit counter in the buffer and the hardware increments the counter during rasterization.

The first mode of the app shows how to use fragment counting to determine the number of pixels that pass the depth and stencil tests. You can use this information to choose the level of detail for a 3D model, or to include or exclude the model from rendering. The second mode shows how to use the occlusion-culling technique to reduce the rendering load by eliminating invisible objects. To use this technique, the app renders a second pass of the objects it wants to test using proxy geometry instead of the full-resolution 3D model. It skips any object in the next frame if the number of pixels that passed the depth test is zero. You can toggle between the two modes by using the segment control in the upper-left corner of the window.

The fragment-counting mode shows a stationary red sphere and a smaller green sphere, and counts the number of fragments rendered for the green sphere. The slider on the lower-left corner of the app moves the green sphere left and right. The top-right corner of the window shows the number of fragments drawn from the green sphere. The number of fragments decrease to zero as the red sphere occludes the green sphere.



The occlusion-culling mode is more complex, rendering multiple spheres and using the visibility results to skip the draw calls for occluded spheres. It renders *icosahedrons*, a 20-sided polyhedron serving as a type of proxy geometry for a sphere, to the previous frame. If the fragment count is zero because it's *occluded* (hidden), the renderer *culls* (skips drawing) it in the next frame. The slider at the bottom left moves the spheres left and right, and the label in the upper right shows the number of rendered spheres.



The app creates Metal objects, renders 3D objects, and demonstrates two ways to use the visibility result buffer. First, it initializes the visibility result buffers and creates the sphere and icosahedron geometry. Next, it uses shared code to position and render the sphere and icosahedron geometry. Lastly, the app uses the state of the segment control to render either the fragment-counting mode or occlusion-culling mode.

Configure the sample code project

To run this sample, you need Xcode 12 and a physical device that supports `MTLGPUFamily.apple3`, such as:

- A Mac with Apple silicon running macOS 11 or later
- An iOS device with an A9 chip or later running iOS 12 or later
- A tvOS device with an A12 chip or later running tvOS 13 or later

This sample can only run on a physical device because it uses the counting occlusion query features that Simulator doesn't support.

Create the visibility result buffer

At initialization time, the app creates a set of visibility result buffers used in the fragment-counting and occlusion-culling modes. A visibility result buffer is an array of `uint64_t` integers stored in an `MTLBuffer`. The app allows up to three frames to be in flight with the `AAPLMaxBuffersInFlight` constant. However, if the app uses only a single buffer for recording and reading visibility results, a data race condition might occur if the GPU simultaneously writes to the visibility result buffer while the CPU reads from that same buffer. To prevent this, the app uses a ring of buffers that contains one more buffer than the number of frames in flight. The `AAPLNumVisibilityBuffers` constant stores this value. At the end of the `loadMetalWithView:` method, the app creates the buffers using the shared resource storage mode to allow CPU access.

```
// Initialize the visibility result buffers.
for (size_t i = 0; i < AAPLNumVisibilityBuffers; ++i)
{
    _visibilityBuffer[i] = [_device newBufferWithLength:AAPLNumObjectsXYZ * sizeof(u
                                                options:MTLResourceStorageModeShared
    _visibilityBuffer[i].label = @"visibilitybuffer";
}
```

Before rendering either of the two modes, the app chooses which visibility buffer to read from and write to. The `_visibilityBufferReadIndex` variable stores the index of the last completed visibility result buffer. The `_visibilityBufferWriteIndex` stores the index of the next visibility result buffer. The completion handler for the command buffer increments the read index as shown in the following code.

```
[commandBuffer addCompletedHandler:^(id<MTLCommandBuffer> buffer)
 {
    // Avoid a data race condition by updating the visibility buffer's read index wh
    self->_visibilityBufferReadIndex = (self->_visibilityBufferReadIndex + 1) % AAPL

    // Allow the app to start another frame by dispatching the in-flight semaphore.
    dispatch_semaphore_signal(block_sema);
}];
```

Before rendering, the `drawInMTKView:` method updates the current `_frameDataBuffer` and `_visibilityBuffer` indices for the frame. It also stores the pointer for the recently completed visibility result buffer in the `_readFromVisibilityResultBuffer` variable to use in both modes, as shown below.

```
// Update the state of the constant data buffers before rendering.
_frameDataBufferIndex = (_frameDataBufferIndex + 1) % AAPLMaxBuffersInFlight;
_visibilityBufferWriteIndex = (_visibilityBufferWriteIndex + 1) % AAPLNumVisibilityE

// Read the visibility buffer result from the previous frame.
_readFromVisibilityResultBuffer = _visibilityBuffer[_visibilityBufferReadIndex].cont
```

Create the proxy geometry

For the occlusion-culling mode, the app procedurally generates two geometric 3D primitives: a sphere and an icosahedron. The app uses an icosahedron as proxy geometry for the sphere in occlusion testing because it approximates a sphere using only 20 vertices.

