RealityKit

Custom Shader API

Developer

Overview 6

Understanding the Relationship Between Application Code and Shader Code 6

Rendering Scenes with Vertex Shaders and Fragment Shaders 7

Creating Custom Materials with Surface Shaders and Geometry Modifiers 7

Surface Shader APIs 8

surface_parameters 10

surface_parameters::textures() 12

surface_parameters::uniforms() 13

surface_parameters::geometry() 14

surface_parameters::material_constants() 15

surface_parameters::surface() 16

uniforms 17

uniforms::time() 18

uniforms::custom_parameter() 19

uniforms::model_to_world() 20

uniforms::model_to_view() 21

uniforms::world_to_view() 22

uniforms::view_to_projection() 23

uniforms::projection_to_view() 24

geometry 25

geometry::screen_position() 27

geometry::world_position() 28

geometry::model_position() 29

geometry::color() 30

geometry::normal() 31

geometry::tangent() 32

geometry::bitangent() 33

geometry::uv0() 34

geometry::uv1() 35

geometry::custom_attribute() 36

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

geometry::view_direction() 37
surface_properties 38
surface_properties::base_color() 40
surface_properties::set_base_color() 41
surface_properties::emissive_color() 42
surface_properties::set_emissive_color() 43
surface_properties::normal() 44
surface_properties::set_normal() 45
surface_properties::roughness() 47
surface_properties::set_roughness() 48
surface_properties::metallic() 49
surface_properties::set_metallic() 50
surface_properties::ambient_occlusion() 52
surface_properties::set_ambient_occlusion() 53
surface_properties::specular() 54
surface_properties::set_specular() 55
surface_properties::clearcoat() 56
surface_properties::set_clearcoat() 57
surface_properties::clearcoat_roughness() 58
surface_properties::set_clearcoat_roughness() 59
surface_properties::opacity() 61
surface_properties::set_opacity() 62

Geometry Modifier APIs 64

geometry_parameters 65
geometry_parameters::uniforms() 67
geometry_parameters::textures() 68
geometry_parameters::geometry() 69
geometry_parameters::material_constants() 70
uniforms 71
uniforms::time() 72
uniforms::custom_parameter() 73

Copyright © 2021 Apple Inc. All Rights Reserved.

uniforms::model_to_world() 74
uniforms::model_to_view() 75
uniforms::world_to_view() 76
uniforms::view_to_projection() 77
uniforms::projection_to_view() 78
geometry 79
geometry::vertex_id() 81
geometry::model_position() 82
geometry::world_position() 83
geometry::model_position_offset() 84
geometry::set_model_position_offset() 85
geometry::world_position_offset() 86
geometry::set_world_position_offset() 87
geometry::color() 88
geometry::set_color() 89
geometry::normal() 90
geometry::set_normal() 91
geometry::tangent() 92
geometry::set_tangent() 93
geometry::bitangent() 94
geometry::set_bitangent() 95
geometry::uv0() 96
geometry::set_uv0() 97
geometry::uv1() 98
geometry::set_uv1() 99

Shared APIs 100

textures 101
textures::base_color() 103
textures::emissive_color() 105
textures::normal() 107
textures::roughness() 108

textures::metallic() 109
textures::ambient_occlusion() 110
textures::specular 111
textures::opacity 112
textures::clearcoat() 114
textures::clearcoat_roughness() 116
textures::custom() 118
material_parameters 119
material_parameters::base_color_tint() 121
material_parameters::roughness_scale() 122
material_parameters::metallic_scale() 123
material_parameters::opacity_scale() 124
material_parameters::opacity_threshold() 125
material_parameters::emissive_color() 126
material_parameters::specular_scale() 127
material_parameters::clearcoat_scale() 128
material_parameters::clearcoat_roughness_scale() 129

Utility Functions 130

unpack_normal() 131
unpack_normal() 132
unpack_normal() 133

Overview

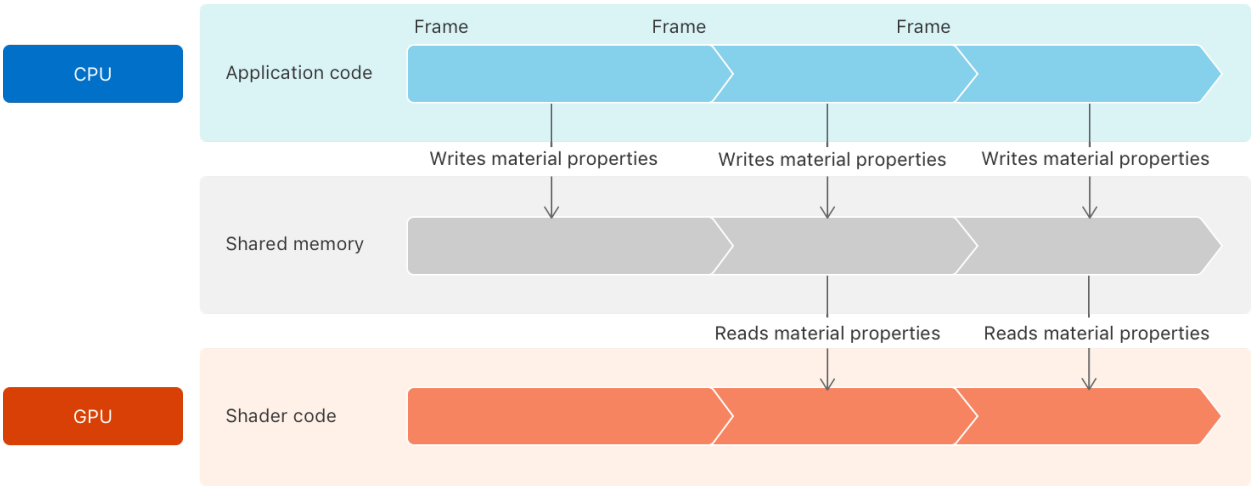
When you create a scene using RealityKit, the framework renders it for you based on the various material properties of the entities that make up your scene. It automatically leverages the device GPU to do rendering calculations quickly, and can render both realistic and stylized scenes by using different material structs, such as `PhysicallyBasedMaterial` or `UnlitMaterial`.

For many apps, RealityKit displays your scenes without you having to write any rendering code at all. To achieve certain rendering or special visual effects, however, you might want to alter the way RealityKit uses material properties and textures. RealityKit's `CustomMaterial`, available on macOS 12 and iOS 15 and later, enables you to do just that by writing Metal shader functions that run on the GPU.

Understanding the Relationship Between Application Code and Shader Code

To make effective use of custom materials, it's important to understand the relationship between your application code, which runs on the CPU, and Metal shader code, which runs on the GPU. Your application code never directly calls the shader functions you write for a custom material. Instead, you initialize the custom material with a reference to shader functions contained in your Xcode project. RealityKit automatically copies those functions over to the GPU, where they execute on every frame because they're called by RealityKit's shaders.

RealityKit copies the relevant values and textures from your custom material into memory that's shared between the CPU and GPU, which means that the Metal shaders can read them. RealityKit provides a Metal library with functions and data structures for retrieving those that data, as well as functions for specifying output values that RealityKit uses for final rendering.



Rendering Scenes with Vertex Shaders and Fragment Shaders

While your application code runs sequentially on its thread, shader code is designed to run concurrently to take advantage of the computing power of the GPU. For example, one of the types of shader that RealityKit uses internally to render a scene is called a *vertex shader*, which executes once for every vertex that makes the entity it's currently drawing. A vertex shader implements logic to do the calculations needed for only a single vertex, but a large number of vertex shaders execute simultaneously to do the calculations for many vertices at once. This makes shader code extremely fast for tasks that lend themselves to massively concurrent execution, like vertex processing.

Another type of shader RealityKit uses to render scenes is called a *fragment shader*, and it implements the logic to draw one fragment. A *fragment* is a single pixel for which the final rendered color is potentially affected by the entity it's drawing. A fragment isn't a pixel, however. It's different in two primary ways. First, not every pixel is necessarily a fragment for a particular entity. If RealityKit is drawing a small entity in the upper left corner of your screen, rendering it won't affect the color of pixels in the lower right part of your screen. Second, multiple fragments can contribute to the final color of a single screen pixel. If there are two entities, one in front of the other, for example, and the closer entity is translucent, both of those entity's fragments contribute to the final color of that one pixel.

Creating Custom Materials with Surface Shaders and Geometry Modifiers

Custom materials support two types of shader functions: surface shaders and geometry modifiers. *Surface shaders* are called by RealityKit's fragment shader and can override RealityKit's rendering calculations for each of the entity's fragments. *Geometry modifiers* are called by RealityKit's vertex shader and can override the properties of any of your entity's vertices, including their position, color, and UV coordinates, which are used to map textures on to the surface of your entity. Every custom material requires a *surface shader*, but a *geometry modifier* is optional.

This document lists all the RealityKit Metal APIs that you use to implement custom materials. The first section contains APIs available only to surface shaders. The second section contains APIs available only to geometry modifiers. The third section documents APIs available in both types of shader functions, and the last section contains utility functions that are available to both types of shaders but aren't part of a struct or class.

For more information on using shader functions with custom materials, see Modifying RealityKit Rendering Using Custom Materials.

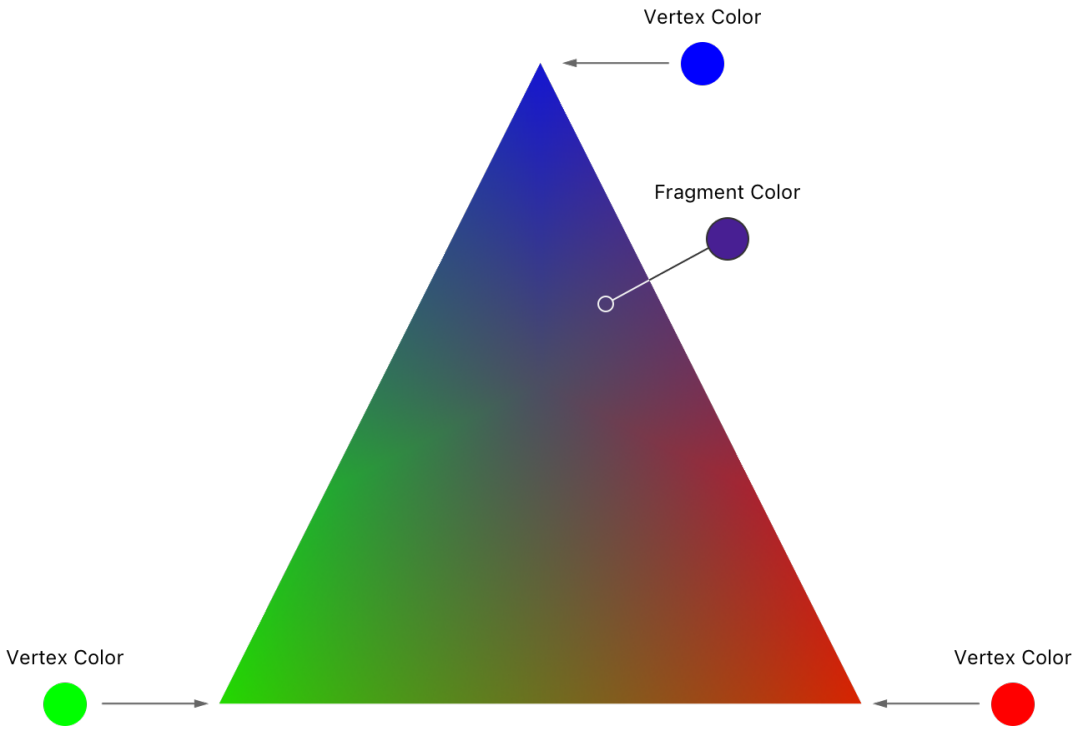
Surface Shader APIs

RealityKit's Metal shaders call a surface shader every frame during fragment shader execution. For entities that use a custom material, a material's surface shader is responsible for setting all the attributes RealityKit needs to render the entity, such as base color and roughness.

Receiving Input

A surface shader takes a single parameter of type `surface_parameters`. That parameter provides access to all the custom material's properties, as well as functions that specify the final value for various attributes.

Certain values that are available to your surface shader are specified per-vertex, such as vertex position, normal, and UV coordinates. When you access these values from a surface shader, Metal interpolates the per-vertex values based on the current fragment's position relative to the three vertices that make up its triangle. For example, vertex color is a per-vertex value. In a surface shader, vertex color is calculated from the vertex color values of the three vertices that make up the fragment's triangle. A color halfway between a red vertex and a blue vertex receives a purple vertex color, even though none of the vertices are actually purple, as the following illustration demonstrates.



Setting Output Values

RealityKit doesn't automatically use any of a custom material's properties to render an entity, which is unlike other material types. Your surface shader must specify the final calculated value for each rendering — for example, base color and roughness — for each fragment. That means that if you set the `baseColor` property on your custom material, your entity will render as white unless your surface shader calls `params.surface().set_base_color()`.

The surface shader outputs that RealityKit uses to render your entity depend on which `lightingModel` you selected when creating the material.

Lighting Model	Supported Output Functions
`.lit`	All except `set_clearcoat()` and `set_clearcoat_roughness()`.
`.clearcoat`	All.
`.unlit`	Only `set_emissive_color()`.

