Writing Projections

Introduction

This documentation is intended to instruct developers in the authoring of custom *projections*. Developers should also consult the `RixProjection.h` header file for complete details. The source code for the `PxrOrthographic and PxrPerspective` projection plugins may be found in the `plugins/projection/simple` subdirectory of the PixarRenderMan-Examples package.

A projection plugin is used to model camera and lens behavior. These plugins are responsible for taking stratified random samples as input from the renderer and turning these into primary camera rays.

`RixProjectionFactory`

`RixProjectionFactory` is a subclass of `RixShadingPlugin`, and therefore shares the same initialization, synchronization, and parameter table logic as other shading plugins. Projections do not support lightweight instances, and therefore `CreateInstanceData()` should not be overridden as any created instance data will not be returned to the factory.

The renderer uses the `RixProjectionFactory` object by invoking `RixBxdfFactory::CreateProjection()` to obtain a `RixProjection`. A description of various options associated with the current render are provided to the factory via the `RixProjectionEnvironment &env`. This class includes information about the current format (the width and height of the image in pixels and the pixel aspect ratio), the screen window, the shutter time values, the near and far clipping planes, and the world to camera transform. Developers are expected to use this information, along with the arguments supplied to the plugin via the `RixParameterList pList`, to create an instance of `RixProjection` that encapsulates the information necessary to model the desired camera and lens behavior.

`RixProjectionEnvironment` contains one field that can be altered: the `deepMetric` field. This is used to indicate to the renderer the depth metric used for computing Z values when rendering deep output. The default value of `k_cameraZ` indicates the renderer should use the distance strictly in the Z axis and can assume that all camera rays go forward in the +Z direction, while `k_rayLength` indicates that the distance should be measured along the ray direction, as the rays may be go in either the -Z or +Z direction. All other fields on `RixProjectionEnvironment` should be considered read only. For example, a plugin that implements the standard perspective projection should return `k_cameraZ`, while a plugin that implements a cylindrical panoramic projection should use `k_rayLength`.

`RixProjection`

Once a `RixProjection` object is obtained, the renderer will invoke the following methods:

- `RixSCDetail RixBxdf::GetProperty(ProjectionProperty property, void const** result)`

  Projection plugins will be queried via this method at the beginning of rendering for properties that are of interest to the renderer. The value of the property is passed back to the renderer via the `result` parameter. These properties are invariant during the frame (i.e. they are *options*). The current list of properties include:

  - `k_DeepMetric`: This is used to indicate to the renderer the depth metric used for computing Z values when rendering deep output. `k_DeepMetric` takes one of two values: `k_cameraZ` or `k_rayLength`. `k_cameraZ` indicates the renderer should use the distance strictly in the Z axis and can assume that all camera rays go forward in the +Z direction, while `k_rayLength` indicates that the distance should be measured along the ray direction, as the rays may be go in either the -Z or +Z direction.
  - `k_DicingHint`: The plugin should return an `enum DicingHint` indicating the general strategy the renderer should use to dice geometry. `DicingHint` takes one of three values: `k_Orthographic`, `k_Perspective`, and `k_Spherical`, and should be set to the type of camera projection that is closest to the one being implemented in the projection plugin. Not setting this property correctly means the renderer may underdice or overdice geometry in the scene, which may impair performance or lead to visual artifacts.
  - `k_FieldOfView`: In conjunction with `k_DicingHint` returning a value of `k_Perspective` or `k_Spherical`, the plugin should return a floating point value indicating the field of view of the projection in degrees. This value is used as a hint to the renderer for dicing geometry purposes. Note that the projection plugin itself is still responsible for actually implementing a camera model that takes into account this field of view.
  - `k_FStop, k_FocalLength, k_FocalDistance`: The plugin describes the desired depth of field (defocus) settings to the renderer. All three properties are floats. If the projection plugin returns values for these properties, the renderer will use them as part of the computation for the initial ray directions supplied to the projection plugin.

  As all of these properties are invariant during the frame, Projection plugins should return `k_RixSCUniform` for any supported properties, otherwise they should return `k_RixSCInvalidDetail`.

- `void RixBxdf::Project(RixProjectionContext &pCtx)`

  The `Project` method is the primary entry point for the plugin. The plugin is primarily responsible for taking the input (screen and lens samples) and mapping these to the output (camera rays and tint). Both the input and output are encapsulated in the `RixProjectionContext` class. The fields of this class are as follows:

  - `int numRays`: The number of rays that the projection plugin is expected to compute. All inputs and outputs on the `RixProjectionContext` class are sized to this number.
  - `RtPoint2 const *screen`: This input contains the screen samples in screen space, which is a 2D coordinate system where X typically has the range [-aspect, aspect] and Y has the range [-1, 1], where aspect is the screen aspect ratio. The exact values of these coordinates are determined by the format, the screen, and crop window settings supplied to the renderer.
  - `RtPoint2 const *lens, *aperture`: The lens samples are the raw canonical samples with stratified distribution in the [0, 1), [0, 1) unit square. The aperture samples are the lens samples warped into a distribution in the [-1, 1], [-1, 1] square by the renderer's depth of field calculations. These calculations will be determined by the description of the aperture supplied to the renderer as part of the scene description. If your plugin does not want to compute depth of field effects, it may choose to ignore these inputs.

- `float *time`: The time samples are the raw stratified samples distributed in the [0, 1) range, where 0 is interpreted as the shutter opening time and 1 is the shutter closing time. The renderer computes a distribution of these values according to the shutter opening description supplied as part of the scene description. Projection plugins may also alter these time values (e.g., for rolling shutter or strobe effects), so long as they remain in the [0, 1) range.
- `RtRayGeometry *rays`: The primary output of a projection plugin. Plugins are expected to use the inputs above, plus any information from the initial `RixParameterList pList` to fill the `origin`, `direction`, `originRadius`, and `raySpread` fields of the ray to model the desired camera and lens behavior. Plugins can optionally also override the `mindist` and `maxdist` fields for any desired clipping effects. All ray properties are defined in terms of camera space, with the camera centered at the origin looking down the +Z axis. Directions should always be unit normalized or set to zero. Rays with a zero direction vector will be culled.
- `RtColorRGB *tint`: An optional tint which is applied to the beauty channel of shaded rays prior to pixel filtering. Defaults to white (1, 1, 1) indicating that the values should be unchanged. Projection plugins can change the tint value to create vignetting, chromatic aberration, spectral bokeh, or other effects.