OpenFX Documentation

Release 1.4+

ofxa

Jan 22, 2022

CONTENTS

The OpenFX documentation is organized as follows:

- The programming guide contains everything to get started with OpenFX to create a new plug-in or host application
- The reference guide contains the full documentation about the OpenFX protocol and design

This documentation is also available online and can be downloaded as a PDF or HTML zip file.

This manual is maintained largely by volunteers.

CHAPTER

ONE

OPENFX REFERENCE

This is a mostly complete reference guide to the OFX image effect plugin architecture. It is a backwards compatible update to the 1.2 version of the API. The changes to the API are listed in an addendum.

1.1 Structure of The OFX and the Image Effect API

1.1.1 The Structure Of The Generic OFX API

OFX is actually several things. At its base it is a generic plug-in architecture which can be used to implement a variety of plug-in APIs. The first such API to be implemented on the core architecture is the OFX Image Effect Plug-in API.

It is all specified using the 'C' programming language. C and C++ are the languages mainly used to write visual effects applications (the initial target for OFX plug-in APIs) and have a very wide adoption across most operating systems with many available compilers. By making the API C, rather than C++, you remove the whole set of problems around C++ symbol mangling between the host and plug-in.

APIs are defined in OFX by only a set of C header files and associated documentation. There are no binary libraries for a plug-in or host to link against.

Hosts rely on two symbols within a plug-in, all other communication is boot strapped from those two symbols. The plug-in has no symbolic dependencies from the host. This minimal symbolic dependency allows for run-time determination of what features to provide over the API, making implementation much more flexible and less prone to backwards compatibility problems.

Plug-ins, via the two exposed symbols, indicate the API they implement, the version of the API, their name, their version and their main entry point.

A host communicates with a plug-in via sending 'actions' to the plug-in's main entry function. Actions are C strings that indicate the specific operation to be carried out. They are associated with sets of properties, which allows the main entry function to behave as a generic function.

A plug-in communicates with a host by using sets of function pointers given it by the host. These sets of function pointers, known as 'suites', are named via a C string and a version number. They are returned on request from the host as pointers within a C struct.

Properties are typed value/name pairs that exist on the various OFX objects and are action argument values to the plug-in's main entry point. They are how a plug-in and host pass individual values back and forth to each other. The property suite, defined inside ofxProperty.h is used to do this.

1.1.2 OFX APIs

An OFX plug-in API is a named set of actions, properties and suites to perform some specific set of tasks. The first such API that has been defined on the OFX core is the OFX Image Effect API. The set of actions, properties and suites that constitute the API makes up the major part of this document.

Various suites and actions have been defined for the OFX image effect API, however many are actually quite generic and could be reused by other APIs. The property suite definitely has to be used by all other APIs, while the memory allocation suite, the parameter suite and several others would probably be useful for all other APIs. For example the parameter suite could be re-used to specify user visible parameters to the other APIs.

Several types are common to all OFX APIs, and as such are defined in ofxCore.h. Most objects passed back to a plug-in are generally specified by blind data handles, for example:

typedef struct OfxPropertySetStruct ***OfxPropertySetHandle**

Blind data structure to manipulate sets of properties through.

This allows for strong typing on functions but allows the implementation of the object to be hidden from the plug-in.

- *OfxStatus*

  Used to define a set of status codes indicating the success or failure of an action or suite function

- *OfxHost*

  A C struct that is used by a plug-in to get access to suites from a host and properties about the host

- **OfxStatus() OfxPluginEntryPoint (const char *action, const void *handle, OfxPropertySetHandle inArgs, OfxPropertySetHandle outArgs)**

  Entry point for plug-ins.

  - *action* - ASCII c string indicating which action to take
  - *instance* - object to which action should be applied, this will need to be cast to the appropriate blind data type depending on the *action*
  - *inData* - handle that contains action specific properties
  - *outData* - handle where the plug-in should set various action specific properties

  This is how the host generally communicates with a plug-in. Entry points are used to pass messages to various objects used within OFX. The main use is within the OfxPlugin struct.

  The exact set of actions is determined by the plug-in API that is being implemented, however all plug-ins can perform several actions. For the list of actions consult OFX Actions.

  A typedef for functions used as main entry points for a plug-in (and several other objects),

- *OfxPlugin*

  A C struct that a plug-in fills in to describe itself to a host.

Several general assumptions have been made about how hosts and plug-ins communicate, which specific APIs *are* allowed to break. The main is the distinction between…

- Descriptors

  Which hosts and plug-ins use to define the general behaviour of an object, e.g. the object used to specify what bit depths an Image Effect Plug-in is willing to accept,

- Instances

  Which hosts and plug-ins use to control the behaviour of a specific **live** object.

In most APIs descriptors are typically passed from a host to a plug-in during the `kOfxActionDescribe` action, whilst all other actions are passed an instance, e.g: the object passed to the `kOfxActionCreateInstance` action.

1.1.3 The OFX Image Effect API.

The OFX Image Effect Plug-in API is designed for image effect plug-ins for 2D visual effects. This includes such host applications as compositors, editors, rotoscoping tools and colour grading systems.

At heart the image effect API allows a host to send a plug-in a set of images, state the value of a set of parameters and get a resulting image back. However how it does this is somewhat complicated, as the plug-in and host have to negotiate what kind of images are handled, how they can be processed and much more.

1.2 The Generic Core API

This chapter describes how plug-ins are distributed and the core API for loading and identifying image effect plug-ins, and the methods of communications between plug-in and host.

1.2.1 OFX Include Files

The *C* include files that define an OFX API are all that are needed by a plug-in or host to implement the API. Most include files define a set of independent *suites* which are used by a plug-in to communicate with a host application.

There are two include files that are used with nearly every derived API. These are…

- ofxCore.h is used to define the basic communication mechanisms between a host and a plug-in. This includes the way in which a plug-in is defined to a host and how to bootstrap the two way communications. It also has several other basic action and property definitions.
- ofxProperty.h specifies the property suite, which is how a plug-in gets and sets values on various objects in a host application.

1.2.2 Identifying and Loading Plug-ins

Plug-ins must implement at least two, and normally three, exported functions for a host to identify the plug-ins and to initiate the bootstrapping of communication between the two.

OfxStatus **`OfxSetHost`**(const *OfxHost* *host)

First thing host should call.

This host call, added in 2020, is not specified in earlier implementation of the API. Therefore host must check if the plugin implemented it and not assume symbol exists. The order of calls is then: 1) OfxSetHost, 2) OfxGet-NumberOfPlugins, 3) OfxGetPlugin The host pointer is only assumed valid until OfxGetPlugin where it might get reset. Plug-in can return kOfxStatFailed to indicate it has nothing to do here, it's not for this Host and it should be skipped silently.

int **`OfxGetNumberOfPlugins`**(void)

Defines the number of plug-ins implemented inside a binary.

A host calls this to determine how many plug-ins there are inside a binary it has loaded. A function of this type must be implemented in and exported from each plug-in binary.

OfxPlugin ***`OfxGetPlugin`**(int nth)

Returns the 'nth' plug-in implemented inside a binary.