The properties on `CustomMaterial` largely mirror those on `PhysicallyBasedMaterial`, and in most situations, it's recommended to use the available properties to calculate the corresponding final output value. For example, you typically use the `baseColor` property on your material to provide the inputs for RealityKit to calculate base color. However, there's flexibility in how you use these inputs. You can choose how to use the material's texture and property inputs in your surface shader.

You might use two textures to animate your entity's base color over time by rotating the UV coordinates of one of the two textures. Because `CustomMaterial` only has one base color texture available, you'd use the `custom` attribute's texture, or the texture of an unused property (like `emissiveColor`) if you use the `custom` property for something else.

For more information on using shader functions with custom materials, see Taking Control of RealityKit Rendering Using Custom Materials.

Struct

surface_parameters

An object the framework uses to pass data into a surface shader.

Namespace

`realitykit`

Declaration

```
struct surface_parameters
```

Overview

The `surface_parameters` struct holds all the information that RealityKit passes into a surface shader function. Values are grouped into sub-objects of related types. For example, you can use the object returned by `textures()` to access textures from your entity's material and the object returned by `material_constants()` to access non-texture values, like the base color tint or roughness scale.

A surface shader must set output values by retrieving `surface()` and using its various `set_` functions to tell RealityKit how to render the entity. For example, to set the entity's base color, a shader function calls `params.surface().set_base_color()`.

RealityKit uses default values for any rendering attribute that your shader function doesn't call, so if a surface shader fails to call `set_base_color()`, for example, that results in RealityKit drawing your entity as a white object, regardless of the base color values you set on the custom material.

Member Functions

`texture::textures textures() const thread`

Retrieves an object the framework uses to pass textures from a custom material to a surface shader.

`surface::uniforms uniforms() const thread`

Retrieves an object the framework uses to pass entity-specific and global constant to shader functions.

`surface::geometry geometry() const thread`

Retrieves geometry properties for the current fragment.

`material::material_parameters material_constants() const thread`

An object the framework uses to pass constant, non-texture values from the entity's material to the surface shader.

`surface::surface_properties surface() thread`

Retrieves an object the surface shader uses to specify outputs values.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

surface_parameters::textures()

Retrieves an object the framework uses to pass textures from a custom material to a surface shader.

Declaration

```
texture::textures textures() const thread
```

Overview

This function returns a read-only object that provides access to all the textures from the shader's material.

Member Function

surface_parameters::uniforms()

Retrieves an object the framework uses to pass entity-specific and global constant to shader functions.

Declaration

```
surface::uniforms uniforms() const thread
```

Overview

This function returns a read-only object that provides access to values that are constant across all vertices and fragments. For example, it might return the material's custom value, global values — like the current elapsed time — as well as utilities the surface shader uses to convert between different coordinate spaces.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

surface_parameters::geometry()

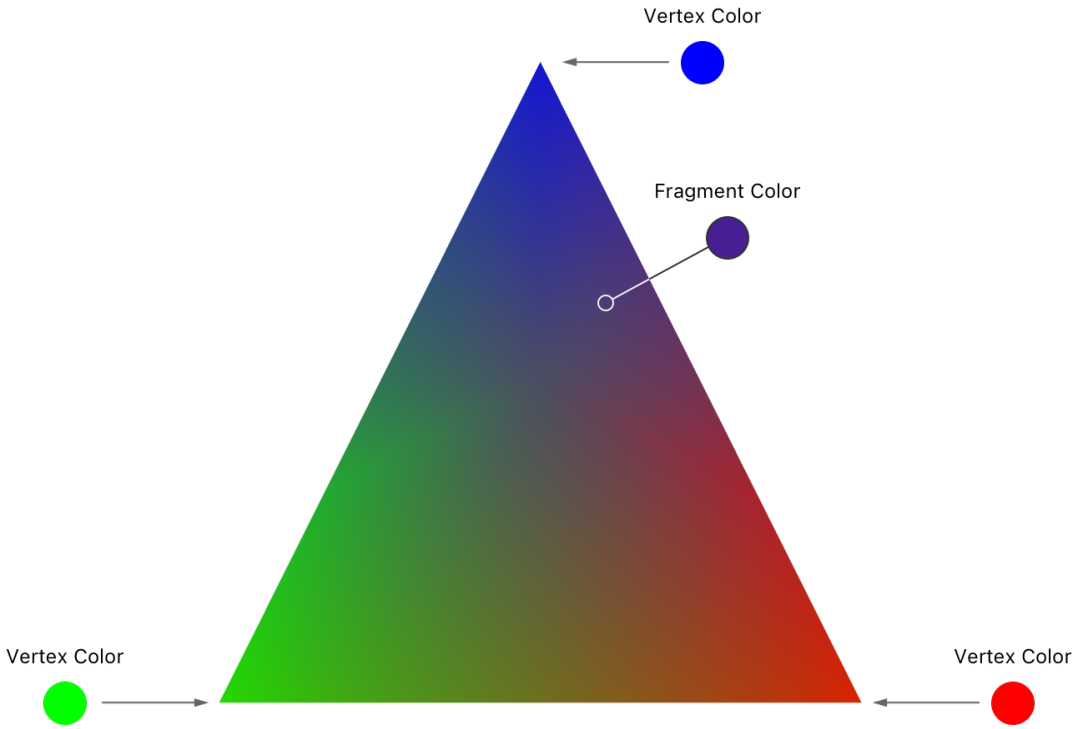
Retrieves geometry properties for the current fragment.

Declaration

```
surface::geometry geometry() const thread
```

Overview

This function returns a read-only object that holds per-vertex data for the entity, for example, the entity's UV coordinates, vertex positions, and vertex colors. The values that this object returns are interpolated by Metal for the current fragment, based on its position relative to the three vertices that make up its triangle:



Member Function

surface_parameters::material_constants()

An object the framework uses to pass constant, non-texture values from the entity's material to the surface shader.

Declaration

```
material::material_parameters material_constants() const thread
```

Overview

This function returns a read-only object the shader uses to retrieve non-texture values, such as base color tint and roughness scale, from the entity's material.

Member Function

surface_parameters::surface()

Retrieves an object the surface shader uses to specify outputs values.

Declaration

```
surface::surface_properties surface() thread
```

Overview

This function returns an object the surface shader uses to specify final rendering values — for example, base color and roughness — for the current fragment.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Struct

uniforms

An object the framework uses to pass constant values into a surface shader.

Namespace

`realitykit::surface`

Declaration

```
struct uniforms
```

Overview

This object holds values that are constant across all vertices and fragments, as well as matrices the surface shader uses to convert between different coordinate spaces.

Member Functions

`float time() const thread`

Returns the elapsed time in seconds.

`float4 custom_parameter() const thread`

Returns the custom vector from the entity's material.

`metal::float4x4 model_to_world() const thread`

Returns a matrix that transforms values from model space into world space.

`metal::float4x4 model_to_view() const thread`

Returns a matrix that transforms values from model space into view space.

`metal::float4x4 world_to_view() const thread`

Returns a matrix that transforms values from model space into view space.

`metal::float4x4 view_to_projection() const thread`

Returns a matrix that transforms values from view space into projection space.

`metal::float4x4 projection_to_view() const thread`

Returns a matrix that transforms values from projection space into view space.

Member Function

uniforms::time()

Returns the elapsed time in seconds.

Declaration

```
float time() const thread
```

Overview

This function returns the number of seconds that have elapsed since RealityKit began rendering the current scene.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

uniforms::custom_parameter()

Returns the custom vector from the entity's material.

Declaration

```
float4 custom_parameter() const thread
```

Overview

This Swift code demonstrates how to set the custom value using a `SIMD4<Float>` vector:

```
customMaterial.custom.value = SIMD4<Float>(x: 0.25,
                                           y: 0.25,
                                           z: 0.25,
                                           w: 1.0)
```

The following Metal code demonstrates how to retrieve that vector value in a surface shader:

```
float4 customVector = params.uniforms().custom_parameter();
```

You can also use this custom parameter to pass up to four individual, scalar values instead of a vector value. The following code demonstrates how to set individual values on a custom material in Swift:

```
customMaterial.custom.value[0] = 0.25
customMaterial.custom.value[1] = 0.75
```

This Metal code demonstrates how to retrieve those individual scalar values in a surface shader function:

```
float value = params.uniforms().custom_parameter()[0];
float otherValue = params.uniforms().custom_parameter()[1];
```

Member Function

uniforms::model_to_world()

Returns a matrix that transforms values from model space into world space.

Declaration

```
metal::float4x4 model_to_world() const thread
```

Overview

This function returns a matrix you can use to convert any model-space vector or matrix to world space using matrix multiplication. The following Metal code demonstrates how to use this matrix to convert a vector from model space to world space:

```
auto modelToWorld = params.uniforms().model_to_world();
float4 myVectorWorld = (myVector * modelToWorld);
```

Vertex-by-matrix multiplication requires vectors that are the same size as the number of columns in the matrix. This requirement means you can multiply a `float4` by `model_to_world`, but not a `float3`. If you need to convert a set of Cartesian coordinates stored in a `float3`, convert it to a `float4` before multiplying, by padding the vector with an extra `1.0` value, like this:

```
// This code assumes there's an existing float3 called myVertex.
float4(myVertex.x, myVertex.y, myVertex.z, 1.0);
auto modelToWorld = params.uniforms().model_to_world();
float3 myVectorWorld = (myVector * worldToView).xyz;
```

Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

uniforms::model_to_view()

Returns a matrix that transforms values from model space into view space.

Declaration

```
metal::float4x4 model_to_view() const thread
```

Overview

This function returns a matrix you can use to convert any model-space vector or matrix to view space using matrix multiplication. The following Metal code demonstrates how to use this matrix to convert a vector from model space to view space:

```
auto modelToView = params.uniforms().model_to_view();
float4 myVectorView = (myVector * modelToView);
```

Vertex-by-matrix multiplication requires vectors that are the same size as the number of columns in the matrix. This requirement means you can multiply a `float4` by `model_to_view`, but not a `float3`. If you need to convert a set of Cartesian coordinates stored in a `float3`, convert it to a `float4` before multiplying, by padding the vector with an extra `1.0` value, like this:

```
// This code assumes there's an existing float3 called myVertex.
float4(myVertex.x, myVertex.y, myVertex.z, 1.0);
auto modelToView = params.uniforms().model_to_view();
float3 myVectorView = (myVector * modelToView).xyz;
```

Member Function

uniforms::world_to_view()

Returns a matrix that transforms values from world space into view space.

Declaration

```
metal::float4x4 world_to_view() const thread
```

Overview

This function returns a matrix you can use to convert any world-space vector or matrix to view space using matrix multiplication. The following Metal code demonstrates how to use this matrix to convert a vector from world space to view space:

```
auto worldToView = params.uniforms().world_to_view();
float4 myVectorView = (myVector * worldToView);
```

Vertex-by-matrix multiplication requires vectors that are the same size as the number of columns in the matrix. This requirement means you can multiply a float4 by `world_to_view`, but not a `float3`. If you need to convert a set of Cartesian coordinates stored in a `float3`, convert it to a `float4` before multiplying, by padding the vector with an extra `1.0` value, like this:

```
// This code assumes there's an existing float3 called myVertex.
float4(myVertex.x, myVertex.y, myVertex.z, 1.0);
auto worldToView = params.uniforms().world_to_view();
float3 myVectorView = (myVector * worldToView).xyz;
```

Member Function

uniforms::view_to_projection()

Returns a matrix that transforms values from view space into projection space.

Declaration

```
metal::float4x4 view_to_projection() const thread
```

Overview

This function returns a matrix you can use to convert any view-space vector or matrix to projection space using matrix multiplication. Projection space is a flattened 2D representation of a 3D scene. RealityKit creates this by applying a perspective transform that makes entities that are farther from the camera appear smaller.

The following Metal code demonstrates how to use this matrix to convert a vector from view space to projection space:

```
auto viewToProjection = params.uniforms().view_to_projection();
float4 myVectorProjection = (myVector * viewToProjection);
```

Vertex-by-matrix multiplication requires vectors that are the same size as the number of columns in the matrix. This requirement means you can multiply a float4 by `view_to_projection`, but not a `float3`. If you need to convert a set of Cartesian coordinates stored in a `float3`, convert it to a `float4` before multiplying, by padding the vector with an extra `1.0` value, like this:

```
// This code assumes there's an existing float3 called myVertex.
float4(myVertex.x, myVertex.y, myVertex.z, 1.0);
auto viewToProjection = params.uniforms().view_to_projection();
float3 myVectorProjection = (myVector * viewToProjection).xyz;
```

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

uniforms::projection_to_view()

Returns a matrix that transforms values from projection space into view space.

Declaration

```
metal::float4x4 projection_to_view() const thread
```

Overview

This function returns a matrix that you can use to convert any projection-space vector or matrix to view space. The following Metal code demonstrates how to use this matrix to convert a vector from projection space to view space:

```
auto projectionToView = params.uniforms().projection_to_view();
float4 myVectorView = (myVector * projectionToView);
```

Vertex-by-matrix multiplication requires vectors that are the same size as the number of columns in the matrix. This requirement means you can multiply a `float4` by `projection_to_view`, but not a `float3`. If you need to convert a set of Cartesian coordinates stored in a `float3`, convert it to a `float4` before multiplying, by padding the vector with an extra `1.0` value, like this:

```
// This code assumes there's an existing float3 called myVertex.
float4(myVertex.x, myVertex.y, myVertex.z, 1.0);
auto projectionToView = params.uniforms().projection_to_view();
float3 myVectorView = (myVector * projectionToView).xyz;
```

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Struct

geometry

An object that contains per-vertex values for the current fragment.

