# Writing Display Filters

This documentation is for developers who wish to author custom display filter plugins. Note that these are distinct from display (non-filter) plugins. See the `RixDisplayFilter.h` header for complete details.

Display filter plugins interpose between the renderer core's framebuffer and the display plugins. They may read and write the contents of various AOV channels for the pixels in a region that is on its way to any display plugins for I/O.

Display filters do *not* apply to deep data!

## Control Flow In Detail

When the renderer wishes to send a bucket (a.k.a. tile) of 2D pixels to the display plugins for I/O such as display in an interactive frame buffer or to commit to an image file, it first makes a copy of the pixels as they currently appear in its internal framebuffer. This copy includes all of the AOV channels associated with the current camera. (Multi-camera scenarios such as stereo renders have a separate framebuffer for each camera and send pixels to their associated display plugins at different times.)

If there is an active display filter plugin for the scene, the renderer will then invoke that plugin and give it access to the pixels. The filter is then free to read from and write to any of the channels for any of the pixels within the bucket.

Note that the render only directly invokes a single display filter at a time. Given multiple display filters defined for a scene, the last one wins. E.g.:

```
DisplayFilter "PxrCopyAOVDisplayFilter" "move" "string readAov" "sampleCount" "string writeAov"
"gradedSampleCount"
DisplayFilter "PxrGradeDisplayFilter" "grade" "string aov" "gradedSampleCount" "color whitePoint" [128 128 128]
DisplayFilter "PxrDisplayFilterCombiner" "combo" "reference displayfilter[2] filter" ["move" "grade"]
```

As shown here, however, a display filter may be given a reference to another that precedes it. The standard `PxrDisplayFilterCombiner`, for example, simply runs a chain of other display filters in sequence.

Once the display filter invoked by the renderer returns, the AOVs in the copy of the pixels are demuxed to all of the display plugins that need to show or write them for that bucket.

It is very important to understand that the changes made by a display filter plugin are never actually commited back to the framebuffer. They apply only in passing on a copy of pixels on their way to the display plugins. For IPR renders and checkpointing, this means that display filters can be applied continuously as the render resolves and do not need to worry about their cumulative effects with each progressive pass or checkpoint.

## Implementing the `RixDisplayFilter` Interface

Display filters implement the `RixDisplayFilter` interface found in `RixDisplayFilter.h`. A `RixDisplayFilter` is a subclass of `RixShadingPlugin`, and therefore shares the same initialization, synchronization, and parameter table logic as other shading plugins. Therefore to start developing your own display filter, you can `#include "RixDisplayFilter.h"` and make sure that your display filter class implements the required methods inherited from the RixShadingPlugin interface: `Init()`, `Finalize()`, `Synchronize()`, `GetParamTable()`, and `CreateInstanceData()`. You should also use the `RIX_DISPLAYFILTERCREATE()` and `RIX_DISPLAYFILTERDESTROY()` macros to define the `CreateRixDisplayFilter()` and `DestroyRixDisplayFilter()` functions for creating and destroying instances of your class.

Often, you will want to have your `CreateInstanceData()` map AOV names from the parameter list to `RixChannelID` values. These channel ids can be found by inspecting the `RixIntegratorEnvironment`. E.g.:

```
RixRenderState* state = reinterpret_cast<RixRenderState*>(ctx.GetRixInterface(k_RixRenderState));
RixRenderState::FrameInfo frame;
state->GetFrameInfo(&frame);
RixIntegratorEnvironment const* env = reinterpret_cast<RixIntegratorEnvironment const*>(
    frame.integratorEnv);

std::string name("foo");

std::vector<RixChannelId> ids;
for (int index = 0; index < env->numDisplays; ++index)
    if (env->displays[index].channel == name)
        ids.push_back(env->displays[index].id)
```

Note that there may be more than one match for a given name. This may be caused by either the `"string source"` parameter to a display channel or the particular AOV being output multiple times. Typically, a display filter will store the list of matches and apply itself to all of them.

Finally, your display filter subclass must implement the main `Filter()` method of the RixDisplayFilter interface. This method may be called simultaneously from multiple threads for the same plugin and data instances and so must be thread-safe and re-entrant.

## Defining `RixDisplayFilter::Filter()`

The `Filter()` method is where the action takes place:

```
virtual void Filter(
    RixDisplayFilterContext& fCtx,
    RtPointer instanceData) = 0;
```

Each time the renderer wants to send out a bucket to the display it calls the `Filter()` method with both a `RixDisplayFilterContext` giving access to the bucket data, and with the blind pointer to the instance data that the plugin setup for itself in `CreateInstanceData()`.

Most plugins will immediately cast this later pointer to a pointer or reference to the internal structure that the plugin uses to stash its internal data for this particular plugin instance. Note that at startup, the renderer will typically only create a single instance of the plugin *class* via a call to the function defined by the `RIX_DISPLAYFILTERCREATE()` macro. But it may call `CreateInstanceData()` many times, each with a different *parameter list* from the scene. Class member data will thus be shared across all invocations while the `instanceData` pointer passed back to the `Filter()` will be unique to its invocation for a specific parameter list.

The `RixDisplayFilterContext` currently provides four main things:

Bucket coordinates:
: The `xmin`, `ymin`, `xmax`, and `ymax` fields give the coordinates of the bucket's rectangle relative to the image. They are inclusive on the lower bounds and exclusive on the upper bounds. Note that due to filter padding, they may be negative or extend outside the image proper.

Read/write access:
: The overloads of the `Read()` and `Write()` method allow the plugin to inspect and change the pixel data. The coordinates must be inside the bucket boundary or else a read will simply store a black result and a write will be silently ignored. Note that with multi-camera rendering not all possible channel ids found in the integrator environment may be valid during a given call to `Filter()`. The read and write functions will return false when called with a channel that is not valid for the current instance data.

Reading arbitrary regions:
: Normally the read functions are only allowed to read pixels within the boundaries of the present bucket. However, for image processing operations that require a neighborhood it may be useful to be able to read from a enlarged set of pixels from around the bucket. The `ReadRegion()` offers the ability to fetch a copy of an arbitrary rectangle of pixels from anywhere in the framebuffer (with zero padding for pixels outside the image). Like `Read()`, it may fail for some channels. Note that due to multi-threading, successive calls to `ReadRegion()` may produce different results (and different results from the equivalent `Read()`) as other rendering threads may continue to update the framebuffer and this may produce transitory bucket artifacts if a display filter does not take care. By contrast, the standard `Read()` and `Write()` calls work on data locked to the current thread and so are stable. See the source to the PxrEdgeDetect plugin in the examples bundle for a use of this call.

Access to other filters:
: The `IsEnabled()` call allows a display filter plugin to get a pointer to the instance data for another plugin instance and determine if that instance is currently enabled. This allows one `Filter()` invocation to recursively invoke another filter instance's `Filter()` function. See the source to the PxrDisplayFilterCombiner in the examples bundle for use of this to chain together plugins.