Returns a pointer to the 'nth' plug-in implemented in the binary. A function of this type must be implemented in and exported from each plug-in binary.

`OfxSetHost` is the very first function called by the host after the binary has been loaded, if it is implemented by the plugin. It passes an *The OfxHost Struct* struct to the plugin to enable the plugin to decide which effects to expose to the host. COMPAT: this call was introduced in 2020; some hosts and/or plugins may not implement it.

`OfxGetNumberOfPlugins` is the next function called by the host after the binary has been loaded and `OFXSetHost` has been called. The returned pointer to OfxGetPlugin and pointers in the struct do not need to be freed in any way by the host.

1.2.3 The Plug-in Main Entry Point And Actions

Actions are how a host communicates with a plug-in. They are in effect generic function calls. Actions are issued via a plug-in's ``mainEntry``function pointer found in its *OfxPlugin struct*. The function signature for the main entry point is

`OfxStatus() OfxPluginEntryPoint (const char *action, const void *handle, OfxPropertySetHandle inArgs, OfxPropertySetHandle outArgs)`

Entry point for plug-ins.

- *action* - ASCII c string indicating which action to take
- *instance* - object to which action should be applied, this will need to be cast to the appropriate blind data type depending on the *action*
- *inData* - handle that contains action specific properties
- *outData* - handle where the plug-in should set various action specific properties

This is how the host generally communicates with a plug-in. Entry points are used to pass messages to various objects used within OFX. The main use is within the OfxPlugin struct.

The exact set of actions is determined by the plug-in API that is being implemented, however all plug-ins can perform several actions. For the list of actions consult OFX Actions.

The *OfxStatus* value returned is dependent upon the action being called; however the value *`kOfxStatReplyDefault`* is returned if the plug-in does not trap the action.

The exact set of actions passed to a plug-in's entry point are dependent upon the API the plug-in implements. However, there exists a core set of generic actions that most APIs would use.

1.2.4 Suites

Suites are how a plug-in communicates back to the host. A suite is simply a set of function pointers in a C struct. The set of suites a host needs to implement is defined by the API being implemented. A suite is fetched from a host via the *`OfxHost::fetchSuite()`* function. This returns a pointer (cast to `void *`) to the named and versioned set of functions. By using this suite fetching mechanism, there is no symbolic dependency from the plug-in to the host, and APIs can be easily expandable without causing backwards compatibility issues.

If the host does not implement a requested suite, or the requested version of that suite, then it should return NULL.

1.2.5 Sequences of Operations Required to Load a Plug-in

The following sequence of operations needs to be performed by a host before it can start telling a plug-in what to do via its `mainEntry` function.

1. the binary containing the plug-in is loaded,
2. (if implemented by plugin and host): the host calls the plug-in's `OfxSetHost` function
3. the number of plug-ins is determined via the `OfxGetNumberOfPlugins` function,
4. for each plug-in defined in the binary
   1. `OfxGetPlugin` is called,
   2. the pluginApi and apiVersion of the returned OfxPlugin struct are examined,
   3. if the plug-in's API or its version are not supported, the plug-in is ignored and we skip to the next one,
   4. the plug-in's pointer is recorded in a plug-in cache,
   5. an appropriate OfxHost struct is passed to the plug-in via setHost in the returned OfxPlugin struct.

1.2.6 Who Owns The Data?

Objects are passed back and forth across the API, and in general, it is the thing that passes the data that is responsible for destroying it. For example the property set handle in the *OfxHost struct* is managed by the host.

There are a few explicit exceptions to this. For example, when an image effect asks for an image from a host it is passed back a property set handle which represents the image. That handle needs to later be disposed of by an effect, by an explicit function call back to the host. These few exceptions are documented with the suite functions that access the object.

Strings

A special case is made for strings. Strings are considered to be of two types, *value* strings and *label* strings. A label string is any string used by OFX to name a property or type. A value string is generally a string value of a property.

More specifically, a label string is a string passed across the API as one of the following…

- a property label (i.e: the char* property argument in the property suites)
- a string argument to a suite function which must be one of a set of predefined set of values e.g: paramType argument to OfxParameterSuiteV1::paramDefine , but not the name argument)

Label strings are considered to be static constant strings. When passed across the API the host/plug-in receiving the string neither needs to duplicate nor free the string, it can simply retain the original pointer passed over and use that in future, as it will not change. A host must be aware that when it unloads a plug-in all such pointers will be invalid, and be prepared to cope with such a situation.

A value string is a string passed across the API as one of the following…

- all value arguments to any of the property suite calls
- any other char* argument to any other function.

Value strings have no assumptions made about them. When one is passed across the API, the thing that passed the string retains ownership of it. The thing getting the string is not responsible for freeing that string. The scope of the string's validity is until the next OFX API function is called. For example, within a plugin

```
// pointer to store the returned value of the host name
char *returnedHostName;

// get the host name
propSuite->propGetString(hostHandle, kOfxPropName, 0, &returnedHostName);

// now make a copy of that before the next API call, as it may not be valid␣
↪after it
char *hostName = strdup(returnedHostName);

paramSuite->getParamValue(instance, "myParam", &value);
```

1.3 The OfxHost Struct

The OfxHost struct is how a host provides plug-ins with access to the various suites that make up the API they implement, as well as a host property set handle which a plug-in can ask questions of. The `setHost` function in the OfxPlugin struct is passed a pointer to an OfxHost as the first thing to boot-strapping plug-in/host communication.

The OfxHost contains two elements,

- host - a property set handle that holds a set of properties which describe the host for the plug-in's API
- fetchSuite - a function handle used to fetch function suites from the host that implement the plug-in's API

The host property set handle in the OfxHost is not global across all plug-ins defined in the binary. It is only applicable for the plug-in whose 'setHost' function was called. Use this handle to fetch things like host application names, host capabilities and so on. The set of properties on an OFX Image Effect host is found in the section *Properties on the Image Effect Host*

The `fetchSuite` function is how a plug-in gets a suite from the host. It asks for a suite by giving the C string corresponding to that suite and the version of that suite. The host will return a pointer to that suite, or NULL if it does not support it. Please note that a suite cannot be fetched until the very first action is called on the plug-in, which is the load action.

struct **`OfxHost`**

Generic host structure passed to OfxPlugin::setHost function.

This structure contains what is needed by a plug-in to bootstrap its connection to the host.

Public Members

OfxPropertySetHandle **`host`**

Global handle to the host. Extract relevant host properties from this. This pointer will be valid while the binary containing the plug-in is loaded.

const void *(***`fetchSuite`**)(*OfxPropertySetHandle* host, const char *suiteName, int suiteVersion)

The function which the plug-in uses to fetch suites from the host.

- *host* - the host the suite is being fetched from this *must* be the *host* member of the *OfxHost* struct containing fetchSuite.
- *suiteName* - ASCII string labelling the host supplied API

- *suiteVersion* - version of that suite to fetch

Any API fetched will be valid while the binary containing the plug-in is loaded.