Namespace

`realitykit::surface`

Declaration

```
struct geometry
```

Overview

This object returns per-vertex values from the entity, for example, the UV texture coordinates, vertex position, and vertex color. These functions return values that Metal has interpolated for the current fragment, based on its position relative to the three vertices that make up the fragment's triangle.

Member Functions

`float4 screen_position() const thread`

Returns the fragment's position in screen space.

`float3 world_position() const thread`

Returns the fragment's position in world space.

`float3 model_position() const thread`

Returns the fragment's position in model space.

`float4 color() const thread`

Returns the fragment's vertex color.

`float3 normal() const thread`

Returns the normal of the fragment's geometry.

`float3 tangent() const thread`

Returns the tangent of the fragment's geometry.

`float3 bitangent() const thread`

Returns the bitangent of the fragment's geometry.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

`float2 uv0() const thread`

Returns the entity's primary UV texture coordinate for the fragment.

`float2 uv1() const thread`

Returns the entity's secondary UV texture coordinate for the fragment.

`float4 custom_attribute() const thread`

Returns a user attribute set by the geometry modifier.

`float3 view_direction() const thread`

Returns a vector that points from this fragment's position to the viewer.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

geometry::screen_position()

Returns the fragment's position in screen space.

Declaration

```
float4 screen_position() const thread
```

Overview

Returns the fragment's projected coordinates in screen space. This returns the `[[position]]` input attribute from the fragment shader.

To retrieve just the two-dimensional screen position of the fragment, use the `X` and `Y` values, as the following Metal code demonstrates:

```
float2 screenCoords = params.geometry().screen_position().xy;
```

Member Function

geometry::world_position()

Returns the fragment's position in world space.

Declaration

```
float3 world_position() const thread
```

Member Function

geometry::model_position()

Returns the fragment's position in model space.

Declaration

```
float3 model_position() const thread
```

Member Function

geometry::color()

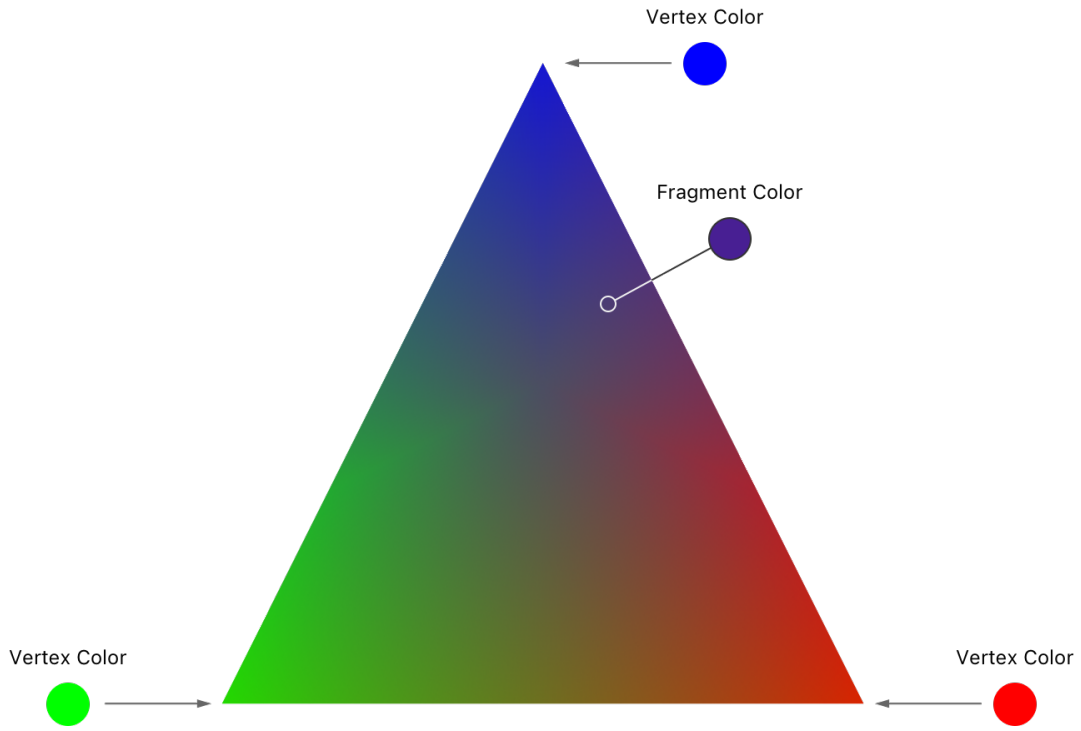
Returns the fragment's interpolated vertex color.

Declaration

```
float4 color() const thread
```

Overview

This function returns the color of the current fragment. This method interpolates its return value from the vertex colors of the three vertices that make up the fragment's triangle:



Member Function

geometry::normal()

Returns the normal of the fragment's geometry.

Declaration

```
float3 normal() const thread
```

Overview

This function returns the fragment’s normal vector. The system calculates this value by interpolating the vertex normals of the three vertices that make up the fragment's triangle, based on the fragment's position relative to those vertices.

Note: This function's return value is a calculated value the system has inferred from the vertex normals. It’s not the surface normal stored in the material's normal map.

Member Function

geometry::tangent()

Returns the tangent of the fragment's geometry.

Declaration

```
float3 tangent() const thread
```

Overview

This function returns the fragment’s tangent vector. The system calculates this value by interpolating the vertex tangents of the three vertices that make up the fragment's triangle, based on the fragment's position relative to those vertices.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

geometry::bitangent()

Returns the bitangent of the fragment's geometry.

Declaration

```
float3 bitangent() const thread
```

Overview

This function returns the fragment’s bitangent vector. The system calculates this value by interpolating the vertex bitangents of the three vertices that make up the fragment's triangle, based on the fragment's position relative to those vertices.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

geometry::uv0()

Returns the entity's primary UV texture coordinate for the fragment.

Declaration

```
float2 uv0() const thread
```

Member Function

geometry::uv1()

Returns the entity's secondary UV texture coordinate for the fragment.

Declaration

```
float2 uv1() const thread
```

Member Function

geometry::custom_attribute()

Returns a user attribute set by the geometry modifier.

Declaration

```
float4 custom_attribute() const thread
```

Overview

A geometry modifier can pass a custom per-vertex value to your surface shader by calling `params.geometry().set_custom_attribute()`. It returns a value the system interpolates from the values you set in the geometry modifier. The system calculates the value based on the current fragment's position relative to the three vertices that make up its triangle.

If your geometry modifier doesn't call `params.geometry().set_custom_attribute()`, this function returns `(0.0, 0.0, 0.0)`.

Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

geometry::view_direction()

Returns a vector that points from this fragment's position to the viewer.

Declaration

```
float3 view_direction() const thread
```

Struct

surface_properties

An object the surface shader uses to specify the rendering attributes.

Namespace

`realitykit::surface`

Declaration

```
struct surface_properties
```

Use this object to set and access the final rendering attribute values for the current fragment.

Member Functions

`half3 base_color() const thread`

Returns the base color of the fragment.

`void set_base_color(half3 value) const thread`

Sets the base color for the fragment.

`half3 emissive_color() const thread`

Returns the emissive color of the fragment.

`void set_emissive_color(half3 value) thread`

Sets the emissive color for the fragment.

`float3 normal() thread`

Returns the tangent-space normal of the fragment.

`void set_normal(float3 value) thread`

Sets the tangent-space normal of the fragment.

`half roughness() const thread`

Returns the roughness value of the fragment.

`void set_roughness(half value) thread`

Sets the roughness value for the fragment.

Copyright © 2021 Apple Inc. All Rights Reserved.

`half metallic() const thread`

Returns the reflectiveness of the fragment.

`void set_metallic(half value) thread`

Sets the reflectiveness of the fragment.

`half ambient_occlusion() const thread`

Returns the ambient occlusion value for the fragment.

`set_ambient_occlusion(half value) thread`

Sets the ambient occlusion value of the fragment.

`half specular() const thread`

Returns the specular value of the fragment.

`void set_specular(half value) thread`

Set the specular value of the fragment.

`half clearcoat() const thread`

Returns the clearcoat value for the fragment.

`void set_clearcoat(half value) thread`

Sets the clearcoat value for the fragment.

`half clearcoat_roughness() const thread`

Returns the clearcoat roughness value for the fragment.

`void set_clearcoat_roughness(half value) thread`

Sets the clearcoat roughness value for the fragment.

`half opacity() const thread`

Returns the opacity for the fragment.

`void set_opacity(half value) thread`

Sets the opacity of the fragment.

Member Function

surface_properties::base_color()

Returns the base color of the fragment.

Declaration

```
half3 base_color() const thread
```

The base color of an entity is the color of the entity before RealityKit applies any lighting or other rendering calculations. This function returns the base color value RealityKit uses to render the fragment. If you don't call `set_base_color()`, this function returns a value of `(1.0, 1.0, 1.0)`.

Member Function

surface_properties::set_base_color()

Set the base color for the fragment.

Declaration

```
void set_base_color(half3 value) const thread
```

Parameters

`value`

An RGB color as a three-component vector. All through components need to be between `0.0` and `1.0`.

Overview

The base color of an entity is the color of the entity before RealityKit applies any lighting or rendering calculations. Use this function to set the base color property for rendering. RealityKit only uses `value` if the material's lighting model is `.lit` or `.clearcoat`. If you call this function for a material set to `.unlit`, RealityKit ignores it.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

surface_properties::emissive_color()

Returns the emissive color scale of the fragment.

Declaration

```
half3 emissive_color() const thread
```

Overview

This function returns the emissive color value RealityKit uses to render this fragment. If you don't call `set_emissive_color()`, this returns a value of `(1.0, 1.0, 1.0)`.

When you use the `.lit` or `.clearcoat` lighting model and this property has a value other than `(0.0, 0.0, 0.0)`, the system gives the fragment the appearance of emitting light, such as objects with LEDs or computer screens.

When you use the `.unlit` lighting model, this property holds the final color that RealityKit renders for this fragment.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

surface_properties::set_emissive_color()

Sets the emissive color for the fragment.

Declaration

```
void set_emissive_color(half3 value) thread
```

Parameters

`value`

An RGB color as a 3-component vector. All three components need to be between `0.0` and `1.0`.

Overview

Use this function to set the emissive color property for rendering.

When you use the `.lit` or `.clearcoat` lighting model and this property has a value other than `(0.0, 0.0, 0.0)`, the fragment has the appearance of emitting light, such as objects with LEDs or computer screens.

When you specify the `.unlit` lighting model, use this function to specify the color that RealityKit uses to render the fragment.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

surface_properties::normal()

Returns the tangent-space normal of the current fragment.

Declaration

```
float3 normal() thread
```

Normal mapping is a real-time rendering technique that captures fine surface details for a model by using a texture instead of increasing the number of polygons in the model. It stores *surface normals*, which are vectors perpendicular to the surface of the model, created from a much higher-resolution version of the same 3D object. A normal map stores one normal vector per pixel by storing the vector's `X`, `Y`, and `Z` values as the `R`, `G`, and `B` components of the corresponding pixel in the UV-mapped image.

Use this function to retrieve the normal value for the current fragment. The value this function returns will have `R` and `G` channels with values between `-1.0` and `1.0` and a `B` channel with a value between `0.0` and `1.0`.

Note: A surface vector with a **Z**-value less than **`0.0`** points away from the viewer and therefore has no effect on lighting calculations, which RealityKit computes based on the viewer's location.

This function returns the normal value for the current fragment. If your surface shader doesn't call `set_normal()`, it returns a value of `(0.0, 0.0, 1.0)`.

Member Function

surface_properties::set_normal()

Sets the tangent-space normal for the current fragment.

Declaration

```
void set_normal(float3 value) thread
```

Parameters

`value`

A surface normal value color as a three-component vector.

Overview

Normal mapping is a real-time rendering technique that captures fine surface details for a model by using a texture instead of increasing the number of polygons in the model. It stores *surface normals*, which are vectors perpendicular to the surface of the model, created from a much higher-resolution version of the same 3D object. A normal map stores one normal vector per pixel by storing the vector's `X`, `Y`, and `Z` values as the `R`, `G`, and `B` components of the corresponding pixel in the UV-mapped image.

Use this function to set the normal value for the current fragment. Typically, you sample this value from a UV-mapped normal map texture. RealityKit uses the value you pass to this function in lighting calculations. This function expects normal map values where the `X` and `Y` values are between `-1.0` and `1.0` and the `Z` value is between `0.0` and `1.0`.

Note: A surface vector with a **`Z`**-value less than **`0.0`** points away from the viewer and therefore has no effect on lighting calculations, which RealityKit computes based on the viewer's location.

RealityKit uses the value passed to this function in the `.lit` and `.clearcoat` lighting models. If you call this function with a material that uses the `.unlit` lighting model, RealityKit ignores it.

