Sample Code

Rendering reflections with fewer render passes

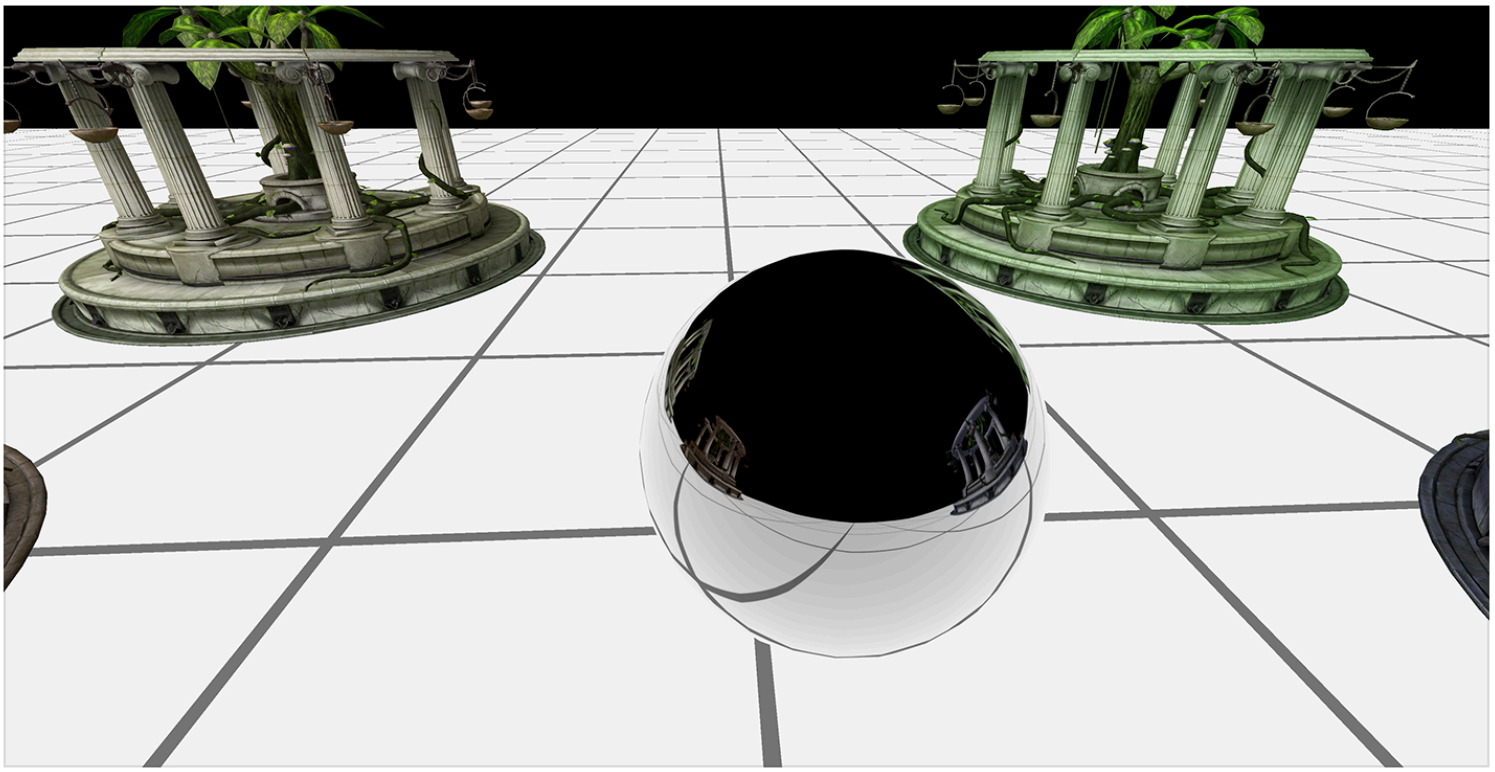
Use layer selection to reduce the number of render passes needed to generate an environment map.