Repeated calls to fetchSuite with the same parameters will return the same pointer.

returns

- NULL if the API is unknown (either the api or the version requested),
- pointer to the relevant API if it was found

1.4 The OfxPlugin Struct

This structure is returned by a plugin to identify itself to the host.

```
typedef struct OfxPlugin {
   const char        *pluginApi;
   int                apiVersion;
   const char        *pluginIdentifier;
   unsigned int       pluginVersionMajor;
   unsigned int       pluginVersionMinor;
   void               (*setHost)(OfxHost *host);
   OfxPluginEntryPoint *mainEntry;
} OfxPlugin;
```

pluginApi This C string tells the host what API the plug-in implements.

apiVersion This integer tells the host which version of its API the plug-in implements.

pluginIdentifier This is the globally unique name for the plug-in.

pluginVersionMajor Major version of this plug-in, this gets incremented whenever software is changed and breaks backwards compatibility.

pluginVersionMinor Minor version of this plug-in, this gets incremented when software is changed, but does not break backwards compatibility.

setHost Function used to set the host pointer (see below) which allows the plug-in to fetch suites associated with the API it implements.

mainEntry The plug-in function that takes messages from the host telling it to do things.

1.4.1 Interpreting the OfxPlugin Struct

When a host gets a pointer back from OfxGetPlugin, it examines the string pluginApi. This identifies what kind of plug-in it is. Currently there is only one publicly specified API that uses the OFX mechanism, this is `"OfxImageEffectPluginAPI"`, which is the image effect API being discussed by this book. More APIs may be created at a future date, for example “OfxImageImportPluginAPI”. Knowing the type of plug-in, the host then knows what suites and host handles are required for that plug-in and what functions the plug-in itself will have. The host passes a OfxHost structure appropriate to that plug-in via its `setHost` function. This allows for the same basic architecture to support different plug-in types trivially.

OFX explicitly versions plug-in APIs. By examining the apiVersion, the host knows exactly what set of functions the plug-in is going to supply and what version of what suites it will need to provide. This also allows plug-ins to implement several versions of themselves in the same binary, so it can take advantages of new features in a V2 API, but present a V1 plug-in to older hosts that only support V1.

If a host does not support the given plug-in type, or it does not support the given version it should simply ignore that plug-in.

A plug-in needs to uniquely identify itself to a host. This is the job of pluginIdentifier. This null terminated ASCII C string should be unique among all plug-ins, it is not necessarily meant to convey a sensible name to an end user. The recommended format is the reverse domain name format of the developer, for example "uk.co.thefoundry", followed by the developer's unique name for the plug-in. e.g. "uk.co.thefoundry.F_Kronos".

A plug-in (as opposed to the API it implements) is versioned with two separate integers in the OfxPlugin struct. They serve two separate functions and are,

- pluginVersionMajor flags the functionality contained within a plug-in. Incrementing this number means that you have broken backwards compatibility of the plug-in. More specifically, this means a setup from an earlier version, when loaded into this version, will not yield the same result.
- pluginVersionMinor flags the release of a plug-in that does not break backwards compatibility, but otherwise enhances that plug-in. For example, increment this when you have fixed a bug or made it faster.

If a host encounters multiple versions of the same plug-in it should,

- when creating a brand new instance, always use the version of a plug-in with the greatest major and minor version numbers,
- when loading a setup, always use the plug-in with the major version that matches the setup, but has the greatest minor number.

As a more concrete example of versioning: the plug-in identified by "org.wibble:Fred" is initially released as 1.0, However a few months later, wibble.org figure out how to make it faster and release it as 1.1. A year later, Fred can now do automatically what a user once needed to set up five parameters to do, thus making it much simpler to use. However this breaks backwards compatibility as the effect can no longer produce the same output as before, so wibble.org then release this as v2.0.

A user's host might now have three versions of the Fred plug-in on it, v1.0, v1.1 and v2.0. When creating a new instance of the plug-in, the host should always use v2.0. When loading an old project which has a setup from a v1.x plug-in, it should always use the latest, in this case being v1.1.

Note that plug-ins can change the set of parameters between minor version releases. If a plug-in does so, it should do so in a backwards compatible manner, such that the default value of any new parameter would yield the same results as previously. See the chapter below about parameters.

1.5 Packaging OFX Plug-ins

Where a host application chooses to search for OFX plug-ins, what binary format they are in and any directory hierarchy is entirely up to it. However, it is strongly recommended that the following scheme be followed.

1.5.1 Binary Types

Plug-ins should be distributed in the following formats, depending on the host operating system. . . .

- Microsoft Windows, as ".dll" dynamically linked libraries,
- Apple OSX, as binary bundles,
- LINUX (and other Unix variants), as native dynamic shared objects.

1.5.2 Installation Directory Hierarchy

Each plug-in binary is distributed as a Mac OS X package style directory hierarchy. Note that there are two distinct meanings of 'bundle', one referring to a binary file format, the other to a directory hierarchy used to distribute software. We are distributing binaries in a bundle package, and in the case of OSX, the binary is a binary bundle. All the binaries must end with `".ofx"`, regardless of the host operating system.

The directory hierarchy is as follows.....

- NAME.ofx.bundle
  - Contents
    - Info.plist
    - Resources
      - NAME.xml
      - EFFECT_A.png
      - EFFECT_A.svg
      - EFFECT_B.png
      - EFFECT_B.svg
      - ...
    - ARCHITECTURE_A
      - NAME.ofx
    - ARCHITECTURE_B
      - NAME.ofx
    - ...
    - ARCHITECTURE_N
      - NAME.ofx

Where...

- Info.plist is relevant for OSX only and needs to be filled in appropriately,
- NAME is the file name you want the installed plug-in to be identified by,
- EFFECT_N.png - is an optional PNG image file image to use as an icon for the effect in the plug-in binary which has a matching `pluginIdentifier` field in the `OfxPlugin` struct,
- EFFECT_N.svg - is an optional scalable vector graphic file to use as an icon for the plug-in in the binary which has a matching `pluginIdentifier` field in the `OfxPlugin` struct,
- ARCHITECTURE is the specific operating system architecture the plug-in was built for, these are currently...
  - `MacOS` - for Apple Macintosh OS X 32 bit and/or universal binaries
  - `MacOS-x86-64` - for Apple Macintosh OS X, specifically on intel x86 CPUs running AMD's 64 bit extensions. 64 bit host applications should check this first, and if it doesn't exist or is empty, fall back to "MacOS" looking for a universal binary.
  - `Win32` - for Microsoft Windows (compiled 32 bit)
  - `Win64` - for Microsoft Windows (compiled 64 bit)
  - `IRIX` - for SGI IRIX plug-ins (compiled 32 bit)

  - `IRIX64` - for SGI IRIX plug-ins (compiled 64 bit)
  - `Linux-x86` - for Linux on x86 CPUs (compiled 32 bit)
  - `Linux-x86-64` - for Linux on x86 CPUs running AMD's 64 bit extensions

Note that not all the above architectures need be supported, at least one.

This structure is necessary on OS X, but it also gives a nice skeleton to hang all other operating systems from in a single install, as well as a clean place to put resources.