When you sample values from a normal map texture, all three channels are between `0.0` and `1.0`, so you must use the `unpack_normal` function to convert the sampled value before passing it to this function, as demonstrated by the following Metal code:

```
// Retrieve the texture coordinates for this fragment.
float2 uv = params.geometry().uv0();

// Invert the y-axis for models loaded from a USDZ
// or .reality file.
```

```
uv.y = 1.0 - uv.y;

// Sample the normal map texture.
auto tex = params.textures();
half3 color = (half3)tex.normal()
              .sample(textureSampler, uv).rgb;

// Convert the normal to the correct format.
float3 normal = (float3)unpack_normal(color);

// Set the fragment's normal using the converted value.
params.surface().set_normal(normal);
```

Member Function

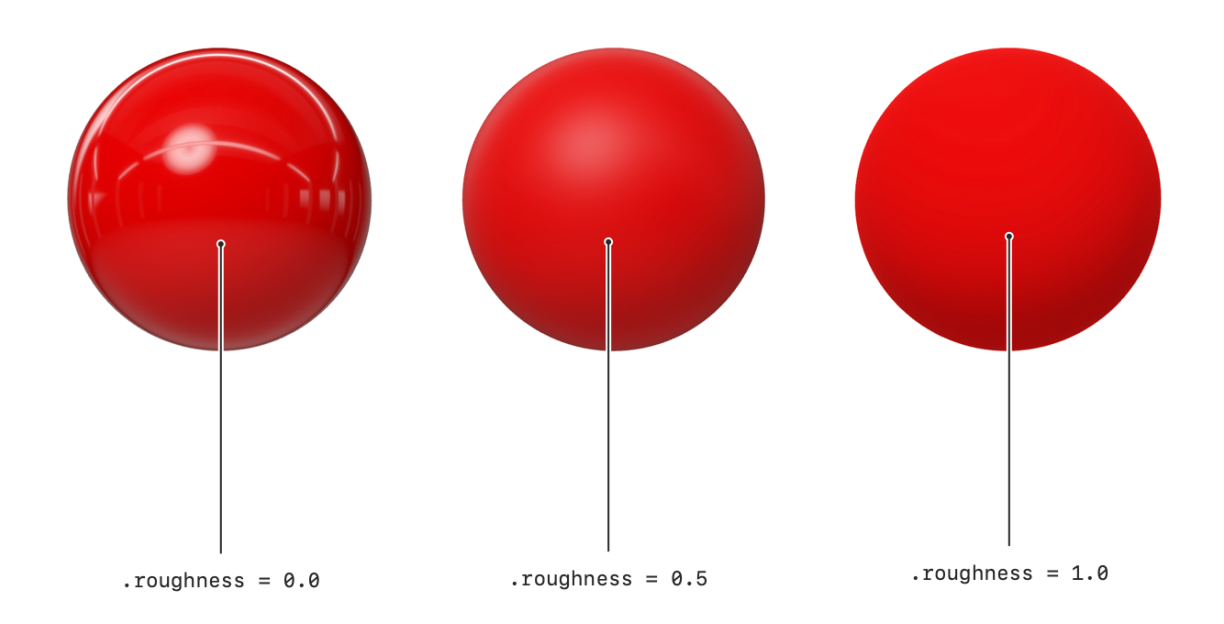
surface_properties::roughness()

Returns the roughness value of the fragment.

Declaration

```
half roughness() const thread
```

The `roughness` property represents the degree to which the surface of the entity scatters light that it reflects. A material with a `roughness` of `1.0` has a matte appearance, whereas one with a `roughness` of `0.0` has a shiny appearance, as demonstrated by the following illustration:



This function returns the roughness value RealityKit uses to render the fragment. If you don't call `set_roughness()`, this function returns a value of `0.0`.

Member Function

surface_properties::set_roughness()

Set the roughness value for the fragment.

Declaration

```
void set_roughness(half value) thread
```

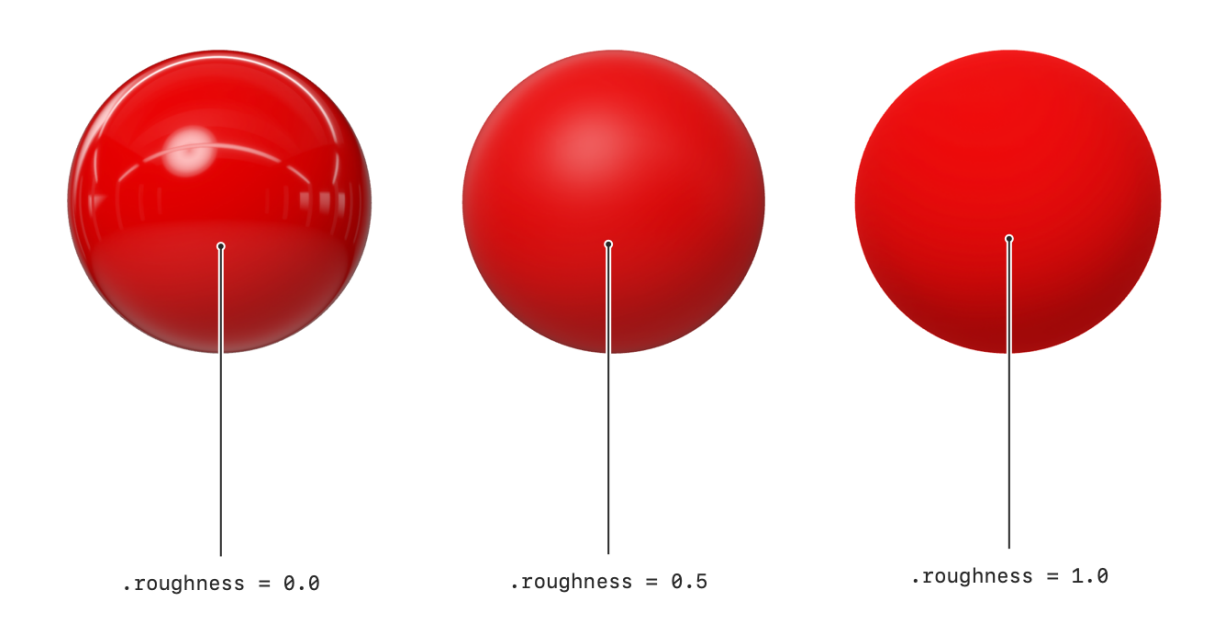
Parameters

`value`

The roughness of the fragment.

Overview

The `roughness` property represents the degree to which the surface of the entity scatters light that it reflects. A material with a `roughness` of `1.0` has a matte appearance, whereas one with a `roughness` of `0.0` has a shiny appearance, as demonstrated by the following illustration:



Use this function to set the roughness property for rendering. RealityKit uses the value passed to this function in the `.lit` and `.clearcoat` lighting models. If you call this function with a material that uses the `.unlit` lighting model, RealityKit ignores it.

Member Function

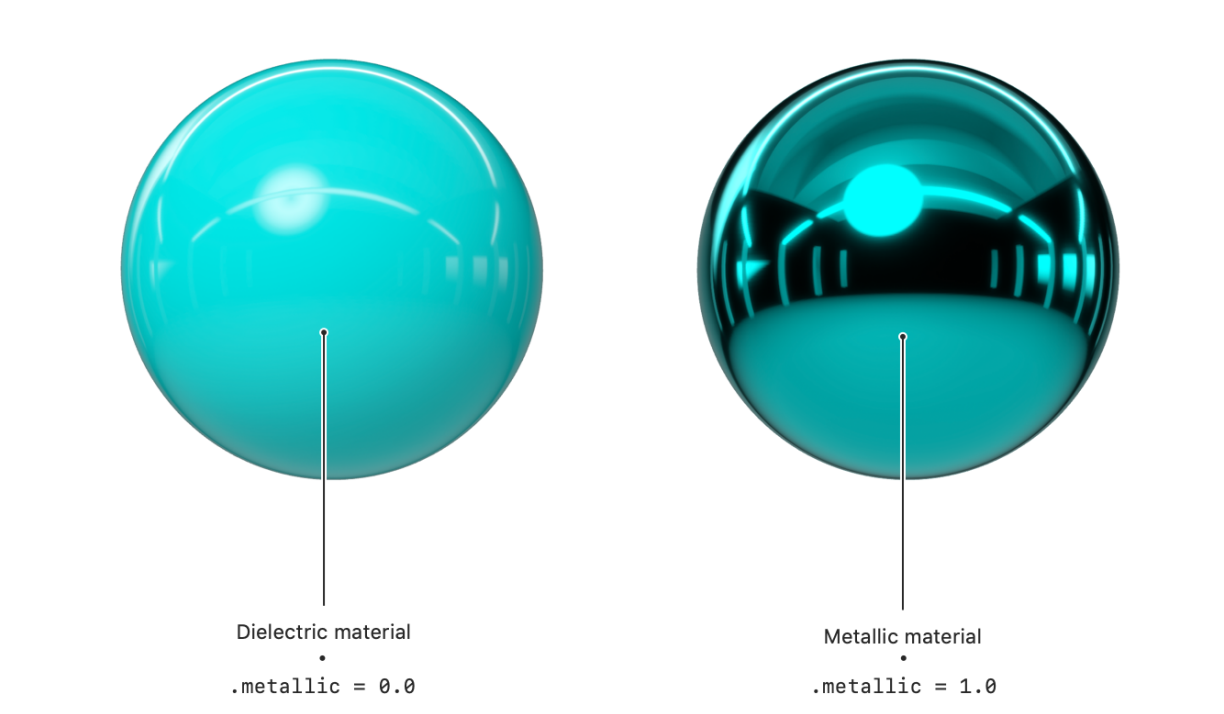
surface_properties::metallic()

Returns the reflectiveness of the fragment.

Declaration

```
half metallic() const thread
```

The `metallic` property represents the reflectiveness of the fragment. This function returns the metallic value RealityKit uses to render this fragment. A value of `1.0` represents a very reflective object, whereas a value of `0.0` represents an object that doesn't reflect the environment at all (a *dielectric* material), other than highlights from direct light sources, as the following illustration demonstrates:



If you don't call `set_metallic()`, this function returns a value of `0.0`.

Member Function

surface_properties::set_metallic()

Sets the reflectiveness of the fragment.

Declaration

```
void set_metallic(half value) thread
```

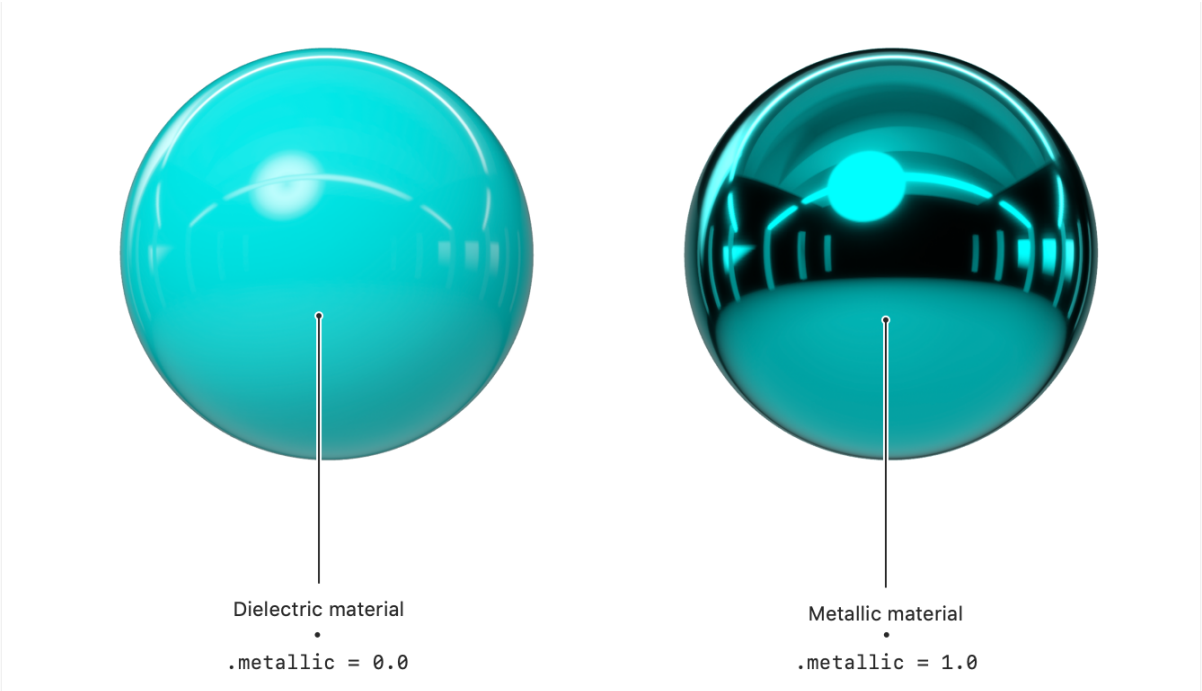
Parameters

`value`

The reflectiveness of the fragment.

Overview

The `metallic` property represents the reflectiveness of the fragment. This function returns the metallic value RealityKit uses to render this fragment. A value of `1.0` represents a very reflective object, whereas a value of `0.0` represents an object that doesn't reflect the environment at all (a *dielectric* material), other than highlights from direct light sources, as the following illustration demonstrates:



Use this function to set the metallic property for rendering. RealityKit uses the value passed to this function in the `.lit` and `.clearcoat` lighting models. If you call this function with a material that uses the `.unlit` lighting model, RealityKit ignores it.