Download

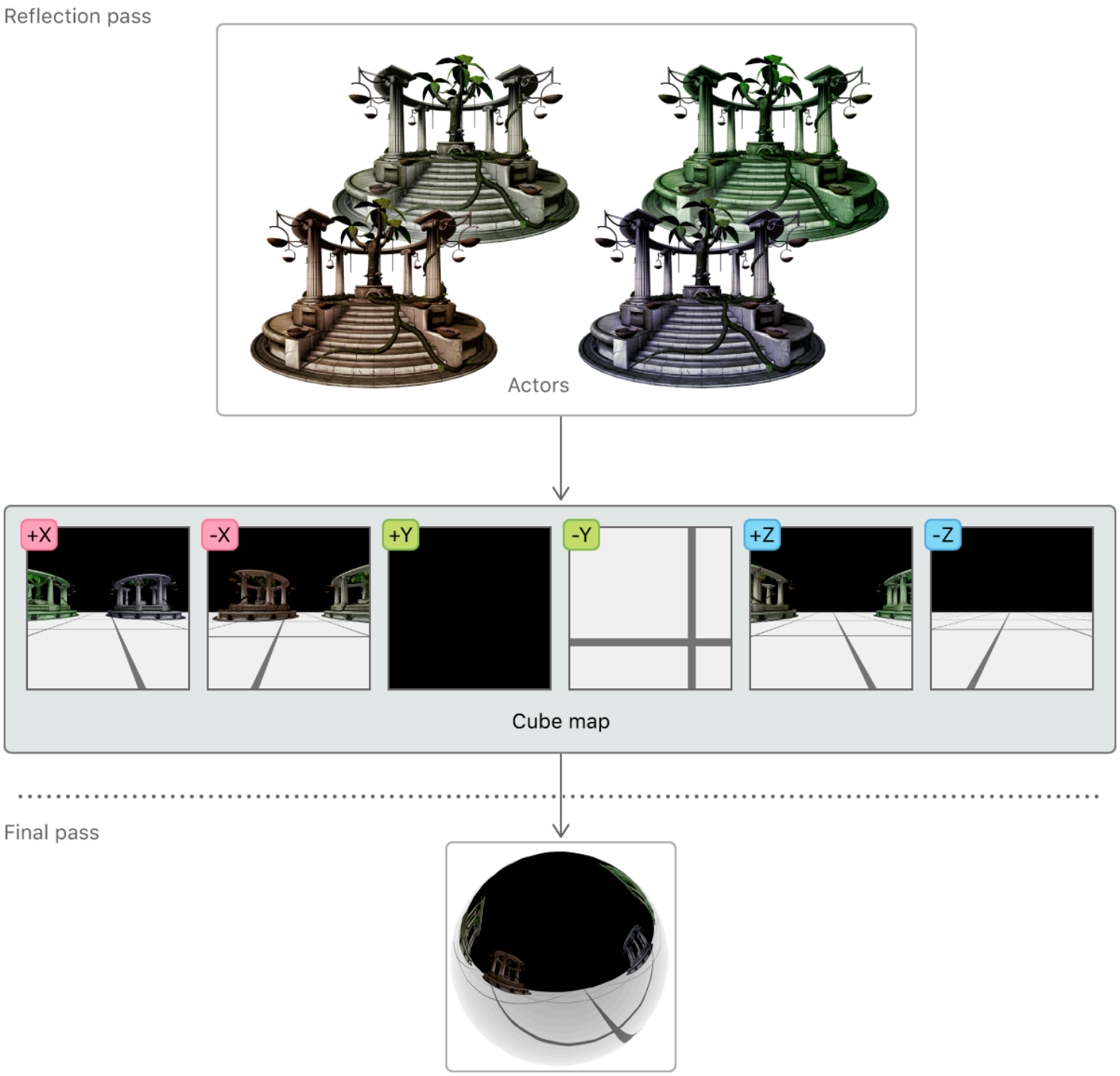
iOS 12.0+ | iPadOS 12.0+ | macOS 10.13+ | Xcode 12.3+

Overview

This sample demonstrates dynamic reflections on a chrome sphere, using layer selection to render the frame in two passes. The first pass renders the environment onto the cube map. The second pass renders the environment reflections onto the sphere; it renders additional actors in the scene; and it renders the environment itself.



You can implement an object that reflects its environment by sampling its reflections from a cube map of the environment. A cube map is a single texture composed of six 2D texture layers arranged in the shape of a cube. The reflections vary based on the positions of other objects in the environment, so each of the cube map's six faces must be rendered dynamically in every frame. This would normally require six separate render passes, one for each face, but Metal allows you to render an entire cube map in a single pass.



Getting started

This sample contains macOS and iOS targets. Run the iOS scheme on a physical device because Metal isn't supported in the simulator.