The `Info.plist` is specific to Apple and you should consult the Apple developer's website for more details. It should contain the following keys…

- `CFBundleExecutable` - the name of the binary bundle in the MacOS directory
- `CFBundlePackageType` - to be `BNDL`
- `CFBundleInfoDictionaryVersion`
- `CFBundleVersion`
- `CFBundleDevelopmentRegion`

1.5.3 Installation Location

plug-ins are searched for in a variety of locations, both default and user specified. All such directories are examined for plug-in bundles and sub directories are also recursively examined.

A list of directories is supplied in the "OFX_PLUGIN_PATH" environment variable, these are examined, first to last, for plug-ins, then the default location is examined.

On Microsoft Windows machines, the plug-ins are searched for in:

1. the ';'-separated directory list specified by the environment variable "OFX_PLUGIN_PATH"
2. the directory returned by `getStdOFXPluginPath` in the following code snippet:

```
#include "shlobj.h"
#include "tchar.h"
const TCHAR *getStdOFXPluginPath(void)
{
  static TCHAR buffer[MAX_PATH];
  static int gotIt = 0;
  if(!gotIt) {
    gotIt = 1;
    SHGetFolderPath(NULL, CSIDL_PROGRAM_FILES_COMMON, NULL, SHGFP_TYPE_CURRENT,␣
↪buffer);
    _tcscat(buffer, __T("\\OFX\\Plugins"));
  }
  return buffer;
}
```

3. the directory `C:\Program Files\Common Files\OFX\Plugins`. This location is deprecated, and it is returned by the code snippet above on English language systems. However it should still be examined by hosts for backwards compatibility.

On Apple OSX machines, the plug-ins are searched for in:

1. the ';'-separated directory list specified by the environment variable "OFX_PLUGIN_PATH"
2. the directory `/Library/OFX/Plugins`

On UNIX, Linux and other UNIX like operating systems, the plug-ins are searched for in:

1. the ':'-separated directory specified by the environment variable "OFX_PLUGIN_PATH"
2. the directory `/usr/OFX/Plugins`

Any bundle or sub-directory name starting with the character '@' is to be ignored. Such directories or bundles must be skipped by the host.

1.5.4 Plug-in Icons

Some hosts may wish to display an icon associated with the effects in a plug-in binary on their interfaces. Any such icon must be in the Portable Network Graphics format (see http://www.libpng.org/) and must contain 32 bits of colour, including an alpha channel. Ideally it should be at least 128x128 pixels.

Note that a single plug-in binary may define more than one effect, when *OfxGetNumberOfPlugins* returns a value greater than 1. These icons are specific to each effect within the plug-in, and are named according to what is returned from *OfxGetPlugin*.

Host applications should dynamically resize the icon to fit their preferred icon size. The icon should not have its aspect changed, rather the host should fill with some appropriate colour any blank areas due to aspect mismatches.

Ideally plug-in developers should not render the plug-in or effect's name into the icon, as this may be changed by the resource file, especially for internationalisation purposes. Hosts should thus present the plug-in and/or effect's name next to the icon in some way.

The icon file must be named as the corresponding `pluginIdentifier` field from the `OfxPlugin`, postpended with '.png' and be placed in the resources sub-directory.

Some hosts may use a scalable vector icon if provided; it should be in SVG format and be named and located just like the `.png` icon but with a `.svg` suffix.

1.5.5 Externally Specified Resources

Some plug-ins may supply an externally specified resource file for particular hosts. Typically this is for tasks such as internationalising interfaces, tweaking user interfaces for specific hosts, and so on. These are XML files and have DTD associated with the specific API, for example OFX Image Effect DTD is found in `ofx.dtd`.

The XML resource file is installed in the `Resources` subdirectory of the bundle hierarchy. Its name should be `NAME.xml`, where name is the base name of the bundle folder and the effect binary.

Plug-ins are free to include other resources in the `Resources` subdirectory.

1.6 The Image Effect API

1.6.1 Introduction

In general, image effects plug-ins take zero or more input clips and produce an output clip. So far so simple, however there are many devils hiding in the details. Several supporting suites are required from the host and the plug-in needs to respond to a range of actions to work correctly. How an effect is intended to be used also complicates the issue, forcing sets of behaviours depending on the context of an effect.

Plug-ins that implement the image effect API set the `pluginApi` member of the *OfxPlugin struct* returned by the global *OfxGetPlugin* to be:

`kOfxImageEffectPluginApi`

String used to label OFX Image Effect Plug-ins.

Set the pluginApi member of the OfxPluginHeader inside any OfxImageEffectPluginStruct to be this so that the host knows the plugin is an image effect.

The current version of the API is 1. This is enough to label the plug-in as an image effect plug-in.

1.6.2 Image Effect API Header Files

The header files used to define the OFX Image Effect API are…

- ofxCore.h Provides the core definitions of the general OFX architecture that allow the bootstrapping of specific APIs, as well as several core actions,
- **ofxProperty.h** Provides generic property fetching suite used to get and set values about objects in the API,
- ofxParam.h Provides the suite for defining user visible parameters to an effect
- ofxMultiThread.h Provides the suite for basic multi-threading capabilities
- ofxInteract.h Provides the suite that allows a plug-in to use OpenGL to draw their own interactive GUI tools
- ofxKeySyms.h Provides key symbols used by 'Interacts' to represent keyboard events
- ofxMemory.h Provides a simple memory allocation suite,
- ofxMessage.h Provides a simple messaging suite to communicate with an end user
- ofxImageEffect.h Defines a suite and set of actions that draws all the above together to create an visual effect plug-in.
- ofxDrawSuite.h Provides an optional suite that allows a plug-in to draw their own interactive GUI tools without using OpenGL

These contain the suite definitions, property definitions and action definitions that are used by the API.

1.6.3 Actions Used by the API

All image effect plug-ins have a main entry point. This is used to trap all the standard actions used to drive the plug-in. They can also have other optional entry points that allow the plug-in to create custom user interface widgets. These *interact* entry points are specified during the two description actions.

The following actions can be passed to a plug-in's main entry point…

- The Generic Load Action called just after a plug-in is first loaded,
- The Generic Unload Action called just before a plug-in is unloaded,
- The Generic Describe Action called to describe a plug-in's behaviour to a host,
- The Generic Create Instance Action called just after an instance is created,
- The Generic Destroy Instance Action called just before an instance is destroyed,
- The Generic Begin/End Instance Changed Actions , a pair of actions used to bracket a set of Instance Changed actions,
- The Generic Instance Changed Action an action used to indicate that a value has changed in a plug-in instance,
- The Generic Purge Caches Action called to have the plug-in delete any temporary private data caches it may have,

- The Sync Private Data Action called to have the plug-in sync any private state data back to its data set,
- The Generic Begin/End Instance Edit Actions a pair of calls that are used to bracket the fact that a user interface has been opened on an instance and it is being edited,
- The Begin Sequence Render Action where a plug-in is told that it is about to render a sequence of images,
- The Render Action where a plug-in is told that it is to render an output image,
- The End Sequence Render Action where a plug-in is told it has finished rendering a sequence of images,
- The Describe In Context Action used to have a plug-in describe itself in a specific context,
- The Get Region of Definition Action where an instance gets to state how big an image it can create,
- The Get Regions Of Interest Action where an instance gets to state how much of its input images it needs to create a give output image,
- The Get Frames Needed Action where an instance gets to state how many frames of input it needs on a given clip to generate a single frame of output,
- The Is Identity Action where an instance gets to state that its current state does not affect its inputs, so that the output can be directly copied from an input clip,
- The Get Clip Preferences Action where an instance gets to state what data and pixel types it wants on its inputs and will generate on its outputs,
- The Get Time Domain Action where a plug-in gets to state how many frames of data it can generate.