Member Function

surface_properties::ambient_occlusion()

Returns the ambient occlusion value of the fragment.

Declaration

```
half ambient_occlusion() const thread
```

Overview

Ambient occlusion represents the entity's exposure to ambient light. A value of black (`0.0`), the darkest pixel value in a grayscale image, represents parts of the model that receive no ambient light because they may be crevices, dents, or recessed areas, or another part of the same entity prevents ambient light from reaching it. Ambient occlusion values of white (`1.0`), the lightest pixel value, represent flat portions of the model that receive full ambient light.

This function returns the ambient occlusion value that RealityKit uses to render this fragment. If you don't call `set_ambient_occlusion()`, it returns a value of `1.0`.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

surface_properties::set_ambient_occlusion()

Sets the ambient occlusion value for the fragment.

Declaration

```
set_ambient_occlusion(half value) thread
```

Parameters

`value`

The ambient occlusion value for the fragment.

Overview

Ambient occlusion represents the entity's exposure to ambient light. A value of black (`0.0`), the darkest pixel value in a grayscale image, represents parts of the model that receive no ambient light because they may be crevices, dents, or recessed areas, or another part of the same entity prevents ambient light from reaching it. Ambient occlusion values of white (`1.0`), the lightest pixel value, represent flat portions of the model that receive full ambient light.

Use this function to set the ambient occlusion property for the fragment. RealityKit uses the value passed to this function in the `.lit` and `.clearcoat` lighting models. If you call this function with a material that uses the `.unlit` lighting model, RealityKit ignores it.

Member Function

surface_properties::specular()

Returns the specular value of the fragment.

Declaration

```
half specular() const thread
```

In physically based rendering (PBR), the bright highlights from light sources reflecting off shiny or reflective objects (*specular* highlights) primarily come from the object's `roughness` value. RealityKit renders materials that have a low `roughness` value with specular highlights, based on the environment lighting and the shape of the entity. As a result, for most materials, you won't need to specify a specular value when using `PhysicallyBasedMaterial`.

For some types of *dielectric* (nonmetallic) materials, like facet-cut glass or gems, PBR algorithms don't create bright enough specular highlights if you just use `roughness`. To accurately simulate those types of materials, use this value to specify additional specular highlights.

If you don't call `set_specular()`, this function returns a value of `0.0`.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

surface_properties::set_specular()

Sets the specular value for the fragment.

Declaration

```
void set_specular(half value) thread
```

Parameters

`value`

The specular value for the fragment.

Overview

In physically based rendering (PBR), the bright highlights from light sources reflecting off shiny or reflective objects (*specular* highlights) primarily come from the object's `roughness` value. RealityKit renders materials that have a low `roughness` value with specular highlights, based on the environment lighting and the shape of the entity. As a result, for most materials, you won't need to specify a specular value when using `PhysicallyBasedMaterial`.

For some types of dielectric (nonmetallic) materials, like facet-cut glass or gems, PBR algorithms don't create bright enough specular highlights if you just use `roughness`. To accurately simulate those types of materials, use this function to specify additional specular highlights.

Use this function to set the specular property for rendering. RealityKit uses the value passed to this function in the `.lit` and `.clearcoat` lighting models. If you call this function with a material that uses the `.unlit` lighting model, RealityKit ignores it.

Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

surface_properties::clearcoat()

Returns the clearcoat value of the fragment.

Declaration

```
half clearcoat() const thread
```

An entity with a custom material renders with a clearcoat if you set the lighting model to `.clearcoat`. A clearcoat is a separate layer of transparent specular highlights that simulates a clear coating, like on a car or the surface of lacquered objects.

This function returns the clearcoat value for this fragment. If you don't call `set_clearcoat()`, this function returns `0.0`.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

surface_properties::set_clearcoat()

Sets the clearcoat value for the fragment.

Declaration

```
void set_clearcoat(half value) thread
```

Parameters

`value`

The clearcoat value for the fragment.

Overview

An entity with a custom material renders with a clearcoat if you set the lighting model to `.clearcoat`. A clearcoat is a separate layer of transparent specular highlights that simulates a clear transparent coating, like on a car or the surface of lacquered objects.

Use this function to set the clearcoat value for the current fragment. With materials that use the `.unlit` or `.lit` lighting models, calling this function has no effect.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

surface_properties::clearcoat_roughness()

Returns the clearcoat roughness value of the fragment.

Declaration

```
half clearcoat_roughness() const thread
```

An entity with a custom material renders with a clearcoat if you set the lighting model to `.clearcoat`. A clearcoat is a separate layer of transparent specular highlights that simulates a clear coating, like on a car or the surface of lacquered objects.

Clearcoat roughness controls to what degree the clearcoat scatters light that bounces off of it, which softens and spreads out the highlights. The function returns the current clearcoat roughness value for the fragment.

If the value returned by this function is greater than `0.0`, the value returned by `clearcoat()` is also greater than `0.0`, and the lighting model is set to `.clearcoat`, RealityKit renders a clearcoat for the entity as a separate layer just above the surface.

If you don't call `set_clearcoat_roughness()`, this function returns `0.0`.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

surface_properties::set_clearcoat_roughness()

Set the clearcoat roughness value for this fragment.

Declaration

```
void set_clearcoat_roughness(half value) thread
```

Parameters

`value`

A scalar value that represents the clearcoat roughness for this fragment.

Overview

An entity with a custom material renders with a clearcoat if you set the lighting model to `.clearcoat`. A clearcoat is a separate layer of transparent specular highlights that simulates a clear coating, like on a car or the surface of lacquered objects. With materials that use the `.unlit` or `.lit` lighting model, calling this function has no effect.

Use this function to set the clearcoat roughness value for the fragment. Clearcoat roughness controls how much the clearcoat scatters light that bounces off of it, which softens and spreads out the highlights. Calling this function has no effect if the fragment's `clearcoat` value is `0.0` or you don't set the material's lighting model to `.clearcoat`.

The following Metal code demonstrates how to replicate the clearcoat behavior of `PhysicallyBasedMaterial` in a surface shader:

```
// Retrieve the clearcoat scale and roughness from
// the CustomMaterial.
float clearcoatScale = params.material_constants()
                       .clearcoat_scale();
float clearcoatRoughnessScale = params.material_constants()
                                .clearcoat_roughness_scale();

// Retrieve the entity's texture coordinates.
float2 uv = params.geometry().uv0();

// Entities you load from a USDZ or .reality file use texture
// coordinates with a flipped y-axis. This compensates for that.
uv.y = 1.0 - uv.y;

// Sample a value from the clearcoat and clearcoat roughness
// textures.
```

```
auto tex = params.textures();
half clearcoat = tex.clearcoat().sample(textureSampler, uv).r;
half clearcoatRoughess = tex.clearcoat_roughness()
                         .sample(textureSampler, uv).r;

// Multiply the sampled clearcoat value by the scale, and
// assign it.
clearcoat *= clearcoatScale;
params.surface().set_clearcoat(clearcoat);

// Multiply the scale and sampled texture value from the
// clearcoat roughness texture, and assign it.
clearcoatRoughess *= clearcoatRoughnessScale;
params.surface().set_clearcoat_roughness(clearcoatRoughess);
```

Member Function

surface_properties::opacity()

Returns the opacity value of the fragment.

Declaration

```
half opacity() const thread
```

This function returns the opacity value for the current fragment, which controls whether the entity is transparent or translucent. A value of `1.0` represents a completely opaque fragment, where the objects behind the entity (from the perspective of the camera) don't show through. A value of `0.0` represents a completely transparent fragment, which means the objects behind this entity are visible through it. Other values represent partially transparent fragments.

If you don't call `set_opacity()`, this function returns `1.0`.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

surface_properties::set_opacity()

Sets the opacity value for the fragment.

Declaration

```
void set_opacity(half value) thread
```

Parameters

`value`

A scalar value that represents the opacity for this fragment.

Overview

Use this function to set the opacity for the current fragment. Setting this value to `0.0` causes this fragment to be completely transparent. Setting it to `1.0` causes this fragment to be completely opaque. Values between `0.0` and `1.0` cause the fragment to be translucent, with objects behind the entity showing through with less intensity as the value approaches `1.0`.

To replicate the behavior of `PhysicallyBasedMaterial` when calling this function, retrieve both the opacity `scale` and the `opacityThreshold` from the material, and sample the opacity `texture`. If the `opacityThreshold` is greater than `0.0`, compare the sampled value to the threshold, and set opacity to either `1.0`, if the value is greater than the threshold, or `0.0` otherwise. If `opacityThreshold` is equal to `0.0`, multiply the opacity scale and the sampled value together to get the final opacity value. The following code demonstrates this process:

```
// Retrieve the opacity scale from the CustomMaterial.
float opacityScale = params.material_constants().opacity_scale();
float opacityThreshold =
params.material_constants().opacity_threshold();

// Retrieve the entity's texture coordinates.
float2 uv = params.geometry().uv0();

// Entities you load from a USDZ or .reality file use texture
// coordinates with a flipped y-axis. This compensates for that.
uv.y = 1.0 - uv.y;

auto tex = params.textures();
half opacity = tex.opacity().sample(textureSampler, uv).r;

if (opacityThreshold > 0.0) {
```

```
    // If the opacity threshold is greater than 0, use masking
    // behavior and set the opacity to either 1.0 or 0.0, depending
    // on the value of the opacity threshold. Ignore opacity scale
    // when using a mask.

    if (opacity > opacityThreshold) {
        params.surface().set_opacity(1.0);
    } else {
        // Setting the opacity to 0.0 using PBR rendering (.lit
        // or .clearcoat) results in a transparent glass-like
        // object. That means that RealityKit might render some
        // value for this fragment due to specular highlights or
        // clearcoat. For masking behavior, completely discard the
        // transparent fragment.
        discard_fragment();
    }
} else {
    // If the opacity threshold is 0, then multiply opacity by
    // scale.
    opacity *= opacityScale;
}
params.surface().set_opacity(opacity);
```

Geometry Modifier APIs

A geometry modifier is an optional Metal shader function for custom materials. Geometry modifiers run during vertex shader execution, which means they execute once every frame for every vertex in the entity being rendered.

A geometry modifier can change the position of any vertex in the entity by setting an offset value for it. Offsetting vertices allows you to change the size or shape of your entity before RealityKit renders it.

Geometry modifiers can also set or calculate new per-vertex values, such as new vertex colors or vertex normals. RealityKit uses these modified values to render the entity. The modified values are also available in your surface shader, where Metal automatically interpolates the value for each fragment based on its relative position to the vertices that make up its triangle.

Changes to vertex data that you make in a geometry modifier are transient and don’t affect physics calculations, collisions, or the size and shape of the entity in your RealityKit scene. Once RealityKit renders the current frame, the offsets are discarded.

For more information on using shader functions with custom materials, see Taking Control of RealityKit Rendering Using Custom Materials.

Struct

geometry_parameters

An object the framework uses to pass data into a geometry modifier function.

Namespace

`realitykit`

Declaration

```
struct geometry_parameters
```

Overview

This struct holds all the information that RealityKit passes into a geometry modifier. The available data is grouped into sub-objects of related types. For example, you can call `textures()` to access textures from your custom material and `material_parameters()` to access the material's non-texture values, like the the base color tint or roughness scale.

Geometry modifier functions are called by RealityKit during vertex shader execution, which means that a geometry modifier function will be called once for every vertex in the entity.

A geometry modifier is optional and can be used to offset the position of an entity's vertices. To offset the position of the current entity, call `set_world_position_offset()` or `set_model_position_offset()` on the object returned by the `geometry()` function.

A geometry modifier can also set or calculate new per-vertex values, such as vertex color or vertex normal. RealityKit uses the modified values to render the entity. RealityKit also makes modified values available in the surface shader. Metal automatically interpolates the values based on the fragment's relative position to the three vertices that make up its triangle.

Changes you make in the geometry modifier only affect rendering and don't affect physics calculations or collisions in the RealityKit scene.

Note: If you offset vertices in a way that changes the shape of your entity, you may have to recalculate and set that vertex's normal, tangent, and bitangent values to ensure that lighting and other calculations have access to the correct value for the vertex's new position.

Member Functions

`geometry_modifier::uniforms uniforms() const thread`

Retrieves an object that holds entity-specific and global constants.

`texture::textures textures() const thread`

Retrieves a data object that holds textures from the custom material.

`geometry_modifier::geometry geometry() thread`

Retrieves an object that holds geometry information about the current vertex.

`material::material_parameters material_constants() const thread`

An object that holds entity-specific and global constants.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

geometry_parameters::uniforms()

Retrieves an object that holds entity-specific and global constants.

Declaration

```
geometry_modifier::uniforms uniforms() const thread
```

Overview

This function returns a read-only object that provides access to:

- Entity-specific values, such as the entity's texture coordinates
- Entity-specific utility functions, such as ones that let you convert values between model space and world space
- Global values, such as the current elapsed time

Member Function

geometry_parameters::textures()

Retrieves a data object that holds textures from the custom material.

Declaration

```
texture::textures textures() const thread
```

Overview

This function returns a read-only object that provides access to the textures set on the shader's custom material.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

geometry_parameters::geometry()

Retrieves an object that holds geometry information about the current vertex.