Layer Selection is supported on all macOS GPUs but only iOS GPUs which support the `MTLFeatureSet.iOS_GPUFamily5_v1` feature set.

You check the GPU that you choose at runtime supports indirect command buffers (ICBs) by calling an `MTLDevice` instance's `supportsFeatureSet(_:)` method.

```
supportsLayerSelection = [_view.device supportsFeatureSet:MTLFeatureSet_iOS_GPUFamil
```

This sample calls 'supportsFeatureSet:' for this purpose within its view controller's `viewDidLoad:` callback.

Separate the scene

A cube map is represented as a render target array with six layers, one for each of its faces. The `[[render_target_array_index]]` attribute qualifier, specified for a structure member of a vertex function return value, identifies each array layer separately. This layer selection feature allows the sample to decide which part of the environment gets rendered to which cube map face.

An `AAPLActorData` object represents an actor in the scene. In this sample, each actor is a temple model with the same mesh data but a different diffuse color. These actors sit on the XZ-plane; they're always reflected in the X or Z direction relative to the sphere and could be rendered to any of the +X, -X, +Z, or -Z faces of the cube map.

Perform culling tests for the reflection pass

Before rendering to the cube map, it's useful to know which faces each actor should be rendered to. Determining this information involves a procedure known as a *culling test*, and it's performed on each actor for each cube map face.

At the start of every frame, for each cube map face, a view matrix is calculated and the view's frustum is stored in the `culler_probe` array.

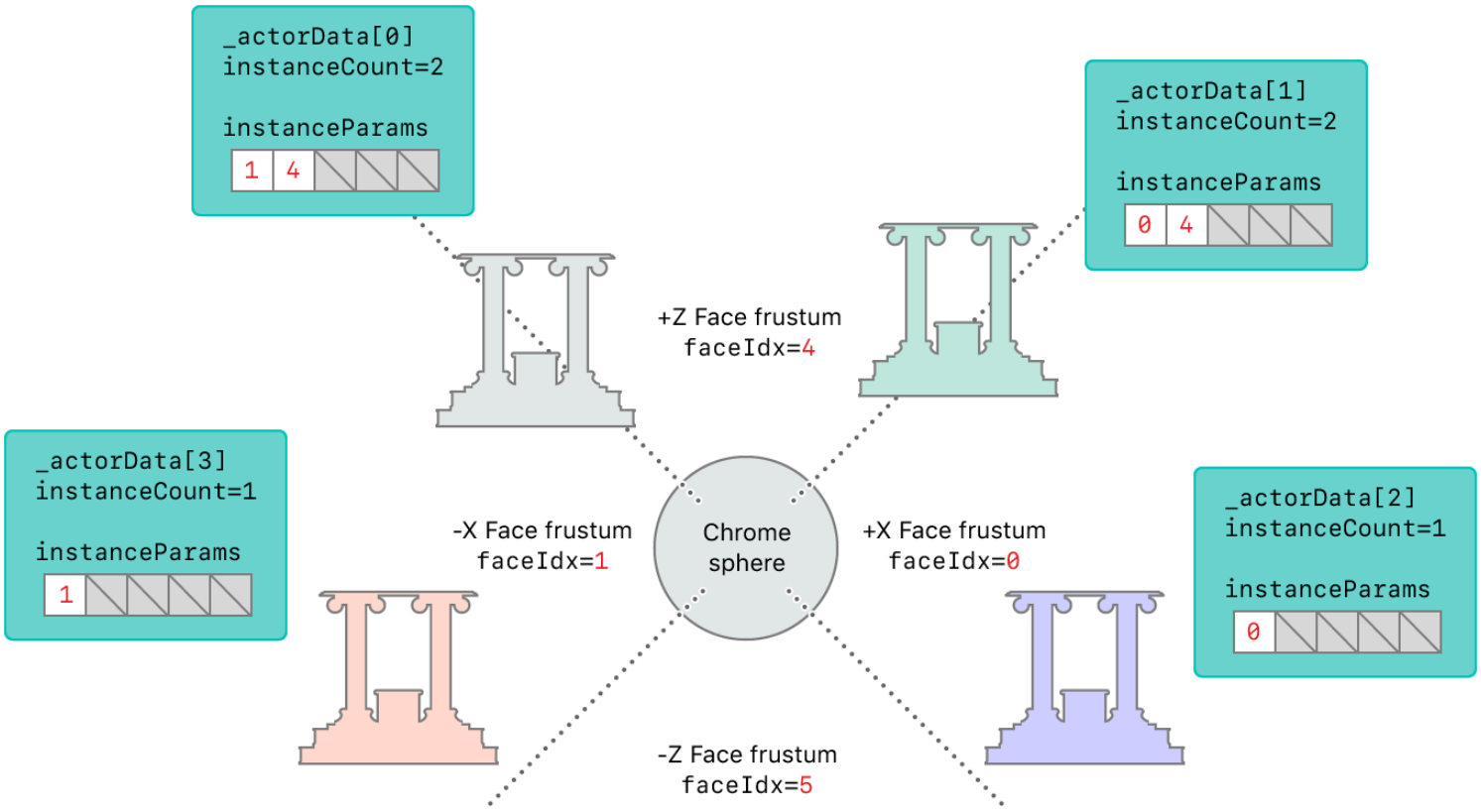
```
// 1) Get the view matrix for the face given the sphere's updated position
viewMatrix[i] = _cameraReflection.GetViewMatrixForFace_LH (i);

// 2) Calculate the planes bounding the frustum using the updated view matrix
//    You use these planes later to test whether an actor's bounding sphere
//    intersects with the frustum, and is therefore visible in this face's viewport
culler_probe[i].Reset_LH (viewMatrix [i], _cameraReflection);
```

