RixShadingContext

Introduction

A `RixShadingContext` encapsulates a *shading context* - a description of the environment in which shading is to be performed. It is the primary domain of operation for the shader classes `RixBxdf`, `RixDisplacement`, `RixLight`, `RixPattern`, and `RixVolume`, and is also of high importance to `RixIntegrator`. Implementors of these classes and their associated plugins will thus need to be intimately familiar with the `RixShadingContext` class.

A `RixShadingContext` usually represents a collection of one or more geometric points that may be shaded. The group of points may arrive via ray tracing hits on geometry, or may be associated with tessellated micropolygons. In either case, shaders will make use of the shading context primarily in one of two ways: inquiring about geometric information through the `GetPrimVar()` and `GetBuiltinVar()` methods, and evaluation of the input parameters to the shading function is provided via the `EvalParam()` methods.

Generally, a shader should consider its associated shading context to be `const` read only. Shaders that need to write to the shading context will need to use a mutable shading context.

Fields

A `RixShadingContext` contains several fields that will be filled in by the renderer or integrator and may contain useful information pertinent to shading computations. For all shader types, this information should be considered read-only, which is enforced through the use of a `const RixShadingContext`. `RixIntegrator` plugins may be required to fill in some of these fields through the course of their operation.

- `int numPts`: The number of points in the shading context. All non-uniform values can be assumed to have this length.
- `struct scTraits`: This struct contains several informational fields.
    - `eyePath`: A flag which is set to 1 if the shading context was created in the context of an eye path.
    - `lightPath`: A flag which is set to 1 if the shading context was created in the context of a bidirectional light path.
    - `primaryHit`: A flag which is set to 1 if the shading context is directly visible to the camera.
    - `missContext`: A flag which is set to 1 if the shading context represents a ray *miss*. A miss context has no geometric information associated with it, cannot provide primvar or parameter evaluation services, and the only available builtin variables will be the ones associated with the incident ray.
    - `reyesGrid`: A flag which is set to 1 if the shading context is associated with a tessellated micropolygon representation. This is true in certain displacement or cached execution contexts.
    - `RixSCShadingMode shadingMode`: This enumeration conveys the current *mode* associated with the current shader execution. Knowledge of the shading mode may allow the shader to minimize certain costs associated with the current execution, i.e. by avoiding certain parameter evaluations when it is known that those parameters do not pertain to the current execution.
    - `bxdf, opacity, displacement, subsurface, volume`: Pointers to instances of the appropriate shader type. It is rare for a shader to need knowledge of other shader types via these fields. If such knowledge is required, the use of the methods `GetBxdf(), GetOpacity(),` etc is encouraged instead of directly accessing the `scTraits`.
- `int* integratorCtxIndex`: This array contains a mapping from the shading context to arrays in the `RixIntegratorContext`: a shading point of index `i` is associated with the corresponding index `integratorCtxIndex[i]`. For example, the shading point with index `i` can figure out the camera ray that ultimately gave rise to the shading point by accessing the `primaryRays[integratorCtxIndex[i]]` field of `RixIntegratorContext`.
- `int* rayId`: This array should usually be ignored by shaders. It is typically used by integrators to track a correspondence between the points in the shading context and any ray specific data structures in the integrator.
- `RtColorRGB *transmission`: The transmittance associated with the incident ray. This transmittance will typically be non-zero if the incident ray traveled through participating media. Most shaders other than volume integrators will ignore this field, while integrators will need to inspect this field in order to compute the total transmittance along a path.
- `float *pointWeight`: If the shading context was created via an importance sampling method, this is the associated weight of that method. Most shaders other than volume integrators will ignore this field, while integrators may need to multiply the shader contributions by this weight.
- `int *pointSampleCount`: Only used by volume integrators to indicate the number of samples taken for multiple scattering. Integrators may need to multiply this value into the total path throughput.
- `RtColorRGB *opacityThroughput`: Per-point part of the throughput resulting from opacity continuations along the ray that produced the associated hit.
- `RtColorRGB *opacity`: Per-point combined opacity: the product of presence and opacity for the given point.
- `float *opacityStochasticWeight`: Per-point probability hit-testing weight. Inverse of the probability of having selected this hit. When dealing with non-fully-opaque surfaces, and when using probabilistic hit-testing, the combined opacity can be interpreted as the probability that we actually hit the surface. When only considering a scalar presence value (i.e. opacity is one, so combined opacity is monochromatic), the probability is usually the inverse of the scalar presence value, and no special weighting needs to be applied to the shading on hits. However, when dealing with arbitrary combined opacity, it is necessary to weight the shading on hit by the product of: `opacity` (the combined opacity on hit), and `opacityStochasticWeight` (the inverse hit-testing probability).
- `RixRNG *m_rngCtx`: A per-point array of random number contexts. These can be used by shaders to generate stratified random numbers. A NULL value here means the current shading mode does not support random number generation, and shader writers should check for a NULL value before relying on these contexts.