1.6.4 Main Objects Used by the API

The image effect API uses a variety of different objects. Some are defined via blind data handles, others via property sets, and some by a combination of the two. These objects are…

- Host Descriptor - a descriptor object used by a host to describe its behaviour to a plug-in,
- Image Effect Descriptor - a descriptor object used by a plug-in to describe its behaviour to a host,
- Image Effect Instance - an instance object maintaining state about an image effect,
- Clip Descriptor - a descriptor object for a sequence of images used as input or output a plug-in may use,
- Clip Instance - a instance object maintaining state about a sequence of images used as input or output to an effect instance,
- Parameter Descriptor - a descriptor object used to specify a user visible parameter in an effect descriptor,
- Parameter Instance - an instance object that maintains state about a user visible parameter in an effect instance,
- Parameter Set Descriptor - a descriptor object used to specify a set of user visible parameters in an effect descriptor,
- Parameter Set Instance - an instance object that maintains state about a set of user visible parameters in an effect instance,
- Image Instance - a instance object that maintains state about a 2D image being passed to an effect instance.
- Interact Descriptor - which describes a custom openGL user interface, for example an overlay over the inputs to an image effect. These have a separate entry point to an image effect.
- Interact Instance - which holds the state on a custom openGL user interface. These have a separate entry point to an image effect.

Host Descriptors

The host descriptor is represented by the properties found on the host property set handle in the *OfxHost struct*. The complete set of read only properties are found in the section Properties on the Image Effect Host.

These sets of properties are there to describe the capabilities of the host to a plug-in, thus giving a plug-in the ability to modify its behaviour depending on the capabilities of the host.

A host descriptor is valid while a plug-in is loaded.

Effects

An effect is an object in the OFX Image Effect API that represents an image processing plug-in. It has associated with it a set of properties, a set of image clips and a set of parameters. These component objects of an effect are defined and used by an effect to do whatever processing it needs to. A handle to an image effect (instance or descriptor) is passed into a plug-in's *main entry point* *handle* argument:

typedef struct OfxImageEffectStruct ***`OfxImageEffectHandle`**

Blind declaration of an OFX image effect.

The functions that directly manipulate an image effect handle are specified in the *`OfxImageEffectSuiteV1`* found in the header file ofxImageEffect.h.

Effect Descriptors

An effect descriptor is an object of type *`OfxImageEffectHandle`* passed into an effect's *main entry point* `handle` argument. The two actions it is passed to are:

- *`kOfxActionDescribe`*
- *`kOfxImageEffectActionDescribeInContext`*

A effect descriptor does not refer to a 'live' effect, it is a handle which the effect uses to describe itself back to the host. It does this by setting a variety of properties on an associated property handle, and specifying a variety of objects (such as clips and parameters) using functions in the available suites.

Once described, a host should cache away the description in some manner so that when an instance is made, it simply looks at the description and creates the necessary objects needed by that instance. This stops the overhead of having every instance be forced to describe itself over the API.

Effect descriptors are only valid in a effect for the duration of the instance they were passed into.

The properties on an effect descriptor can be found in the section Properties on an Effect Descriptor.

Effect Instances

A effect instance is an object of type *`OfxImageEffectHandle`* passed into an effect's *main entry point* `handle` argument. The `handle` argument should be statically cast to this type. It is passed into all actions of an image effect that a descriptor is not passed into.

The effect instance represents a 'live' instance of an effect. Because an effect has previously been described, via a effect descriptor, an instance does not have to respecify the parameters, clips and properties that it needs. These means, that when an instance is passed to an effect, all the objects previously described will have been created.

Generally multiple instances of an effect can be in existence at the same time, each with a different set of parameters, clips and properties.

Effect instances are valid between the calls to *kOfxActionCreateInstance* and *kOfxActionDestroyInstance*, for which it is passed as the `handle` argument.

The properties on an effect instance can be found in the section Properties on an Effect Instance.

Clips

A clip is a sequential set of images attached to an effect. They are used to fetch images from a host and to specify how a plug-in wishes to manage the sequence.

Clip Descriptors

Clip descriptors are returned by the *OfxImageEffectSuiteV1::clipDefine()* function. They are used during the *kOfxActionDescribe* action by an effect to indicate the presence of an input or output clip and how that clip behaves.

A clip descriptor is only valid for the duration of the action it was created in.

The properties on a clip descriptor can be found in the section Properties on a Clip Descriptor.

Clip Instances

typedef struct OfxImageClipStruct ***OfxImageClipHandle**

Blind declaration of an OFX image effect.

Clip instances are returned by the *OfxImageEffectSuiteV1::clipGetHandle()* function. They are are used to access images and and manipulate properties on an effect instance's input and output clips. A variety of functions in the *OfxImageEffectSuiteV1* are used to manipulate them.

A clip instance is valid while the related effect instance is valid.

The properties on a clip instance can be found in the section Properties on a Clip Instance.

Parameters

Parameters are user visible objects that an effect uses to specify its state, for example a floating point value used to control the blur size in a blur effect. Parameters (both descriptors and instances) are represented as blind data handles of type:

typedef struct OfxParamStruct ***OfxParamHandle**

Blind declaration of an OFX param.

Parameter sets are the collection of parameters that an effect has associated with it. They are represented by the type `OfxParamSetHandle`. The contents of an effect's parameter set are defined during the :c:macro:`kOfxImageEffectActionDescribeInContext action. Parameters cannot be dynamically added to, or deleted from an effect instance.

Parameters can be of a wide range of types, each of which have their own unique capabilities and property sets. For example a colour parameter differs from a boolean parameter.

Parameters and parameter sets are manipulated via the calls and properties in the *OfxParameterSuiteV1* specified in ofxParam.h. The properties on parameter instances and descriptors can be found in the section Properties on Parameter Descriptors and Instances.

Parameter Set Descriptors

Parameter set descriptors are returned by the :cpp:func`OfxImageEffectSuiteV1::getParamSet` function. This returns a handle associated with an image effect descriptor which can be used by the parameter suite routines to create and describe parameters to a host.

A parameter set descriptor is valid for the duration of the `kOfxImageEffectActionDescribeInContext` action in which it is fetched.

Parameter Descriptors

Parameter descriptors are returned by the `OfxParameterSuiteV1::paramDefine()` function. They are used to define the existence of a parameter to the host, and to set the various attributes of that parameter. Later, when an effect instance is created, an instance of the described parameter will also be created.

A parameter descriptor is valid for the duration of the `kOfxImageEffectActionDescribeInContext` action in which it is created.