Declaration

```
geometry_modifier::geometry geometry() thread
```

Overview

This function returns an object that provides access to the geometry data for the current vertex and allows modifying per-vertex values, like vertex color and vertex normal.

Member Function

geometry_parameters::material_constants()

An object that holds entity-specific and global constants.

Declaration

```
material::material_parameters material_constants() const thread
```

Overview

This function returns a read-only object that holds entity-specific constants from the entity's material.

Struct

uniforms

An object used to pass data into a geometry modifier function.

Namespace

`realitykit::geometry_modifier`

Declaration

```
struct uniforms
```

Overview

This object holds values that are constant across all vertices and all fragments. It also returns matrices for converting vectors and other matrices between different coordinate spaces.

Member Functions

`float time() const thread`

Returns the elapsed time in seconds.

`float4 custom_parameter() const thread`

Returns the custom vector from the entity's material.

`metal::float4x4 model_to_world() const thread`

Returns a matrix that transforms values from model space into world space.

`metal::float4x4 model_to_view() const thread`

Returns a matrix that transforms values from model space into view space.

`metal::float4x4 world_to_view() const thread`

Returns a matrix that transforms values from world space into view space.

`metal::float4x4 view_to_projection() const thread`

Returns a matrix that transforms values from view space into projection space.

`metal::float4x4 projection_to_view() const thread`

Returns a matrix that transforms values from projection space into view space.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

uniforms::time()

Returns the elapsed time in seconds.

Declaration

```
float time() const thread
```

Overview

This function returns the number of seconds that have elapsed since RealityKit began rendering the current scene.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

uniforms::custom_parameter()

Returns the custom vector from the entity's material.

Declaration

```
float4 custom_parameter() const thread
```

Overview

This function returns `value` from the `custom` property of the entity's material. The following Swift code demonstrates how to set this value on a custom material with a single `SIMD4<Float>` vector:

```
customMaterial.custom.value = SIMD4<Float>(x: 0.25,
                                           y: 0.25,
                                           z: 0.25,
                                           w: 1.0)
```

The following Metal code demonstrates how to retrieve that vector value in your surface shader function:

```
float4 customVector = params.uniforms().custom_parameter();
```

You can also use the `custom` parameter to pass up to four individual scalar values instead of a vector value. The following code demonstrates how to set individual values on a custom material in Swift:

```
customMaterial.custom.value[0] = 0.25
customMaterial.custom.value[1] = 0.75
```

And the following Metal code demonstrates how to retrieve those individual scalar values in a surface shader function:

```
float value = params.uniforms().custom_parameter()[0];
float otherValue = params.uniforms().custom_parameter()[1];
```

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

uniforms::model_to_world()

Returns a matrix that transforms values from model space into world space.

Declaration

```
metal::float4x4 model_to_world() const thread
```

Overview

This function returns a matrix you can use to convert any model-space vector or matrix to world space using matrix multiplication. The following Metal code demonstrates how to use this matrix to convert a vector from model space to world space:

```
auto modelToWorld = params.uniforms().model_to_world();
float4 myVectorWorld = (myVector * modelToWorld);
```

Vertex-by-matrix multiplication requires vectors that are the same size as the number of columns in the matrix. This requirement means you can multiply a `float4` by `model_to_world`, but not a `float3`. If you need to convert a set of Cartesian coordinates stored in a `float3`, convert it to a `float4` before multiplying, by padding the vector with an extra `1.0` value, like this:

```
// This code assumes there's an existing float3 called myVertex.
float4(myVertex.x, myVertex.y, myVertex.z, 1.0);
auto modelToWorld = params.uniforms().model_to_world();
float3 myVectorWorld = (myVector * worldToView).xyz;
```

Member Function

uniforms::model_to_view()

Returns a matrix that transforms values from model space into view space.

Declaration

```
metal::float4x4 model_to_view() const thread
```

Overview

This function returns a matrix you can use to convert any model-space vector or matrix to view space using matrix multiplication. The following Metal code demonstrates how to use this matrix to convert a vector from model space to view space:

```
auto modelToView = params.uniforms().model_to_view();
float4 myVectorView = (myVector * modelToView);
```

Vertex-by-matrix multiplication requires vectors that are the same size as the number of columns in the matrix. This requirement means you can multiply a `float4` by `model_to_view`, but not a `float3`. If you need to convert a set of Cartesian coordinates stored in a `float3`, convert it to a `float4` before multiplying, by padding the vector with an extra `1.0` value, like this:

```
// This code assumes there's an existing float3 called myVertex.
float4(myVertex.x, myVertex.y, myVertex.z, 1.0);
auto modelToView = params.uniforms().model_to_view();
float3 myVectorView = (myVector * modelToView).xyz;
```

Member Function

uniforms::world_to_view()

Returns a matrix that transforms values from world space into view space.

Declaration

```
metal::float4x4 world_to_view() const thread
```

This function returns a matrix you can use to convert any world-space vector or matrix to view space using matrix multiplication. The following Metal code demonstrates how to use this matrix to convert a vector from world space to view space:

```
auto worldToView = params.uniforms().world_to_view();
float4 myVectorView = (myVector * worldToView);
```

Vertex-by-matrix multiplication requires vectors that are the same size as the number of columns in the matrix. This requirement means you can multiply a `float4` by `world_to_view`, but not a `float3`. If you need to convert a set of Cartesian coordinates stored in a `float3`, convert it to a `float4` before multiplying, by padding the vector with an extra `1.0` value, like this:

```
// This code assumes there's an existing float3 called myVertex.
float4(myVertex.x, myVertex.y, myVertex.z, 1.0);
auto worldToView = params.uniforms().world_to_view();
float3 myVectorView = (myVector * worldToView).xyz;
```

Member Function

uniforms::view_to_projection()

Returns a matrix that transforms values from view space into projection space.

Declaration

```
metal::float4x4 view_to_projection() const thread
```

Overview

This function returns a matrix you can use to convert any view-space vector or matrix to projection space using matrix multiplication. Projection space flattens the 3D scene into a 2D space by applying a perspective transform that makes entities that are farther away from the camera appear smaller.

The following Metal code demonstrates how to use this matrix to convert a vector from view space to projection space:

```
auto viewToProjection = params.uniforms().view_to_projection();
float4 myVectorProjection = (myVector * viewToProjection);
```

Vertex-by-matrix multiplication requires vectors that are the same size as the number of columns in the matrix. This requirement means you can multiply a `float4` by `view_to_projection`, but not a `float3`. If you need to convert a set of Cartesian coordinates stored in a `float3`, convert it to a `float4` before multiplying, by padding the vector with an extra `1.0` value, like this:

```
// This code assumes there's an existing float3 called myVertex.
float4(myVertex.x, myVertex.y, myVertex.z, 1.0);
auto viewToProjection = params.uniforms().view_to_projection();
float3 myVectorProjection = (myVector * viewToProjection).xyz;
```

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

uniforms::projection_to_view()

Returns a matrix that transforms values from projection space into view space.

Declaration

```
metal::float4x4 projection_to_view() const thread
```

Overview

This function returns a matrix that you can use to convert any projection-space vector or matrix to view space. The following Metal code demonstrates how to use this matrix to convert a vector from projection space to view space:

```
auto projectionToView = params.uniforms().projection_to_view();
float4 myVectorView = (myVector * projectionToView);
```

Vertex-by-matrix multiplication requires vectors that are the same size as the number of columns in the matrix. This requirement means you can multiply a `float4` by `projection_to_view`, but not a `float3`. If you need to convert a set of Cartesian coordinates stored in a `float3`, convert it to a `float4` before multiplying, by padding the vector with an extra `1.0` value, like this:

```
// This code assumes there's an existing float3 called myVertex.
float4(myVertex.x, myVertex.y, myVertex.z, 1.0);
auto projectionToView = params.uniforms().projection_to_view();
float3 myVectorView = (myVector * projectionToView).xyz;
```

Copyright © 2021 Apple Inc. All Rights Reserved.

Struct

geometry

An object that contains geometry properties for the current vertex.

Namespace

`realitykit::geometry_modifier`

Declaration

```
struct geometry
```

Overview

This object returns values relating to the current entity's mesh resource, such as the UV texture coordinates. Geometry data is derived from the mesh data of the shader's entity. The values returned from this object's functions contain per-vertex data that RealityKit has interpolated for the current fragment.

Member Functions

`uint vertex_id() const thread`

Returns the Metal vertex ID for the current vertex.

`float3 model_position() const thread`

Returns the position of the current vertex in model space.

`float3 world_position() const thread`

Returns the position of the current vertex in world space.

`float3 model_position_offset() const thread`

Returns the offset value for the current vertex in model space.

`void set_model_position_offset(float3 value) thread`

Sets the offset value for the current vertex in model space.

`float3 world_position_offset() const thread`

Returns the offset value for the current vertex in world space.

`void set_world_position_offset(float3 value) thread`

Sets the offset value for the current vertex in world space.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

`float4 color() const thread`

Returns the color of the current vertex.

`void set_color(float4 value) thread`

Sets the color for the current vertex.

`float3 normal() const thread`

Returns the normal value of the current vertex.

`set_normal(float3 value) thread`

Sets the normal value for the current vertex.

`float3 tangent() const thread`

Returns the tangent value for the current vertex.

`void set_tangent(float3 value) thread`

Sets the tangent value for the current vertex.

`float3 bitangent() const thread`

Returns the bitangent value for the current vertex.

`void set_bitangent(float3 value) thread`

Sets the bitangent value for the current vertex.

`float2 uv0() const thread`

Returns the primary UV texture coordinates of the current vertex.

`void set_uv0(float2 value) thread`

Sets the primary UV texture coordinates for the current vertex.

`float2 uv1() const thread`

Returns the secondary UV texture coordinates of the current vertex.

`void set_uv1(float2 value) thread`

Sets the secondary UV texture coordinates for the current vertex.

Member Function

geometry::vertex_id()

Returns the Metal vertex identifier for the current vertex.

Declaration

```
uint vertex_id() const thread
```

Member Function

geometry::model_position()

Returns the position of the current vertex in model space.

Declaration

```
float3 model_position() const thread
```

Member Function

geometry::world_position()

Returns the position of the current vertex in world space.

Declaration

```
float3 world_position() const thread
```

Member Function

geometry::model_position_offset()

Returns the offset for the current vertex in model space.

Declaration

```
float3 model_position_offset() const thread
```

Overview

This value defaults to `(0.0, 0.0, 0.0)`.

Member Function

geometry::set_model_position_offset()

Sets the offset for the current vertex in model space.

Declaration

```
void set_model_position_offset(float3 value) thread
```

Parameters

`value`

A vector representing the offset value for this vertex as model-space coordinates.

Overview

Before RealityKit renders the entity, it adds this value to the vertex position. Offset values are transient and don't persist from frame to frame.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

geometry::world_position_offset()

Returns the offset for the current vertex in world space.

Declaration

```
float3 world_position_offset() const thread
```

Overview

This value defaults to `(0.0, 0.0, 0.0)`.

Member Function

geometry::set_world_position_offset()

Sets the offset for the current vertex in world space.

Declaration

```
void set_model_position_offset(float3 value) thread
```

Parameters

`value`

A vector representing the offset value for this vertex as world-space coordinates.

Overview

Before RealityKit renders the entity, it adds `value` to the vertex position. Offset values are transient and don't persist from frame to frame.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

geometry::color()

Returns the color of the current vertex.

Declaration

```
float4 color() const thread
```

Member Function

geometry::set_color()

Sets the color for the current vertex.

Declaration

```
void set_color(float4 value) thread
```

Parameters

`value`

A vector holding an RGB value representing the new color for this vertex.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

geometry::normal()

Returns the normal vector for the current vertex.

Declaration

```
float3 normal() const thread
```

Member Function

geometry::set_normal()

Sets the normal value for the current vertex.

Declaration

```
void set_normal(float3 value) thread
```

Parameters

`value`

A vertex normal vector.

Overview

If your geometry modifier offsets vertices in a way that changes the shape of your entity, you may need to recalculate the normal to ensure that later calculations in the fragment shader and surface shader are using correct values.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

geometry::tangent()

Returns the tangent vector for the current vertex.

Declaration

```
float3 tangent() const thread
```

Member Function

geometry::set_tangent()

Sets the tangent value for the current vertex.

Declaration

```
void set_tangent(float3 value) thread
```

Parameters

`value`

A vertex tangent vector.

Overview

If your geometry modifier offsets vertices in a way that changes the shape of your entity, you may need to recalculate the tangent to ensure that later calculations in the fragment shader and surface shader are using correct values.

Member Function

geometry::bitangent()

Returns the bitangent vector for the current vertex.

Declaration

```
float3 bitangent() const thread
```

Member Function

geometry::set_bitangent()

Sets the bitangent value for the current vertex.

Declaration

```
void set_bitangent(float3 value) thread
```

Parameters

`value`

A vertex bitangent vector.

Overview

If your geometry modifier offsets vertices in a way that changes the shape of your entity, you may need to recalculate the bitangent to ensure that later calculations in the fragment shader and surface shader are using correct values.

Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

geometry::uv0()

Returns the primary UV texture coordinates of the current vertex.

Declaration

```
float2 uv0() const thread
```

Overview

This function returns the UV texture coordinates for the current vertex from the entity's primary texture coordinates.

Member Function

geometry::set_uv0()

Sets the primary UV texture coordinates for the current vertex.