Builtin Variables

Builtin variables represent values which are usually always present for all geometry types, and are often used in fundamental shader calculations. Some variables like `P`, the shading position, and `Nn`, the normalized shading normal, are derived from the geometry definition. Others like `incidentRaySpread` and `VLen` are derived from the incoming rays or from a combination of the geometry definition and the incoming rays.

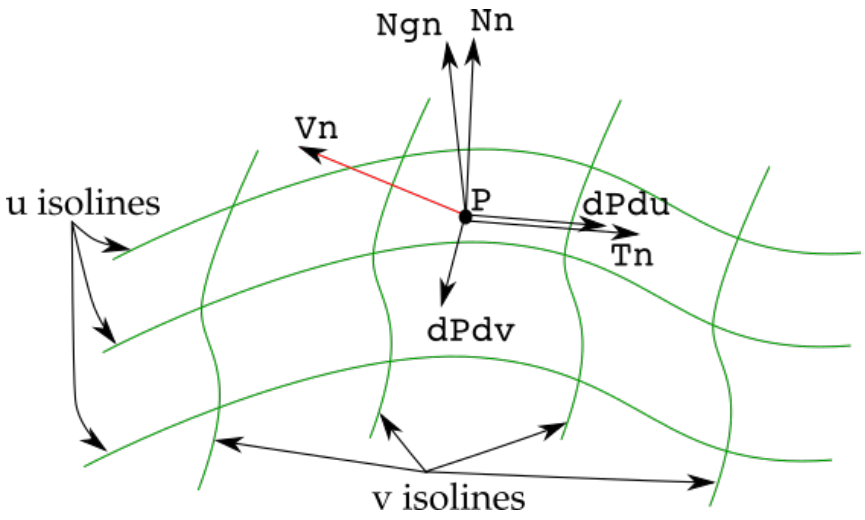
Builtin variables may be queried using the `GetBuiltinVar()` method. The `BuiltinVar` enumeration is used to select one of the following variables:

- `RtFloat3 P`: The position of the point being shaded. Derived directly from the underlying tessellated representation used during hit testing.
- `float PRadius`: When the geometry is a collection of hits, the radius of the incident ray at P. Otherwise it is half the micropolygon radius at P.
- `RtFloat3 Po`: If the geometry was displaced, the value of the undisplaced position that became P after displacement
- `RtFloat3 Nn`: The normalized *shading normal*, which is the normal that is typically used for shading because it has the smoothest appearance in all circumstances.
- `RtFloat3 Ngn`: The normalized *geometric normal*. The geometric normal is the normal derived directly from the underlying tessellated representation used during hit testing, and may be preferred to the shading normal for certain uses such as horizon culling.
- `RtFloat3 Non`: The normalized undisplaced normal, i.e. equivalent to Nn if the geometry did not undergo displacement.
- `RtFloat3 Tn`: The shading tangent. Typically a vector which is orthonormal to the shading normal Nn, often used in conjunction with same to create an orthonormal basis.
- `RtFloat3 Vn`: The normalized view vector, pointing away from P. In ray tracing contexts, this is the reverse direction of the incoming ray. In other contexts where there is no view direction, this may be set to the same as Nn.
- `float VLen`: The length of the view vector.
- `float curvature`: The local mean surface curvature, which is the average of `curvatureU` and `curvatureV`.
- `float incidentRaySpread`: How much the ray radius increases for each unit distance the incident travels.
- `float incidentRayRadius`: Radius of incident ray at P.
- `int incidentLobeSampled`: RixBXLobeSampled id of incident rays.
- `float mpSize`: The micropolygon size. 0 for non-tessellated surfaces. May be used as a hint for biasing purposes.
- `float biasR`: A renderer-computed bias used for reflected rays.
- `float biasT`: A renderer-computed bias used for transmitted rays.
- `float u`: The position of the current point on the current surface in parametric space.
- `float v`: The position of the current point on the current surface in parametric space.
- `float w`: The position of the current point on the current surface in parametric space.
- `float du`: The size of the ray footprint (radius) in parametric space.
- `float dv`: The size of the ray footprint (radius) in parametric space.
- `float dw`: The size of the ray footprint (radius) in parametric space.
- `RtFloat3 dPdu`: The surface derivative of P with respect to u.
- `RtFloat3 dPdv`: The surface derivative of P with respect to v.
- `RtFloat3 dPdw`: The surface derivative of P with respect to w.
- `RtFloat3 dPdtime`: The instantaneous velocity of P in current space.
- `float time`: The shading time of the point being shaded, normalized between shutter open and shutter close.
- `int Id`: The value of any Attribute "identifier" "id" associated with the geometry.
- `int Id2`: The value of any Attribute "identifier" "id2" associated with the geometry.
- `float outsideIOR`: The incident index of refraction.
- `RtFloat3 Oi`: The opacity.
- `RixLPEState lpeState`: The current LPE state.
- `int launchShadingCtxId`: The ID of the shading context that launched the incident ray.
- `RtFloat3 motionFore`: Forward 2D raster-space motion vector.
- `RtFloat3 motionBack`: Backwards 2D raster-space motion vector.
- `float curvatureU`: The principal curvature in the parametric u direction, computed via dot(dNu, dPu) / |dPu|^2, where dNu is a difference between two surface normals computed at P and P' (P' being a point near P offset in the u direction) and dPu is the difference between P and P`. This is a signed quantity.
- `float curvatureV`: The principal curvature in the parametric v direction, computed via dot(dNv, dPv) / |dPv|^2, where dNv is a difference between two surface normals computed at P and P' (P' being a point near P offset in the v direction) and dPv is the difference between P and P`. This is a signed quantity.
- `RtFloat3 dPcameradtime`: The instantaneous velocity of P relative to the camera.
- `float wavelength`: The wavelength of the incident ray.