These culler probes test the intersection between an actor and the viewing frustum of each cube map face. The test results determine how many faces the actor is rendered to (`instanceCount`) in the reflection pass, and which faces (`instanceParams`) it's rendered to.

```
if (_actorData[actorIdx].passFlags & EPassFlags::Reflection)
{
    int instanceCount = 0;
    for (int faceIdx = 0; faceIdx < 6; faceIdx++)
    {
```

```
        // Check if the actor is visible in the current probe face
        if (culler_probe [faceIdx].Intersects (_actorData[actorIdx].modelPosition.xy
        {
            // Add this face index to the the list of faces for this actor
            InstanceParams instanceParams = {(ushort)faceIdx};
            instanceParams_reflection [MaxVisibleFaces * actorIdx + instanceCount].v
            instanceCount++;
        }
    }
    _actorData[actorIdx].instanceCountInReflection = instanceCount;
}
```

The following diagram shows the results of the culling tests performed on the temple actors, based on their positions relative to the reflective sphere. Because `_actorData[0]` and `actorData[1]` bisect two viewing frustums, their `instanceCount` property is set to 2, and there are two elements in their `instanceParams` array. (This array contains the cube map face indices of the viewing frustums that the actors intersect.)



Configure render targets for the reflection pass

The render target for the reflection pass is a cube map. The sample configures the render target by using a `MTLRenderPassDescriptor` object with a color render target, a depth render target, and six layers. The `renderTargetArrayLength` property sets the number of cube map faces and allows the render pipeline to render into any or all of them.

```
reflectionPassDesc.colorAttachments[0].texture    = _reflectionCubeMap;
reflectionPassDesc.depthAttachment.texture        = _reflectionCubeMapDepth;
reflectionPassDesc.renderTargetArrayLength        = 6;
```

Issue draw calls for the reflection pass

The `drawActors:pass:` method sets up the graphics rendering state for each actor. Actors are only drawn if they are visible in any of the six cube map faces, determined by the `visibleVpCount` value (accessed through the `instanceCountInReflection` property). The value of `visibleVpCount` determines the number of instances for the instanced draw call.

```
[renderEncoder drawIndexedPrimitives: metalKitSubmesh.primitiveType
                          indexCount: metalKitSubmesh.indexCount
                           indexType: metalKitSubmesh.indexType
                         indexBuffer: metalKitSubmesh.indexBuffer.buffer
                   indexBufferOffset: metalKitSubmesh.indexBuffer.offset
                       instanceCount: visibleVpCount
                          baseVertex: 0
                        baseInstance: actorIdx * MaxVisibleFaces];
```

In this draw call, the sample sets the `baseInstance` parameter to the value of `actorIdx * 5`. This setting is important because it tells the vertex function how to select the appropriate render target layer for each instance.