Parameter Set Instances

Parameter set instances are returned by the `OfxImageEffectSuiteV1::getParamSet()` function. This returns a handle associated with an image effect instance which can be used by the parameter suite routines to fetch and describe parameters to a host.

A parameter set handle instance is valid while the associated effect instance remains valid.

Parameter Instances

Parameter instances are returned by the `OfxParameterSuiteV1::paramGetHandle()` function. This function fetches a previously described parameter back from the parameter set. The handle can then be passed back to the various functions in the `OfxParameterSuite1V` to manipulate it.

A parameter instance handle remains valid while the associated effect instance remains valid.

Image Instances

An image instance is an object returned by the `OfxImageEffectSuiteV1::clipGetImage()` function. This fetches an image out of a clip and returns it as a property set to the plugin. The image can be accessed by looking up the property values in that set, which includes the data pointer to the image.

An image instance is valid until the effect calls `OfxImageEffectSuiteV1::clipReleaseImage()` on the property handle. The effect *must* release all fetched images before it returns from the action.

The set of properties that make up an image can be found in the section Properties on an Image.

Interacts

An interact is an OFX object that is used to draw custom user interface elements, for example overlays on top of a host's image viewer or custom parameter widgets. Interacts have their own *main entry point*, which is separate to the effect's main entry point. Typically an interact's main entry point is specified as a pointer property on an OFX object, for example the `kOfxImageEffectPluginPropOverlayInteractV1` property on an effect descriptor.

The functions that directly manipulate interacts are in the `OfxInteractSuiteV1` found in the header file ofxInteract.h , as well as the properties and specific actions that apply to interacts.

Interact Descriptors

Interact descriptors are blind handles passed to the `kOfxActionDescribeInteract` sent to an interact's separate main entry point. They should be cast to the type `OfxInteractHandle`.

The properties found on a descriptor are found in section Properties on Interact Descriptors.

Interact Instances

Interact instances are blind handles passed to all actions but the `kOfxActionDescribe` sent to an interact's separate main entry point. They should be cast to the type

typedef struct OfxInteract ***`OfxInteractHandle`**

Blind declaration of an OFX interactive gui.

The properties found on an instance are found in section Properties on Interact Instance.

1.7 Image Processing Architectures

OFX supports a range of image processing architectures. The simpler ones being special cases of the most complex one. Levels of support, in both plug-in and host, are signalled by setting appropriate properties in the plugin and host.

This chapter describes the most general architecture that OFX can support, with simpler cases just being specialisations of the general case.

1.7.1 The Image Plane

At it's most generalised, OFX allows for a complex imaging architecture based around an infinite 2D plane on which we are filling in pixels.

Firstly, there is some subsection of this infinite plane that the user wants to be the end result of their work, call this the project extent. The project extent is always rooted, on its bottom left, at the origin of the image plane. The project extent defines the upper right hand corner of the project window. For example a PAL sized project spans (0, 0) to (768, 576) on the image plane.

We define an image effect as something that can fill in a rectangle of pixels in this infinite plane, possibly using images defined at other locations on this image plane.

1.7.2 Regions of Definition

An effect has a **Region of Definition** (RoD), this is is the maximum area of the plane that the effect can fill in. for example: a 'read source media' effect would only be able to fill an area as big as it's source media. An effect's RoD may need to be based on the RoD of its inputs, for example: the RoD of a contrast/brightness colour corrector would generally be the RoD of it's input, while the RoD of a rotation effect would be bigger than that of it's input image.

The purpose of the *kOfxImageEffectActionGetRegionOfDefinition* action is for the host to ask an effect what its region of definition is. An effect calculates this by looking at its input clips and the values of its current parameters.

Hosts are not obliged to render all an effects RoD, as it may have fixed frame sizes, or any number of other issues.

Infinite RoDs

Infinite RoDs are used to indicate an effect can fill pixels in anywhere on the image plane it is asked to. For example a no-input noise generator that generates random colours on a per pixel basis. An infinite RoD is flagged by setting the minimums to be:

kOfxFlagInfiniteMin
Used to flag infinite rects. Set minimums to this to indicate infinite.

This is effectively INT_MIN

and the maxmimums to be:

kOfxFlagInfiniteMax
Used to flag infinite rects. Set minimums to this to indicate infinite.

This is effectively INT_MAX.

for both double and integer rects. Hosts and plug-ins need to be infinite RoD aware. Hosts need to clip such RoDs to an appropriate rectangle, typically the project extent. Plug-ins need to check for infinite RoDs when asking input clips for them and to pass them through unless they explicitly clamp them. To indicate an infinite RoD set it as indicated in the following code snippet.

```
outputRoD.x1 = kOfxFlagInfiniteMin;
outputRoD.y1 = kOfxFlagInfiniteMin;
outputRoD.x2 = kOfxFlagInfiniteMax;
outputRoD.y2 = kOfxFlagInfiniteMax;
```

1.7.3 Regions Of Interest

An effect will be asked to fill in some region of this infinite plane. The section it is being asked to fill in is called the **Region of Interest** (RoI).

Before an effect has been asked to process a given RoI, it will be asked to specify the area of each input clip it will need to process that area. For example: a simple colour correction effect only needs as much input as it does output, while a blur will need an area that is larger than the specified RoI by a border of the same width as the blur radius.

The purpose of the *kOfxImageEffectActionGetRegionsOfInterest* action is for the host to ask an effect what areas it needs from each input clip, to render a specific output region. An effect needs to examine its set of parameters and the region it has been asked to render to determine how much of *each* input clip it needs.

1.7.4 Tiled Rendering

Tiling is the ability of an effect to manage images that are less than full frame (or in our current nomenclature, less than the full Region of Definition). By tiling the images it renders, a host will render an effect in several passes, say by doing the bottom half, then the top half.

Hosts may tile rendering for a variety of reasons. Usually it is in an attempt to reduce memory demands or to distribute rendering of an effect to several different CPUs or computers.

Effects that in effect only perform per pixel calculations (for example a simple colour gain effect) tile very easily. However in the most general case for effects, tiling may be self defeating, as an effect, in order to render a tile, may need significantly more from its input clips than the tile in question. For example, an effect that performs an 2D transform on its input image, may need to sample all that image even when rendering a very small tile on output, as the input image may have been scaled down so that it only covers a few pixels on output.

1.7.5 Tree Based Architectures

The most general compositing hosts allow images to be of any size at any location on our image plane. They also plumb the output of effects into other effects, to create effect trees. When evaluating this tree of effects, a general host will want to render the minimum number of pixels it needs to fill in the final desired image. Typically the top level of this compositing tree is being rendered at a certain project size, for example PAL SD, 2K film and so on. This is where the RoD/RoI calls come in handy.

The host asks the top effect how much picture information it can produce, which in turn asks effects below it their RoDs and so on until leaf effects are reached, which report back up the tree until the top effect calculates its RoD and reports back to the host. The host typically clips that RoD to its project size.

Having determined in this way the window it wants rendered at the top effect, the host asks the top node the regions of interest on each of it's inputs. This again propagates down the effect tree until leaf nodes are encountered. These regions of interest are cached at effect for later use.

