Display Services

Introduction

The display services API provides the link between integrator plugins and the frame buffer. Typically, an integrator will receive a batch of camera rays as input and produce a radiance and opacity estimate for each ray as output. It may also output geometric quantities, breakdowns of the radiance by light path expressions, and other AOVs. All of these outputs are passed back to the renderer by calling the methods in the display services API.

The display services are also made available to pattern and bxdf plugins while the integrator is running. This use of the display services is rare, but can be helpful for debugging or special effects. For example, the PxrTee pattern passes a float or color value from its input to its output while simultaneously writting it to a display channel via the display services.

After the integrator has returned control to the renderer, any sample filter plugins in use will be run and given a chance to read and modify the values that were sent to the display services. Once these are finished, the renderer will use the final values to update the adaptive sampler and then pixel filter and accumulate them into the frame buffer for display.

API

RixDisplayChannel

This struct conveys information about one of the display channels currently in use (normally plugins will be given a pointer to an array of these structures). The key property here is the `id` which serves as a handle to the channel and can be passed to most of the calls in the display services. The other fields are more informational: the `style` tells a plugin whether the channel is for a light path expression or for a simple AOV. The `type` tells the plugin whether the channel is a scalar float, an RGB color, or something else. Finally, the `channel` simply gives the name of the channel.

RixDisplayServices

The `Splat()` and `Write()` methods update a table containing the values to be pixel filtered for each channel for each ray. The main difference between the two is that `Splat()` is accumulative and simply sums the new value into the current entry in the table. It is useful for situations where an integrator wants to add light received at the camera from each bounce of a path. The `Write()` methods, on the other hand, simply overwrite the existing value in the table. These are most useful for channels conveying things like normals, positions, or other geometric quantities where the values in the channel are not additive. Note, however, that a call to `Write()` does not overwrite the value in the frame buffer. It only overwrites the value to be combined into the frame buffer after the integrator and sample filters return.

The `WriteDistance()` method modifies the depth of a sample. This primarily affects deep images and depth channels such as "z" or "zfiltered" in shallow images. By default, the depth will be distance to the camera ray hit. However, for volumes this initial ray hit will come from the front of the volume envelope. Instead, an integrator will want to set it to the distance to an actual light scattering or absorption interaction.

The `AddSample()` can be used to add additional samples along a camera ray where they have the same screen position but different depths and opacities. This can be useful for an integrator if a camera ray passes in a straight line through multiple transparent surfaces or volumes, each of which partially attenuates or scatters light into the ray but does not significantly deflect it. (It should *not* be used for refraction or reflection bounces.) This is primarily used so that deep images can have samples at all relevant points rather than just attributing everything to the closest hit point.

Pattern and Bxdf plugins can use `GetDisplayChannels()` to get a list of available display channels in order to look up a `RixChannelId` to write to with the `Splat()` and `Write()` methods. Integrator plugins may also use this. However, the same information is provided to integrators during their initialization. It may be more efficient for them to look up channel ids and cache them at that time.

Finally, `DiscardIteration()` may be used to tell the renderer to abandon everything sent to the display services during this call to the integrator and not to actually commit them to the frame buffer. For example, Metropolis algorithms often have a period of startup bias before the results are sufficiently randomized. A Metropolis-based integrator could use this function to prevent results from this period from contaminating the frame buffer.

See Also

The `RixLPE` interface assists integrators by matching light paths to light path expressions. Its main methods take a pointer to the display services and automatically call them on the integrator's behalf for any matched channels.

The utility functions in `PxrGeoAovs.h` can be used by an integrator to build a list of channels matching a standard list of names for geometric AOV and then compute those values from the shading contexts and call display services to show them.