Declaration

```
void set_uv0(float2 value) thread
```

Parameters

`value`

The new UV coordinates.

Overview

Changes made by calling this function are transient and aren't applied to the entity in your RealityKit scene.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

geometry::uv1()

Returns the secondary UV texture coordinates of the current vertex.

Declaration

```
float2 uv1() const thread
```

Member Function

geometry::set_uv1()

Sets the secondary UV texture coordinates for the current vertex.

Declaration

```
void set_uv1(float2 value) thread
```

Use this function to change the secondary UV mapping for the current entity. Changes made by calling this function are transient and aren't applied to the entity in your RealityKit scene.

Copyright © 2021 Apple Inc. All Rights Reserved.

Shared APIs

Surface shaders and geometry modifiers use distinct APIs with unique namespaces. There are some APIs, however, that are available to both surface shaders and geometry modifiers. These shared APIs include functions for retrieving textures specified on the custom material, like `baseColor.texture` and `roughness.texture`, as well as methods for retrieving non-texture properties from the material, such as `baseColor.tint`.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Struct

textures

An object the framework uses to pass textures from the custom material to a shader function.

Namespace

`realitykit::texture`

Declaration

```
struct textures
```

Overview

This object provides access to all of the material's textures from a shader function's `CustomMaterial`, regardless of the material's lighting modes, even if the modes don't support the corresponding output. For example, you can use the `clearcoat` texture as an input even if your material uses the `.unlit` lighting model, which doesn't support clearcoat rendering.

Both `base_color()` and `emissive_color()` support sRGB textures and use an embedded color-space profile if the original image file includes one. All other functions return raw values from the original image and ignore color-space information.

Because shader functions fire once for each vertex or fragment, your shader function needs to sample textures to to get the correct value for the current vertex or fragment. To do that, declare a sampler object that specifies how you wish to sample the texture, including how it's resized if needed and how UV texture coordinates outside of the normal range are handled. Here's an example sampler declaration:

```
constexpr sampler textureSampler(coord::normalized,
                                 address::repeat,
                                 filter::linear,
                                 mip_filter::linear);
```

You can use the same sampler multiple times on multiple textures, so if you wish to sample two textures the same way, use the same sampler for both. For RealityKit custom shader functions, you must declare no more than eight samplers.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

For more information on samplers, see the Metal Shading Language Specification.

To use a sampler, call the `sample()` function on a returned texture object, supplying the sampler you declared earlier and the UV coordinate value for the current vertex or fragment, as the following code demonstrates:

```
float2 uv = params.geometry().uv0();
uv.y = 1.0 - uv.y; // Flip the coordinates for loaded models.
auto tex = params.textures();
half3 color = (half3)tex.base_color()
              .sample(textureSampler, uv).rgb;
```

Member Functions

`metal::texture2d<half> base_color() const thread`

Returns the base color texture from the custom material.

`metal::texture2d<half> emissive_color() const thread`

Returns the emissive color texture from the custom material.

`metal::texture2d<half> normal() const thread`

Returns the normal map texture from the custom material.

`metal::texture2d<half> roughness() const thread`

Returns the roughness texture from the custom material.

`metal::texture2d<half> metallic() const thread`

Returns the metallic texture from the custom material.

`metal::texture2d<half> ambient_occlusion() const thread`

Returns the ambient occlusion texture from the custom material.

`metal::texture2d<half> specular() const thread`

Returns the specular texture from the custom material.

`metal::texture2d<half> opacity() const thread`

Returns the opacity texture from the custom material.

`metal::texture2d<half> clearcoat() const thread`

Returns the clearcoat texture from the custom material.

`metal::texture2d<half> clearcoat_roughness() const thread`

Returns the clearcoat roughness texture from the custom material.

`metal::texture2d<half> custom() const thread`

Returns the texture from the custom property of the custom material.

Member Function

textures::base_color()

Returns the base color texture from the custom material.

Declaration

```
metal::texture2d<half> base_color() const thread
```

Overview

This function returns the texture from the `baseColor` property of the shader's material. The base color texture supports sRGB images and applies the embedded color-space adjustment to the texture before sending it to the GPU. As a result, the sampled value from this texture is already adjusted for the rendering device based on the color-space information in the original file.

Although this property primarily exists so shader functions can calculate the value to pass to `set_base_color()`, what you actually use it for in your shader functions is completely up to you. If, for example, you don't need a base color for your entity, but you need two textures to define the entity's emissive color, you can use base color to submit the second texture that you need to calculate the value to pass to `set_base_color()`.

Note: Although you can use the **`base_color()`** texture for any purpose, avoid using it as a non-color input because the sampled values may be significantly different than the raw value in your image file.

To replicate the behavior of `PhysicallyBasedMaterial` in your surface shader, retrieve the base color `tint` and sample the base color `texture`, then multiply them together, as the following Metal code demonstrates:

```
// Retrieve the base color tint from the CustomMaterial.
half3 baseColorTint = (half3)params.material_constants()
                      .base_color_tint();

// Retrieve the primary texture coordinates.
float2 uv = params.geometry().uv0();

// Flip the UV coordinate's y-axis for models loaded
// from a USDZ or .reality file.
uv.y = 1.0 - uv.y;

auto tex = params.textures();
half3 color = (half3)tex.base_color()
```

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

```
                .sample(textureSampler, uv).rgb;

    // Multiply the tint and the sampled value from the texture,
    // and assign the result to the shader's base color property.
    color *= baseColorTint;
    params.surface().set_base_color(color);
```

Member Function

textures::emissive_color()

Returns the emissive color texture from the custom material.

Declaration

```
metal::texture2d<half> emissive_color() const thread
```

Overview

This function returns the texture from the `emissiveColor` property of the shader's material. The base color texture supports sRGB images and applies the embedded color-space adjustment to the texture before sending it to the GPU. As a result, the sampled value from this texture is already adjusted for the rendering device based on the color-space information in the original file.

Although this property primarily exists so shader functions can calculate the value to pass to `set_base_color()`, what you actually use it for in your shader functions is completely up to you. If, for example, you don't need a base color for your entity, but you need two textures to define the entity's emissive color, you can use base color to submit the second texture that you need to calculate the value to pass to `set_base_color()`.

Note: Although you can use the **`emissive_color()`** texture for any purpose, avoid using it as a non-color input because the sampled values may be significantly different than the raw value in your image file.

To replicate the behavior of `PhysicallyBasedMaterial` in your surface shader, retrieve both the emissive color `tint` and sample the emissive color `texture`, then multiply them together, as the following code demonstrates:

```
// Retrieve the emissive color tint from the CustomMaterial.
half3 emissiveColorTint = (half3)params.material_constants()
                          .emissive_color_tint();

// Retrieve the primary texture coordinates.
float2 uv = params.geometry().uv0();

// Flip the UV coordinate's y-axis for models loaded from
// a USDZ or .reality file.
uv.y = 1.0 - uv.y;

auto tex = params.textures();
half3 color = (half3)tex.emissive_color()
```

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

```
                .sample(textureSampler, uv).rgb;

    // Multiply the tint and the sampled value from the texture,
    // and assign the result to the shader's emissive color
    // property.
    color *= emissiveColorTint;
    params.surface().set_emissive_color(color);
```

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

textures::normal()

Returns the normal map texture from the custom material.

Declaration

```
metal::texture2d<half> normal() const thread
```

Overview

This function provides access to the `texture` from the `normal` property of the shader's custom material. This property's primary purpose is to enable shader functions to calculate the value to pass to `set_normal()`, but you can use it to calculate other values if your entity doesn't use a normal map. Sampled texture values are between `0.0` and `1.0`.

When sampling any texture property as a normal map, convert the value sampled from the texture before passing it to `surface_properties.set_normal()`, which expects the `R` and `G` values to be between `-1.0` and `1.0`. and the `B` value to be between `0.0` and `1.0` The range of the `blue` is different because it represents the z-axis of the normal vector. A value of less than `0.0` indicates a vector that points away from the camera, which has no impact on lighting calculations.

The following code demonstrates how to use a standard tangent-space normal map to set the normal in a surface shader:

```
// Retrieve the primary texture coordinates.
float2 uv = params.geometry().uv0();

// Flip the UV coordinate's y-axis for models loaded from
// a USDZ or .reality file.
uv.y = 1.0 - uv.y;

// Sample the value from the normal map.
auto tex = params.textures();
half3 color = (half3)tex.normal()
              .sample(textureSampler, uv).rgb;

// Convert the sample value to a surface normal vector.
float3 normal = (float3)unpack_normal(color);
params.surface().set_normal(normal);
```

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

textures::roughness()

Returns the roughness texture from the custom material.

Declaration

```
metal::texture2d<half> roughness() const thread
```

Overview

This function returns the texture from the `roughness` property of the shader's material. The roughness texture doesn't support sRGB images and discards color-space data embedded in the source image.

This property's primary purpose is to enable shader functions to calculate the value to pass to `set_roughness()`, but what you actually use it for in your shader function is completely up to you. If, for example, you don't need a roughness map for your entity, but you need two textures to define another non-color attribute, like ambient occlusion, you can use `roughness` to submit the second texture.

To replicate the behavior of `PhysicallyBasedMaterial` in your surface shader, retrieve both the roughness `scale` and sample the roughness `texture`, then multiply them together, as the following code demonstrates:

```
// Retrieve the emissive color tint from the CustomMaterial.
float roughnessScale = (half3)params.material_constants()
                       .roughness_scale();

// Retrieve the primary texture coordinates
float2 uv = params.geometry().uv0();

// Flip the UV coordinate's y-axis for models loaded from
// a USDZ or .reality file.
uv.y = 1.0 - uv.y;

auto tex = params.textures();
half roughness = (half3)tex.roughness()
                 .sample(textureSampler, uv).r;

// Multiply the scale and the sampled value from the texture,
// and assign the result to the shader's roughness property.
roughness *= roughnessScale;
params.surface().set_roughness(roughness);
```

Member Function

textures::metallic()

Returns the metallic texture from the custom material.

Declaration

```
metal::texture2d<half> metallic() const thread
```

Overview

This function returns the texture from the `metallic` property of the shader's material. The metallic texture doesn't support sRGB images and discards color-space data embedded in the source image.

This property's primary purpose is to enable shader functions to calculate the value to pass to `set_metallic()`, but what you actually use it for in your shader function is completely up to you. If, for example, you don't need a metallic map for your entity, but you need two textures to define another non-color attribute, like roughness, you can use the `metallic` property to submit the second texture.

To replicate the behavior of `PhysicallyBasedMaterial` in your surface shader, retrieve both the metallic `scale` and sample the metallic `texture`, then multiply them together, as the following code demonstrates:

```
// Retrieve the emissive color tint from the CustomMaterial.
float metallicScale = (half3)params.material_constants()
                      .metallic_scale();

// Retrieve the primary texture coordinates.
float2 uv = params.geometry().uv0();

// Flip the UV coordinate's y-axis for models loaded
// from a USDZ or .reality file.
uv.y = 1.0 - uv.y;

auto tex = params.textures();
half color = (half3)tex.metallic()
              .sample(textureSampler, uv).r;

// Multiply the scale and the sampled value from the texture,
// and assign the result to the shader's base color property.
color *= metallicScale;
params.surface().set_metallic(color);
```

Member Function

textures::ambient_occlusion()

Returns the ambient occlusion texture from the custom material.

Declaration

```
metal::texture2d<half> ambient_occlusion() const thread
```

Overview

This function returns the texture from the `ambientOcclusion` property of the shader's material. The ambient occlusion texture doesn't support sRGB images and discards color-space data embedded in the source image.

This property's primary purpose is to enable shader functions to calculate the value to pass to `set_ambient_occlusion()`, but what you actually use it for in your shader function is completely up to you. If, for example, you don't need an ambient occlusion map for your entity, but you need two textures to define another non-color attribute, like roughness, you can use the `ambientOcclusion` property on `CustomMaterial` to submit the second texture.

To replicate the behavior of `PhysicallyBasedMaterial` in your surface shader, sample the ambient occlusion `texture`, and pass it to `set_ambient_occlusion()`, as the following code demonstrates:

```
// Retrieve the primary texture coordinates.
float2 uv = params.geometry().uv0();

// Flip the UV coordinate's y-axis for models loaded
// from a USDZ or .reality file.
uv.y = 1.0 - uv.y;

auto tex = params.textures();
half ao = tex.ambient_occlusion()
          .sample(textureSampler, uv).r;

params.surface().set_ambient_occlusion(ao);
```

Member Function

textures::specular

Returns the specular texture from the custom material.

Declaration

```
metal::texture2d<half> specular() const thread
```

Overview

This function returns the texture from the `specular` property of the shader's material. The specular texture doesn't support sRGB images and discards color-space data embedded in the source image.

This property's primary purpose is to enable shader functions to calculate the value to pass to `set_specular()`, but what you actually use it for in your shader function is completely up to you. If, for example, you don't need a metallic map for your entity, but you need two textures to define another non-color attribute, like roughness, you can use the `specular` property to submit the second texture.