At this point the host can start rendering, from the bottom of the tree upwards, by asking each effect to fill in the region of interest that was previously specified in the RoI walk. These regions are then passed to the next level up to render and so on.

Another complication is tiling. If a host tiles, it will need to walk the tree and perform the RoI calculation for each tile that it renders.

The details may differ on specific hosts, but this is more or less the most generic way compositing hosts currently work.

1.7.6 Simpler Architectures

The above architecture is quite complex, as the inputs supplied can lie anywhere on the image plane, as can the output, and they can be subsections of the 'complete' image. Not all hosts work in this way, generally it is only the more advance compositing systems working on large resolution images.

Some other systems allow for images to be anywhere on the image plane, but always pass around full RoD images, never tiles.

The simplest systems, don't have any of of the above complexity. The RoDs, RoIs, images and project sizes in such systems are exactly the same, always. Often these are editing, as opposed to compositing, systems.

Similarly, some plugin effects cannot handle sub RoD images, or even images not rooted at the origin.

The OFX architecture is meant to support all of them. Assuming a plugin supports the most general architecture, it will trivially run on hosts with simpler architectures. However, if a plugin does not support tiled, or arbitrarily positioned images, they may not run cleanly on hosts that expect them to do so.

To this end, two properties are provided that flag the capabilities of a plugin or host…

- *kOfxImageEffectPropSupportsMultiResolution* which indicates support for images of differing sizes not centred on the origin,
- *kOfxImageEffectPropSupportsTiles* which indicates support for images that contain less than full frame pixel data

A plug-in should flag these appropriately, so that hosts know how to deal with the effect. A host can either choose to refuse to load a plugin, or, preferentially, pad images with an appropriate amount of black/transparent pixels to enable them to work.

The *kOfxImageEffectActionGetRegionsOfInterest* is redundant for plugins that do not support tiled rendering, as the plugin is asking that it be given the full Region of Definition of all its inputs. A host may have difficulty doing this (for example with an input that is attached to an effect that can create infinite images such as a random noise generator), if so, it should clamp images to some a size in some manner.

The RoD/RoI actions are potentially redundant on simpler hosts. For example fixed frame size hosts. If a host has no need to call these actions, it simply should not.

1.8 Image Effect Contexts

How an image effect is used by an end user affects how it should interact with a host application. For example an effect that is to be used as a transition between two clips works differently to an effect that is a simple filter. One must have two inputs and know how much to mix between the two input clips, the other has fewer constraints on it. Within OFX we have standardised several different uses and have called them *contexts*.

More specifically, a context mandates certain behaviours from an effect when it is described or instantiated in that context. The major issue is the number of input clips it takes, and how it can interact with those input clips.

All OFX contexts have a single output clip and zero or more input clips. The current contexts defined in OFX are:

- `kOfxImageEffectContextGenerator`

  No compulsory input clips used by a host to create imagery from scratch, e.g: a noise generator

- `kOfxImageEffectContextFilter`

  A single compulsory input clip. A traditional 'filter effect' that transforms a single input in some way,
  e.g: a simple blur

- `kOfxImageEffectContextTransition`

  Two compulsory input clips and a compulsory 'Transition' double parameter Used to perform transitions between clips, typically in editing

  applications, eg: a cross dissolve,

- `kOfxImageEffectContextPaint`

  Two compulsory input clips, one image to paint onto, the other a mask to control where the effect happens Used by hosts to use an effect under a paint brush

- `kOfxImageEffectContextRetimer`

  A single compulsory input clip, and a compulsory 'SourceTime' double parameter Used by a host to change the playback speed of a clip,

- `kOfxImageEffectContextGeneral`

  An arbitrary number of inputs, generally used in a 'tree' compositing environment, a catch all context.

A host or plug-in need not support all contexts. For example a host that does not have any paint facility within it should not need to support the paint context, or a simple blur effect need not support the retimer context.

An effect may say that it can be used in more than one context, for example a blur effect that acts as a filter, with a single input to blur, and a general effect, with an input to blur and an optional input to act as a mask to attenuate the blur. In such cases a host should choose the most appropriate context for the way that host's architecture. With our blur example, a tree based compositing host should simply ignore the filter context and always use it in the general context.

Plugins and hosts inform each other what contexts they work in via the multidimensional *kOfxImageEffectPropSupportedContexts* property.

A host indicates which contexts it supports by setting the *kOfxImageEffectPropSupportedContexts* property in the global host descriptor. A plugin indicates which contexts it supports by setting this on the effect descriptor passed to the *kOfxActionDescribe* action.

Because a plugin can work in different ways, it needs the ability to describe itself to the host in different ways. This is the purpose of the *kOfxImageEffectActionDescribeInContext* action. This action is called once for each context that the effect supports, and the effect gets to describe the input clips and parameters appropriate to that context. This means that an effect can have different sets of parameters and clips in different contexts, though it will most likely have a core set of parameters that it uses in all contexts. From our blur example, both the filter and general contexts would have a 'blur radius' parameter, but the general context might have an 'invert matte' parameter.

During the *kOfxImageEffectActionDescribeInContext* action, an effect must describe all clips and parameters that it intends to use. This includes the mandated clips and parameters for that context.

A plugin instance is created in a specific contex which will not changed over the lifetime of that instance. The context can be retrieved from the instance via the *kOfxImageEffectPropContext* property on the instance handle.

1.8.1 The Generator Context

A generator context is for cases where a plugin can create images without any input clips, eg: a colour bar generator.

In this context, a plugin has the following mandated clips,

- an output clip named *Output*

Any input clips that are specified must be optional.

A host is responsible for setting the initial preferences of the output clip, it must do this in a manner that is transparent to the plugin. So the pixel depths, components, fielding, frame rate and pixel aspect ratio are under the control of the host. How it arrives at these is a matter for the host, but as a plugin specifies what components it can produce on output, as well as the pixel depths it supports, the host must choose one of these.

Generators still have Regions of Definition. This should generally be,

- based on the project size eg: an effect that renders a 3D sky simulation,
- based on parameter settings eg: an effect that renders a circle in an arbitrary location,
- infinite, which implies the effect can generate output anywhere on the image plane.

The pixel preferences action is constrained in this context by the following,

- a plugin cannot change the component type of the *Output* clip,

1.8.2 The Filter Context

A filter effect is the ordinary way most effects are used with a single input. They allow track or layer based hosts that cannot present extra input to use an effect.

In this context, a plugin has the following mandated objects…

- an input clip named *Source*
- an output clip named *Output*

Other input clips may be described, which must all be optional. However there is no way to guarantee that all hosts will be able to wire in such clips, so it is suggested that in cases where effects can take single or multiple inputs, they expose themselves in the filter context with a single input and the general context with multiple inputs.

The pixel preferences action is constrained in this context by the following,

- a plugin cannot change the component type of the *Output* clip, it will always be the same as the *Source* clip,

1.8.3 The Transition Context

Transitions are effects that blend from one clip to another over time, eg: a wipe or a cross dissolve.