Because proxy geometries are efficient to render, the occlusion culling uses them to quickly test visibility. If the visibility result buffer contains a nonzero value, the proxy geometry isn’t occluded and the app presumes the original geometry is also not occluded. However, to use proxy geometry for visibility testing, the 3D primitive must cover at least the same pixels as the original geometry. For the occlusion-culling mode, the app generates an icosahedron so that it fully inscribes (contains) a sphere.

Using proxy geometry that only covers the original model works well for stationary geometry but may not be sufficient for animated geometry. In the occlusion-culling mode, the sample uses the geometry visible in the current frame to determine the objects to draw on the next frame. This is an efficient approach for apps encoding work on the CPU. However, as objects change positions, the visibility results of one frame may not be accurate for the next frame, which may lead to objects popping in on subsequent frames.

To minimize this undesirable pop-in effect, the app scales the proxy geometry with an animation-adjustment factor. Increasing the size of the proxy geometry increases the screen area the app uses for testing the occluding geometry, making it less likely for the renderer to omit drawing occluded objects moving behind foreground objects that are visible in the next frame. Apps that choose larger-scale factors may reduce the amount of popping in but increase the draw count of occluded objects. Smaller-scale factors reduce the draw count but can cause objects farther away to pop into view more often.

The following code initializes the icosahedron radius by a scale factor of four. In the occlusion-culling mode, this factor reduces the amount of pop-in while also significantly reducing the number of spheres drawn.

```
/// A sphere's radius that allows plenty of space between spheres with a grid spacir
const double sphereRadius = 0.7;

/// An adjustment factor that scales up the proxy geometry size.
///
/// The sample sets the factor small enough to avoid a "pop in" effect when the sphe
/// In other words, the sphere may not be visible in the current frame, but visible
/// For example, a sphere that's moving behind another sphere may cause the occlusic
/// This causes the renderer to erroneously omit the sphere in the next frame.
///
/// The app uses a simple approach that sizes up the icosahedrons by an adjustment f
/// Sizing up the icosahedron gives each object extra pixels for the occlusion query
/// Since the renderer doesn't know exactly where the objects are located on the nex
/// this is like giving each object a little wiggle room.
/// If the adjustment value is too small, the renderer may not draw objects that are
/// If it's too large, then renderer may draw more spheres that are hidden by neare
/// A lower value of `1` is best for a stationary scene and does not size up the prc
/// A higher value of `8` sizes the proxy geometry significantly and reduces the occ
const double animationAdjustmentFactor = 4;

/// An icosahedron size that's large enough to inscribe a sphere.
///
/// The icosahedron radius includes the previous animation adjustment.
const double icosahedronRadius = animationAdjustmentFactor * sphereRadius;
```

Render the geometry

For convenience, the app stores all the matrices and colors for each object in a single array of `AAPLFrameDataPerObject` structures.

```
typedef struct
{
    matrix_float4x4 modelViewMatrix;
    matrix_float4x4 modelViewProjMatrix;
    vector_float3 color;
} AAPLFrameDataPerObject;

typedef struct
{
    AAPLFrameDataPerObject objects[AAPLNumObjectsXYZ];
} AAPLFrameData;
```

The vertex shader uses the `meshIndex` value to get the index to retrieve the matrices and colors for each object. During rendering, the following code shows both the `setVertexBytes` call to set the index and the `drawIndexedPrimitives` to draw the geometry.

```
/// Renders a mesh and sets the index of the mesh so the shader can reference its ap
- (void)renderMeshWithIndex:(uint32_t)meshIndex
{
    [_renderEncoder setVertexBytes:&meshIndex
                            length:sizeof(uint32_t)
                           atIndex:AAPLBufferIndexMeshIndex];
    [_renderEncoder drawIndexedPrimitives:MTLPrimitiveTypeTriangle
                               indexCount:_curMeshIndexCount
                                indexType:MTLIndexTypeUInt32
                              indexBuffer:_curMeshIndexBuffer
                        indexBufferOffset:0];
}
```