To replicate the behavior of `PhysicallyBasedMaterial` in your surface shader, retrieve both the specular `scale` and sample the specular `texture`, then multiply them together, as the following code demonstrates:

```
// Retrieve the emissive color tint from the CustomMaterial.
float specularScale = (half3)params.material_constants()
                      .specular_scale();

// Retrieve the primary texture coordinates.
float2 uv = params.geometry().uv0();

// Flip the UV coordinate's y-axis for models loaded
// from a USDZ or .reality file.
uv.y = 1.0 - uv.y;

auto tex = params.textures();
half specular = (half3)tex.specular()
              .sample(textureSampler, uv).r;

// Multiply the scale and the sampled value from the texture,
// and assign the result to the shader's specular property.
specular *= specularScale;
params.surface().set_specular(specular);
```

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

textures::opacity

Returns the opacity texture from the custom material.

Declaration

```
metal::texture2d<half> opacity() const thread
```

Overview

This function returns the texture from the `opacity` property of the shader's custom material. The opacity texture doesn't support sRGB images and discards color-space data embedded in the source image.

This property's primary purpose is to enable shader functions to calculate the value to pass to `set_opacity()`, but what you actually use it for in your shader function is completely up to you. If, for example, you don't need an opacity map for your entity, but you need two textures to define another attribute, you can use the `opacity` property to submit the second texture.

To replicate the behavior of `PhysicallyBasedMaterial` in your surface shader, retrieve both the opacity `scale` and the `opacityThreshold` from the material and sample the opacity `texture`. If the `opacityThreshold` is greater than `0.0`, compare the sampled value to the threshold and set opacity to either `1.0`, if the value is greater than the threshold, or `0.0` otherwise. If the `opacityThreshold` is equal to `0.0`, multiply the opacity scale and the sampled value together to get the final opacity value. The following code demonstrates:

```
// Retrieve the primary texture coordinates.
float2 uv = params.geometry().uv0();

// Flip the UV coordinate's y-axis for models loaded
// from a USDZ or .reality file.
uv.y = 1.0 - uv.y;

// Retrieve the opacity scale from the CustomMaterial.
float opacityScale = params.material_constants()
                     .opacity_scale();
float opacityThreshold = params.material_constants()
                     .opacity_threshold();

// Sample the opacity texture.
auto tex = params.textures();
half opacity = tex.opacity().sample(textureSampler, uv).r;

if (opacityThreshold > 0.0) {
    // If the opacity threshold is greater than 0, use masking
```

```
    // behavior and set the opacity to either 1.0 or 0.0
    // depending on the value of the opacity threshold. The
    // opacity scale is ignored when using a mask.
    opacity = (opacity <= opacityThreshold) ? 0.0 : 1.0;
} else {
    // If the opacity threshold is 0, then multiply the opacity
    // by the scale.
    opacity *= opacityScale;
}
params.surface().set_opacity(opacity);
```

Member Function

textures::clearcoat()

Returns the clearcoat texture from the custom material.

Declaration

```
metal::texture2d<half> clearcoat() const thread
```

Overview

This function returns the texture from the `clearcoat` property of the shader's material. The clearcoat texture doesn't support sRGB images and discards color-space data embedded in the source texture.

This property's primary purpose is to enable shader functions to calculate the value to pass to `set_clearcoat()`, but what you actually use it for in your shader function is completely up to you. If, for example, you don't need a clearcoat map for your entity, but you need two textures to define another non-color attribute, like roughness, you can use the `specular` property to submit the second texture.

To replicate the behavior of `PhysicallyBasedMaterial` in your surface shader, retrieve both the clearcoat `scale` and sample the clearcoat `texture`, then multiply them together, as the following code demonstrates:

```
// Retrieve the emissive color tint from the CustomMaterial.
float clearcoatScale = (half3)params.material_constants()
                     .specular_scale();

// Retrieve the primary texture coordinates.
float2 uv = params.geometry().uv0();

// Flip the UV coordinate's y-axis for models loaded
// from a USDZ or .reality file.
uv.y = 1.0 - uv.y;

auto tex = params.textures();
half clearcoat = (half3)tex.clearcoat()
                 .sample(textureSampler, uv).r;

// Multiply the scale and the sampled value from the texture
// and assign the result to the shader's clearcoat property.
clearcoat *= clearcoatScale;
params.surface().set_clearcoat(clearcoat);
```

Note: The code below doesn't take clearcoat roughness into account. For a code sample that replicates the full behavior of **`PhysicallyBasedMaterial`** using both clearcoat and clearcoat roughness, see **`textures::clearcoat_roughness()`**.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

textures::clearcoat_roughness()

Returns the clearcoat roughness texture from the custom material.

Declaration

```
metal::texture2d<half> clearcoat_roughness() const thread
```

Overview

This function returns the texture from the `clearcoatRoughness` property of the shader's custom material. The clearcoat roughness texture doesn't support sRGB images and discards color-space data embedded in the source texture.

This property's primary purpose is to enable shader functions to calculate the value to pass to `set_clearcoat_roughness()`, but what you actually use it for in your shader function is completely up to you. If, for example, you don't need a clearcoat roughness map for your entity, but you need two textures to define another non-color attribute, like roughness, you can use the `clearcoatRoughness` property to submit the second texture.

Note: Calling **`set_clearcoat_roughness()`** does nothing unless the custom material's lighting model is **`.clearcoat`**.

To replicate the behavior of `PhysicallyBasedMaterial` in your surface shader, retrieve the clearcoat roughness `scale` and sample the clearcoat roughness `texture`, then multiply them together. You must also retrieve the clearcoat `scale` and `roughness` because clearcoat roughness has no affect unless you also set clearcoat to a value greater than `0.0`.

The following code simulates the clearcoat and clearcoat roughness behavior of `PhysicallyBasedMaterial`.

```
// Retrieve the base color tint from the CustomMaterial.
float clearcoatScale = params.material_constants()
                       .clearcoat_scale();
float clearcoatRoughnessScale = params.material_constants()
                       .clearcoat_roughness_scale();

// Retrieve the UV texture coordinates for this fragment.
float2 uv = params.geometry().uv0();

// Invert the y-axis for models loaded from USDZ or
// .reality files.
uv.y = 1.0 - uv.y;
```

```
// Sample the clearcoat texture to get this fragment's value.
auto tex = params.textures();
half clearcoat = tex.clearcoat().sample(textureSampler, uv).r;

// Sample the clearcoat roughness texture to get this fragment's
// value.
half clearcoatRoughess = tex.clearcoat_roughness()
                         .sample(textureSampler, uv).r;

// Multiply the scales by the sampled values.
clearcoat *= clearcoatScale;
clearcoatRoughess *= clearcoatRoughnessScale;

// Use the the results as the values for rendering.
params.surface().set_clearcoat(clearcoat);
params.surface().set_clearcoat_roughness(clearcoatRoughess);
```

Member Function

textures::custom()

Returns the texture from the custom property of the custom material.

Declaration

```
metal::texture2d<half> custom() const thread
```

Overview

On `CustomMaterial`, the `custom` property allows you to pass a custom texture and custom vector into your shader functions. This function returns the texture from the `custom` property of the material.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Struct

material_parameters

An object the frameworks uses to pass uniform values for the current entity to shader functions.

Namespace

`realitykit::material`

Declaration

```
struct material_parameters
```

Overview

This object provides access to non-texture properties from the shader function's material. You can retrieve any material property that contains a `tint` or `scale` value by calling `params.material_constants()`.

Member Functions

`float3 base_color_tint() const thread`

Returns the tint value from the material's base color property.

`float roughness_scale() const thread`

Returns the scale value from the material's roughness property.

`float metallic_scale() const thread`

Returns the scale value from the material's metallic property.

`float opacity_scale() const thread`

Returns the scale value from the material's blending property.

`float opacity_threshold() const thread`

Returns the opacity threshold value from the material.

`float3 emissive_color() const thread`

Returns the color value from the material's emissive color property.

`float specular_scale() const thread`

Returns the scale value from the material's specular property.

Copyright © 2021 Apple Inc. All Rights Reserved.

`float clearcoat_scale() const thread`

Returns the scale value from the material's clearcoat property.

`float clearcoat_roughness_scale() const thread`

Returns the scale value from the material's clearcoat roughness property.

Member Function

material_parameters::base_color_tint()

Returns the tint value from the material's base color property.

Declaration

```
float3 base_color_tint() const thread
```

This function returns the `tint` value from the material's `baseColor` property and returns `(1.0, 1.0, 1.0)` if you don't set a base color tint on your material.

Member Function

material_parameters::roughness_scale()

Returns the scale value from the material's roughness property.

Declaration

```
float roughness_scale() const thread
```

This function returns the `scale` value from the material's `roughness` property and returns `1.0` if you don't set a roughness scale on your material.

Member Function

material_parameters::metallic_scale()

Returns the scale value from the material's roughness property.

Declaration

```
float metallic_scale() const thread
```

This function returns the `scale` value from the material's `metallic` property and returns `1.0` if you don't set a metallic scale on your material.

Member Function

material_parameters::opacity_scale()

Returns the scale value from the material's blending property.

Declaration

```
float opacity_scale() const thread
```

This function returns the `scale` value from the material's `opacity` property and returns `1.0` if you don't set an opacity scale on your material.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

material_parameters::opacity_threshold()

Returns the opacity threshold value from the material.

Declaration

```
float opacity_threshold() const thread
```

This function returns the `opacityThreshold` value from the material and returns `0.0` if you don't set an opacity threshold on your material.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

material_parameters::emissive_color()

Returns the color value from the material's emissive color property.

Declaration

```
float3 emissive_color() const thread
```

This function returns the `tint` value from the material's `emissiveColor` property and returns `(1.0, 1.0, 1.0)` if you don't set an emissive color tint on your material.

Member Function

material_parameters::specular_scale()

Returns the scale value from the material's specular property.

Declaration

```
float specular_scale() const thread
```

This function returns the `scale` value from the material's `specular` property and returns `1.0` if you don't set a specular scale on your material.

Member Function

material_parameters::clearcoat_scale()

Returns the scale value from the material's clearcoat property.

Declaration

```
float clearcoat_scale() const thread
```

This function returns the `scale` value from the material's `clearcoat` property and returns `1.0` if you don't set a metallic scale on your material.

Copyright © 2021 Apple Inc. All Rights Reserved.

Member Function

material_parameters::clearcoat_roughness_scale()

Returns the scale value from the material's clearcoat roughness property.

Declaration

```
float clearcoat_roughness_scale() const thread
```

This function returns the `scale` value from the material's `clearcoatRoughness` property and returns `1.0` if you don't set a metallic scale on your material.

Utility Functions

The RealityKit Metal APIs include a number of utility functions that aren't contained by a struct or class, but provide useful functionality for writing shader functions, such as converting a sampled normal map value into a normal vector.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

unpack_normal()

Unpacks a tangent-space normal value from a three-channel normal map value.

Namespace

`realitykit`

Declaration

```
half3 unpack_normal(half3 packed_normal)
```

Parameters

`packed_normal`

An RGB value sampled from a normal map texture.

Return Value

The unpacked surface normal vector.

Overview

Typical normal maps store surface normal vectors in an image texture using the `R`, `G`, and `B` channels to store the `X`, `Y`, and `Z` values from the normal vector. Because the `X`, `Y`, and `Z` values of a surface normal vector are between `-1.0` and `1.0` but sampled textures return a value between `0.0` and `1.0`, you need to convert the sampled value from the normal map texture before passing it to `surface_properties.set_normal()`. This function performs that conversion.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

Function

unpack_normal()

Unpacks a tangent-space normal value from a two-channel normal map.

Namespace

`realitykit`

Declaration

```
half3 unpack_normal(half2 packed_normal)
```

Parameters

`packed_normal`

A two-component value sampled from a normal map texture.

Return Value

The unpacked and reconstituted surface normal vector.

Overview

Some normal maps store surface normal vectors in an image texture using only the `R` and `G` channels to store the `X` and `Y` values from the normal vector. Because it's possible to calculate the vector's `Z`-value from the `X` and `Y` values, a two-channel normal map takes up less memory and can be stored in a smaller image file.

Because the `X`, `Y`, and `Z` values of a surface normal vector are between `-1.0` and `1.0` but sampled textures return a value between `0.0` and `1.0`, you need to convert the sampled value from the normal map texture before passing it to `surface_properties.set_normal()`. When using a two-channel normal map, you also need to reconstitute the `Z` value. This function handles both of these tasks.

July 8, 2021 | Copyright © 2021 Apple Inc. All Rights Reserved.

unpack_normal()

Unpacks a tangent-space normal value from a three-channel normal map value with a specified intensity.

Namespace

`realitykit`

Declaration

```
half3 unpack_normal(half3 packed_normal, half intensity)
```

Parameters

`packed_normal`

An RGB value sampled from a normal map texture.

`intensity`

The intensity at which to apply the surface details contained in the normal map.

Return Value

The unpacked surface normal vector.

Overview

Typical normal maps store surface normal vectors in an image texture using the `R`, `G`, and `B` channels to store the `X`, `Y`, and `Z` values from the normal vector. Because the `X`, `Y`, and `Z` values of a surface normal vector are between `-1.0` and `1.0` but sampled textures return a value between `0.0` and `1.0`, you convert the sampled value from the normal map texture before passing it to `surface_properties.set_normal()`. This function performs that conversion.

The `intensity` parameter adjust the intensity of the details contained in the normal map. A value greater than `1.0` makes surface details from the normal map stand out more, whereas values less than `1.0` mute those details.

Copyright © 2021 Apple Inc. All Rights Reserved.