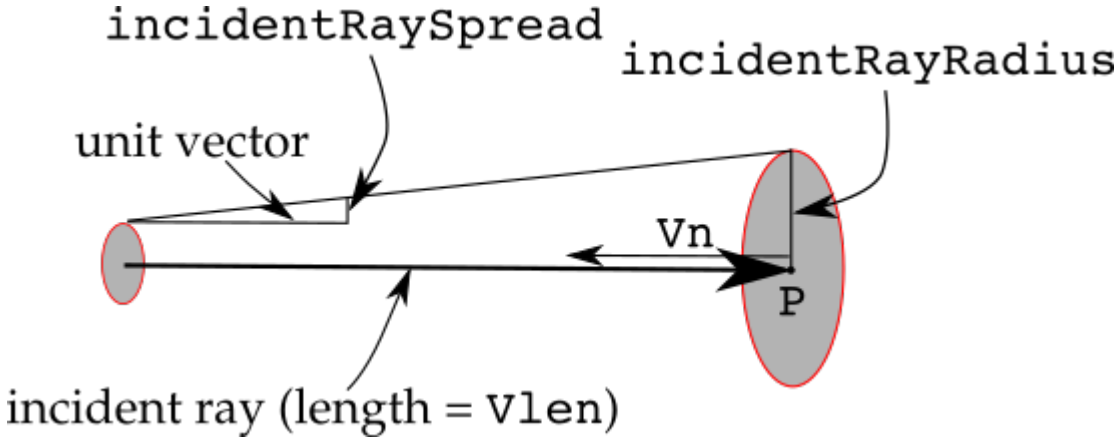
`BuiltinVar` must be passed a pointer to a pointer of the appropriate type for the variable. The returned pointer points at *const* storage containing the value of the variable, one value for each point in the `RixShadingContext`. This storage is owned by the renderer and is valid for the duration of the shader execution.

For normal shading contexts, `SetBuiltinVar()` is a no-op. Shaders that wish to change the values of builtin variables will need to create a mutable shading context.

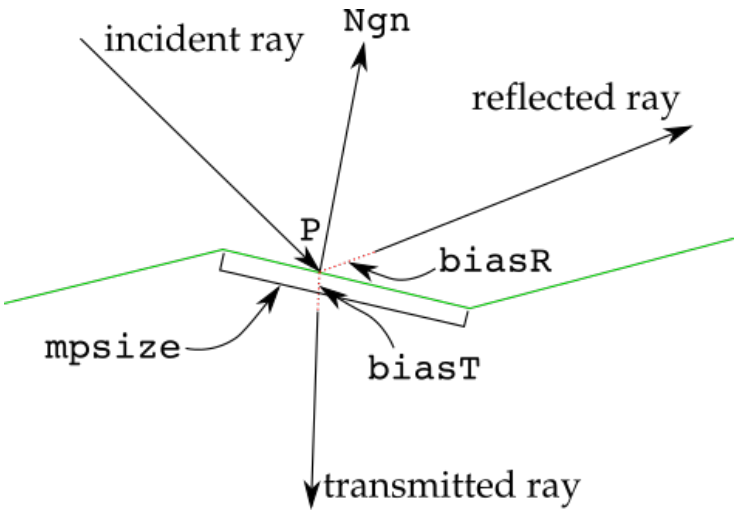
The following diagram illustrates several fundamental builtin variables: the surface position `P`, the normalized shading normal `Nn` and the normalized shading tangent `Tn`, the geometric normal `Ngn`, and the parametric derivatives `dPdu` and `dPdv`.