Render the reflection pass

In the `vertexTransform` vertex function, the `instanceParams` argument points to the buffer that contains the cube map faces that each actor should be rendered to. The `instanceId` value indexes into the `instanceParams` array.

```
vertex ColorInOut vertexTransform (const Vertex in                              [[
                                   const uint   instanceId                      [[
                                   const device InstanceParams* instanceParams  [[
                                   const device ActorParams&    actorParams     [[
                                   constant     ViewportParams* viewportParams  [[
```

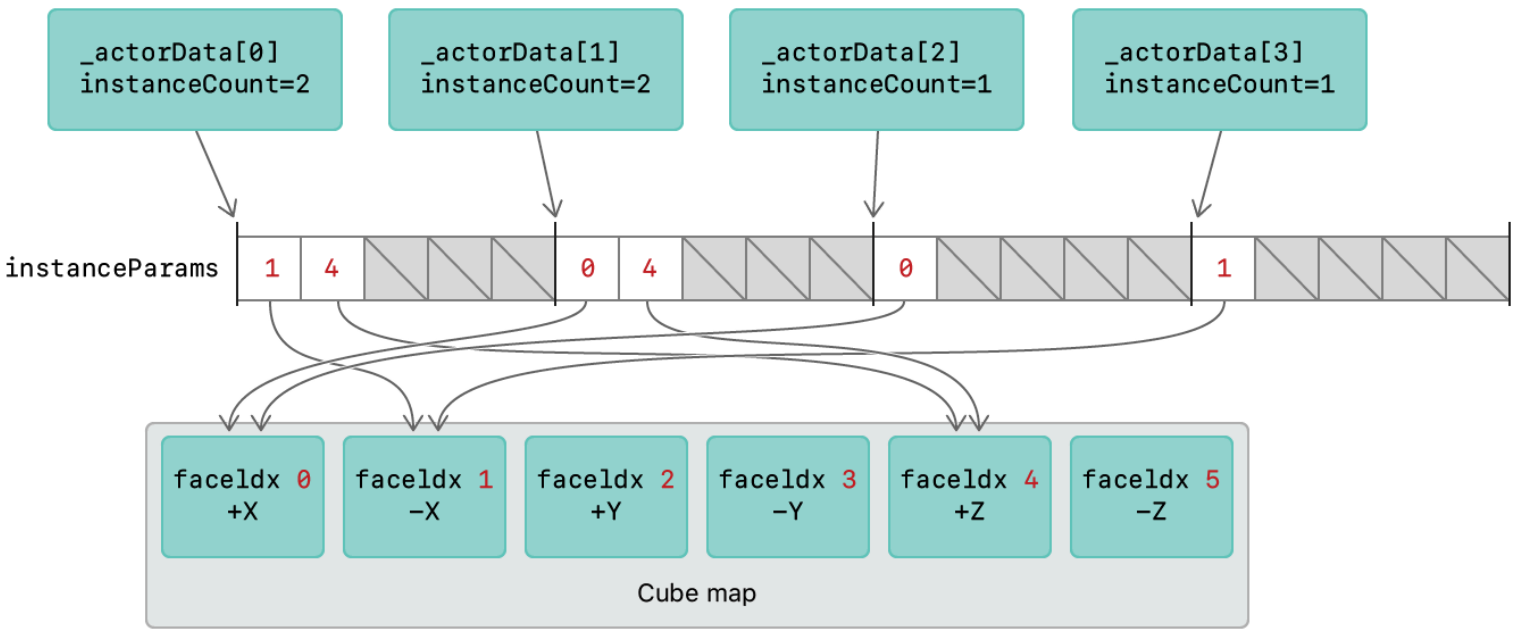
The output structure of the vertex function, `ColorInOut`, contains the `face` member that uses the `[[render_target_array_index]]` attribute qualifier. The return value of `face` determines the cube map face that the render pipeline should render to.

```
struct ColorInOut
{
    float4 position [[position]];
    float2 texCoord;

    half3  worldPos;
    half3  tangent;
    half3  bitangent;
    half3  normal;
    uint   face [[render_target_array_index]];
};
```

Because the value of the draw call's `baseInstance` parameter is set to `actorIdx * 5`, the `instanceId` value of the first instance drawn in the draw call is equal to this value. Each subsequent rendering of an instance increments the `instanceId` value by 1. The `instanceParams` array has five slots for each actor because an actor can be visible in up to five cube map faces. As a result, the `instanceParams[instanceId]` element always contains one of the face indices in which the actor is visible. Therefore, the sample uses this value to select a valid render target layer.

```
out.face = instanceParams[instanceId].viewportIndex;
```

In summary, to render each actor to the reflective cube map, the sample issues an instanced draw call for the actor. The vertex function uses the built-in `instanceId` variable to index into the `instanceParams` array that contains the index of the cube map face that the instance should be rendered to. Therefore, the vertex function sets this face index in the `face` return value member, which uses the `[[render_target_array_index]]` attribute qualifier. This ensures that each actor is rendered to each cube map face it should appear in.