Count the visible fragments

The fragment-counting mode of the app counts the number of fragments rendered for the green sphere. It has two main steps performed by the `updateFragmentCountingMode:` and `renderFragmentCountingMode:` methods. The update method sets the projection and view matrices and the color for the first object, the red sphere. Then it sets up the matrices and color for the second object, the green sphere. Next comes the rendering phase in the `renderFragmentCountingMode:` method. Before the app renders the frame, it reads the first element from the visibility result buffer and copies it to the numVisibleFragments property. The view controller uses this value to update the label.

```
self->_numVisibleFragments = _readFromVisibilityResultBuffer[AAPLGreenSphereIndex];
```

The rendering process starts with configuring the visibilityResultBuffer in the render pass descriptor to point to the current write buffer.

```
// Add the visibility result buffer to this render pass.
renderPassDescriptor.visibilityResultBuffer = _visibilityBuffer[_visibilityBufferWri

// Create a render encoder for drawing.
_renderEncoder = [commandBuffer renderCommandEncoderWithDescriptor:renderPassDescrip
```

Next, it sets the render pipeline, depth state, and per-frame constant data buffers. The following code shows how the renderer draws the red and green spheres. It uses the setVisibilityResultMode(_:offset:) API to set the visibility result buffer mode to MTLVisibilityResultMode.counting. The offset into the buffer is defined by the constant AAPLGreenSphereIndex (which is set to 1), that means that the second element gets the result. Then it encodes the green sphere. When rendering completes, the visibility result buffer contains the number of fragments that passed the depth and stencil tests.

```
// Encode the sphere's vertex data.
[self setMeshBuffers:_sphereVertices indexBuffer:_sphereIndices indexCount:_sphereIn

// Encode the red sphere mesh for drawing.
[self renderMeshWithIndex:(uint32_t)AAPLRedSphereIndex];

// Set the offset into the visibility result buffer.
[_renderEncoder setVisibilityResultMode:MTLVisibilityResultModeCounting
                                 offset:AAPLGreenSphereIndex * sizeof(uint64_t)];

// Encode the green sphere for drawing.
[self renderMeshWithIndex:(uint32_t)AAPLGreenSphereIndex];
```

Skip the draw calls with occlusion culling

In the occlusion-culling mode, the app uses the visibility result buffer to determine which spheres to draw. Then it disables depth writes (but not the depth test) and tests proxy geometry against the image to generate visibility results for the next frame. It counts the number of drawn spheres in the `numSpheresDrawn` property for the view controller to use to update the UI label. Like the first mode, there's an update method `updateOcclusionCullingMode:` that sets the projection and model-view matrices and the color for each sphere.

The `renderOcclusionCullingMode:` method divides the occlusion testing into two parts: rendering the spheres and performing the occlusion query. In the following code, the app draws sphere `i` if the value `isVisibleResult[i]` is not zero.

```
// Encode the sphere's vertex data.
[self setMeshBuffers:_sphereVertices indexBuffer:_sphereIndices indexCount:_sphereIr

// Draw a visible sphere for every corresponding visible icosahedron.
for (size_t i = 0; i < AAPLNumObjectsXYZ; ++i)
{
    // If an icosahedron is visible, draw the sphere.
    if (_readFromVisibilityResultBuffer[i])
    {
        [self renderMeshWithIndex:(uint32_t)i];
        numDrawCalls++;
    }
}
```

After this loop, the occlusion query stage begins. First, the draw code disables color and depth writes using the `_pipelineStateNoRender` pipeline state object and the `_depthStateDisableWrites` depth stencil state. In addition, the `_pipelineStateNoRender` object doesn't have a fragment function because fragment processing isn't necessary to get visibility results.

```
// Configure the pipeline state object and depth state to disable writing to the col
[_renderEncoder setRenderPipelineState:_pipelineStateNoRender];
[_renderEncoder setDepthStencilState:_depthStateDisableWrites];
```

Then the render method encodes all the icosahedron draw calls. It calculates the offset into the visibility result buffer by multiplying the index of the icosahedron with the size of a `uint64_t`.

```
// Encode the icosahedron's vertices.
[self setMeshBuffers:_icosahedronVertices indexBuffer:_icosahedronIndices indexCount

// Draw each icosahedron and check its visibility.
for (size_t i = 0; i < AAPLNumObjectsXYZ; ++i)
{
    [_renderEncoder setVisibilityResultMode:MTLVisibilityResultModeBoolean
                                      offset:i * sizeof(uint64_t)];
    [self renderMeshWithIndex:(uint32_t)i];
}
```

When Metal renders the frame, the GPU updates the visibility result buffer with a nonzero value if any fragment passes the depth and stencil test. Drawing with proxy geometry allows the testing to remain efficient.

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.

{} 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.