The following diagram illustrates the `incidentRaySpread` and `incidentRayRadius` properties of the ray which are related to ray differentials, and are important to the computation of filter widths for antialiasing.



The next diagram illustrates an incident ray and potential reflected rays and transmitted rays that may be returned by a Bxdf as the next indirect ray fired by an integrator. Here, the green lines represent a side view of the tessellated micropolygon geometry. Note that the geometric normal Ngn and the micropolygon size mpsize is derived directly from the tessellated representation, while the biasR and biasT variables can be used to bias the origins of the indirect rays.



Primitive Variables

Primitive variables (often referred to as *primvars*) represent values which may be optionally present on the geometry. These primvars are uniquely named and may be supplied with the geometry definition. Shaders are usually written knowing the name and data type of certain primvars in advance, such as texture coordinates. However, since the primvars are optional, shaders are expected to fail gracefully if the primvars are missing.

The first variant of `GetPrimVar()` can be used to efficiently query the existence, type, and array length of a primvar. The other variants return a pointer to *const* storage containing the value of the primvar in the `var` output parameter, if the primvar exists. If the primvar cannot be found, storage will still be allocated and will be filled with the value of the input `fill` parameter. The optional `radius` output parameter if supplied will point at storage containing the approximate isotropic filter radius associated with the primvar. This value is computed by the renderer from the ray differentials and can be used to perform any necessary shader antialiasing.

There are also variants of `GetPrimVar()` which return the partial derivatives of the primvar measured with respect to the builtin variables u and v. These methods may be used in certain conditions that are otherwise hard to filter correctly.

For normal shading contexts, `SetPrimVar()` is a no-op. Shaders that wish to change the values of primitive variables will need to create a mutable shading context.

> ⚠ Primitive variables of type `RtFloat3` and `RtMatrix4x4` are automatically transformed and returned in **current** space, which is generally considered the optimal space for the renderer to perform operations. Plugin authors should make no assumptions about what current space actually maps to, and should transform such primvars from current to another space if required using the transformation routines.

Parameter Evaluation

Evaluation of the input parameters to the shader is provided via the `EvalParam()` methods. Input parameters may be constant (the direct parameters to the shader invocation), or may be *connected* to an upstream node and trigger an upstream of that node (and in turn, any of its dependencies). All such upstream computation will be automatically cached by the renderer for the duration of the `RixShadingContext`.

The desired input parameter to the shader is selected by an integer `paramId`, which is the ordinal position of the parameter in the parameter table returned by `RixShadingPlugin::GetParamTable()`. Callers are expected to know the `paramId`, the type of the associated parameter and are expected to pass a pointer to a pointer of the appropriate type to `EvalParam()`.

If the value cannot be obtained from the shading network or from the direct parameters associated with the shader invocation, the result will be filled with the default value provided in `dflt`.

Depending on whether the input parameter is the default value, or a constant value or a connection, the call to `EvalParam()` will either return a pointer to storage containing a single value or an entire `numPts` worth of values. This can be changed by setting the optional parameter `promoteToVarying` to be true, in which case the storage for the result will be guaranteed to be an entire numPts worth of values; however, this does not alter the return value. This storage is allocated by the renderer and has a lifetime associated with the shading context; it does not need to be managed by the caller.

The return value of `EvalParam` is a value of `RixSCDetail`: `k_RixSCUniform` indicating that the input parameter was uniform (a constant or default value) or `k_RixSCVarying` indicating the parameter was varying (a connection). If the value cannot be found (typically because it does not have the correct matching type, i.e. the wrong version of `EvalParam` was used) then `k_RixSCInvalidDetail` will be returned.

In some circumstances, users of `EvalParam()` that do not know the `paramId` (perhaps due to the use of a dynamic parameter table) or the type of the parameter can introspect for the paramId or information about the parameter via the `GetParamId()` and the `GetParamInfo()` methods.