Perform culling tests for the final pass

The sample performs similar view updates for the main camera in the final pass. At the start of every frame, a view matrix is calculated and the view's frustum is stored in the `culler_final` variable.

```
_cameraFinal.target   = SceneCenter;

_cameraFinal.rotation = fmod ((_cameraFinal.rotation + CameraRotationSpeed), M_PI*2.
matrix_float3x3 rotationMatrix = matrix3x3_rotation (_cameraFinal.rotation,  CameraF

_cameraFinal.position = SceneCenter;
_cameraFinal.position += matrix_multiply (rotationMatrix, CameraDistanceFromCenter);

const matrix_float4x4 viewMatrix       = _cameraFinal.GetViewMatrix();
const matrix_float4x4 projectionMatrix = _cameraFinal.GetProjectionMatrix_LH();

culler_final.Reset_LH (viewMatrix, _cameraFinal);

ViewportParams *viewportBuffer = (ViewportParams *)_viewportsParamsBuffers_final[_ur
viewportBuffer[0].cameraPos            = _cameraFinal.position;
viewportBuffer[0].viewProjectionMatrix = matrix_multiply (projectionMatrix, viewMatr
```

This final culler probe is used to test the intersection between an actor and the viewing frustum of the camera. The test result simply determines whether or not each actor is visible in the final pass.

```
if (culler_final.Intersects (_actorData[actorIdx].modelPosition.xyz, _actorData[acto
{
    _actorData[actorIdx].visibleInFinal = YES;
}
else
{
    _actorData[actorIdx].visibleInFinal = NO;
}
```

Configure render targets for the final pass

The render target for the final pass is the view's *drawable*, a displayable resource obtained by accessing the view's `currentRenderPassDescriptor` property. However, you must not access this property prematurely because it implicitly retrieves a drawable. Drawables are expensive system resources created and maintained by the Core Animation framework. You should

always hold a drawable as briefly as possible to avoid resource stalls. In this sample, a drawable is acquired just before the final render pass is encoded.

```
MTLRenderPassDescriptor* finalPassDescriptor = view.currentRenderPassDescriptor;

if(finalPassDescriptor != nil)
{
    finalPassDescriptor.renderTargetArrayLength = 1;
    id<MTLRenderCommandEncoder> renderEncoder =
    [commandBuffer renderCommandEncoderWithDescriptor:finalPassDescriptor];
    renderEncoder.label = @"FinalPass";

    [self drawActors: renderEncoder pass: EPassFlags::Final];

    [renderEncoder endEncoding];
}
```

Issue draw calls for the final pass

The `drawActors:pass:` method sets up the graphics rendering state for each actor. Actors are only drawn if they are visible to the main camera, as determined by the `visibleVpCount` value (accessed through the `visibleInFinal` property).

Because each actor is drawn only once in the final pass, the `instanceCount` parameter is always set to 1 and the `baseInstance` parameter is always set to 0.

Render the final pass

The final pass renders the final frame directly to the view's drawable, which is then presented onscreen.

```
[commandBuffer presentDrawable:view.currentDrawable];
```

See Also

Lighting techniques

{} Rendering a scene with forward plus lighting using tile shaders

Implement a forward plus renderer using the latest features on Apple GPUs.

{} Rendering a scene with deferred lighting in Objective-C

Avoid expensive lighting calculations by implementing a deferred lighting renderer optimized for immediate mode and tile-based deferred renderer GPUs.

{} Rendering a scene with deferred lighting in Swift

Avoid expensive lighting calculations by implementing a deferred lighting renderer optimized for immediate mode and tile-based deferred renderer GPUs.

{} Rendering a scene with deferred lighting in C++

Avoid expensive lighting calculations by implementing a deferred lighting renderer optimized for immediate mode and tile-based deferred renderer GPUs.