Some examples for reading different types of input parameters are shown below.

```
// Read a uniform integer value, and store the result in the
// RtInt noiseType variable. m_noiseType is a PxrCustomNoise
// member variable that contains the default noiseType value.
RtInt *noiseTypePtr;
sctx->EvalParam(k_noiseType, -1, &noiseTypePtr, &m_noiseType, uniform);
RtInt const noiseType(*noiseTypePtr);

// Read a varying float value for the threshold input parameter.
// m_threshold is a PxrCustomNoise member variable that contains
// the default value.
RtFloat *threshold;
sctx->EvalParam(k_threshold, -1, &threshold, &m_threshold, varying);

// Read one value from a varying float[2] array.
RtFloat *repeatUV0, *repeatUV1;
sctx->EvalParam(k_repeatUV, 0, &repeatUV0, &m_repreatUV, varying);
// Read the other value from a varying float[2] array. Note that
// the arrayIndex parameter is set to 1 to read the second value.
sctx->EvalParam(k_repeatUV, 1, &repeatUV1, &m_repreatUV, varying);

// Read in a float[3] array of values into a RtFloat3 variable.
RtFloat3 *scale;
sctx->EvalParam(k_scale, -1, &scale, &m_scale, varying);

// Read in a varying color value.
RtColorRGB* defaultColor;
sctx->EvalParam(k_defaultColor, -1, &defaultColor, &m_defaultColor, varying);
```

Memory Management

`RixShadingContext` provides a fast memory allocation service tailored to shaders that need to execute quickly. The main provider of this service is the `Allocate()` routine; the `New()` routine and the `class Allocator` are wrappers around `Allocate()`. The backing store for this memory allocation are memory pools which are tailored specifically to the lifetime requirements of `RixPattern` and `RixBxdf`, with the latter category equating to the lifetime of the `RixShadingContext`. Clients that use these memory management services should use "placement new" semantics, and should not rely on the invocation of a destructor.

Tracing Rays

`RixShadingContext` provides a limited service for tracing rays. The `GetNearestHits()` routine allows shaders to trace rays to determine the nearest hit. These rays do not trigger shading on the hit geometry, and should thus be considered *geometric probe* rays: the only information that can be returned from `GetNearestHits` is a `RtHitPoint` struct, which contains a minimal set of geometric information including the distance, the `P`, `Ng`, `u`, `v` builtins of the hit geometry, and the filter and micropolygon sizes. A ray that missed is indicated by the distance being set to 0. While limited, this service is sufficient to allow for calculation of effects such as ambient occlusion. This service should not be confused with the `RixIntegratorContext` ability to trace rays, which is provided only to implementors of `RixIntegrator`.

Transformations

Transformation of `RtFloat3` data between two coordinate systems known to the renderer can be accomplished using the `Transform()` method. The interpretation of the `RtFloat3` data as point, vector, or normal data is specified using the `TransformInterpretation` enum. The array of data must be `numPts` in size and the data will be transformed in place. A non-zero return value indicates a failed transformation, typically due to unknown coordinate systems being specified.

In addition, transform matrices between two coordinate systems can be returned directly via the `GetTransform()` method. The `matrix` output parameter points at storage allocated by the renderer containing the matrices. The size of this storage is indicated by the `numMatrices` output parameter. If the transformation is time varying, the `numMatrices` returned will be the same as `numPts`. If the transformation is uniform, `numMatrices` will be set to 1. A non-zero return value indicates a failed transformation, typically due to unknown coordinate systems being specified.

Querying Ray Properties

As long as the `RixIntegrator` used supports the appropriate ray property query, plugin authors may want to query ray properties such as the ray depth or eye throughput, in order to allow for artistic control or optimization. For instance, as an optimization a `RixBxdf` may want to skip the evaluation of a particularly expensive lobe, if the current ray depth of the hit point is beyond some arbitrary threshold. This service is provided by the `GetProperty()` routine on the `RixShadingContext.` Callers of this routine are responsible for allocating and deallocating the `result` memory. The integrator is responsible for filling in the result. Below is a code snippet demonstrating how to use this feature.

```
// Query ray depths for each point in the shading context
int nPts = shadingCtx->numPts;
int* rayDepths = new int [nPts];
if (vCtx->GetProperty(k_RayDepth, rayDepths))
{
    // Do something expensive for small ray depths, or something cheaper for large ray depths
}
delete[] rayDepths;
```

For documentation on the available ray properties, please consult the integrator ray property query documentation.