In this context, a plugin has the following mandated objects…

- an input clip names 'SourceFrom'
- an input clip names 'SourceTo'
- an output clip named *Output*
- a single double parameter called 'Transition' (see Mandated Parameters)

Any other input clips that are specified must be optional. Though it is suggested for simplicity's sake that only the two mandated clips be used.

The 'Transition' parameter cannot be labelled, positioned or controlled by the plug-in in anyway, it can only have it's value read, which will have a number returned between the value of 0 and 1. This number indicates how far through the transition the effect is, at 0 it should output all of 'SourceFrom', at 1 it should output all of 'SourceTo', in the middle some appropriate blend.

The pixel preferences action is constrained in this context by the following,

- the component types of the "SourceFrom", "SourceTo" and *Output* clips will always be the same,
- the pixel depths of the "SourceFrom", "SourceTo" and *Output* clips will always be the same,
- a plugin cannot change any of the pixel preferences of any of the clips.

1.8.4 The Paint Context

Paint effects are effects used inside digital painting system, where the effect is limited to a small area of the source image via a masking image. Perhaps 'brush' would have been a better choice for the name of the context.

In this context, a plugin has the following mandated objects…

- an input clip names *Source*,
- an input clip names *Brush*, the only component type it supports is 'alpha',
- an output clip named *Output*.

Any other input clips that are specified must be optional.

The masking images consists of pixels from 0 to the white point of the pixel depth. Where the mask is zero the effect should not occur, where the effect is whitepoint the effect should be ‘full on’, where it is grey the effect should blend with the source in some manner.

The masking image may be smaller than the source image, even if the effect states that it cannot support multi-resolution images.

The pixel preferences action is constrained in this context by the following,

- the pixel depths of the *Source*, *Brush* and *Output* clips will always be the same,
- the component type of *Source* and *Output* will always be the same,
- a plugin cannot change any of the pixel preferences of any of the clips.

1.8.5 The Retimer Context

The retimer context is for effects that change the length of a clip by interpolating frames from the source clip to create an inbetween output frame.

In this context, a plugin has the following mandated objects…

- an input clip names *Source*
- an output clip named *Output*
- a 1D double parameter named ‘SourceTime’ (see Mandated Parameters)

Any other input clips that are specified must be optional.

The ‘SourceTime’ parameter cannot be labelled, positioned or controlled by the plug-in in anyway, it can only have it’s value read. Its value is how the source time to maps to the output time. So if the output time is ‘3’ and the ‘SourceTime’ parameter returns 8.5 at this time, the resulting image should be an interpolated between source frames 8 and 9.

The pixel preferences action is constrained in this context by the following,

- the pixel depths of the *Source* and *Output* clips will always be the same,
- the component type of *Source* and *Output* will always be the same,
- a plugin cannot change any of the pixel preferences of any of the clips.

1.8.6 The General Context

The general context is to some extent a catch all context, but is generally how a ‘tree’ effect should be instantiated. It has no constraints on its input clips, nor on the pixel preferences actions.

In this context, has the following mandated objects…

- an output clip named *Output*

1.8.7 Parameters Mandated In A Context

The retimer and transition context both mandate a parameter be declared, the double params 'SourceTime' and 'Transition'. The purpose of these parameters is for the host to communicate with the plug-in, they are *not* meant to be treated as normal parameters, exposed on the user plug-in's user interface.

For example, the purpose of a transition effect is to dissolve in some interesting way between two separate clips, under control of the host application. Typically this is done on systems that edit. The mandated 'Transition' double pseudo-parameter is not a normal one exposed on the plug-in UI, rather it is the way the host indicates how far through the transition the effect is. For example, think about two clips on a time line based editor with a transition between them, the host would set the value value of the 'Transition' parameter implicitly by how far the frame being rendered is from the start of the transition, something along the lines of…

```
Transition = (currrentFrame - startOfTransition)/lengthOfTransition;
```

This means that the host is completely responsible for any user interface for that parameter, either implicit (as in the above editing example) or explicit (with a curve).

Similarly with the 'SourceTime' double parameter in the retimer context. It is up to the host to provide a UI for this, either implicitly (say by stretching a clip's length on the time line) or via an explicit curve. Note that the host is not limited to using a UI that exposes the 'SourceTime' as a curve, alternately it could present a 'speed' parameter, and integrate that to derive a value for 'SourceTime'.

1.9 Thread and Recursion Safety

Hosts are generally multi-threaded, those with a GUI will most likely have an interactive thread and a rendering thread, while any host running on a multi-CPU machine may have a render thread per CPU. Host may batch effects off to a render farm, where the same effect has separate frames rendered on completely different machines. OFX needs to address all these situations.

Threads in the host application can be broken into two categories…

- main theaads , where any action may be called
- render threads where only a subset of actions may be called.

For a given effect instance, there can be only one main thread and zero or more render threads. An instance must be able to handle simultaneous actions called on the main and render threads. A plugin can control the number of simultaneous render threads via the *kOfxImageEffectPluginRenderThreadSafety* effect descriptor property.

The only actions that can be called on a render thread are…

- *kOfxImageEffectActionBeginSequenceRender*
- *kOfxImageEffectActionRender*
- *kOfxImageEffectActionEndSequenceRender*
- *kOfxImageEffectActionIsIdentity*
- *kOfxImageEffectActionGetFramesNeeded*
- *kOfxImageEffectActionGetRegionOfDefinition*
- *kOfxImageEffectActionGetRegionsOfInterest*

If a plugin cannot support this multi-threading behaviour, it will need to perform explicit locking itself, using the locking mechanisms in the suites defined in ofxMultiThread.h.

This will also mean that the host may need to perform locking on the various function calls over the API. For example, a main and render thread may both simultaneously attempt to access a parameter from a single effect instance. The locking should…

- block write/read access
- not block on read/read access
- be fine grained at the level of individual function calls,
- be transparent to the plugin, so it will block until the call succeeds.

For example, a render thread will only cause a parameter to lock out writes only for the duration of the call that reads the parameter, not for the duration of the whole render action. This will allow a main thread to continue writing to the parameter during a render. This is especially important if you have a custom interactive GUI that you want to keep working during a render call.

Note that a main thread should generally issue an abort to any linked render thread when a parameter or other value affecting the effect (eg: time) has been changed by the user. A re-render should then be issued so that a correct frame is created.

How an effect handles simulanteous calls to render is dealt with in the multi-thread rendering section.

Many hosts get around the problem of sharing a single instance in a UI thread and a render thread by having two instances, one for the user to interact with and a render only one that shadows the UI instance.

1.9.1 Recursive Actions

When running on a main thread, some actions may end up being called recursively. A plug-in must be able to deal with this. For example consider the following sequence of events in a plugin…

1. user sets parameter A in a GUI
2. host issues `kOfxActionInstanceChanged` action
3. plugin traps that and sets parameter B
    1. host issues a new `kOfxActionInstanceChanged` action for parameter B
    2. plugin traps that and changes some internal private state and requests the overlay redraw itself
        1. `kOfxInteractActionDraw` issued to the effect's overlay
        2. plugin draws overlay
        3. `kOfxInteractActionDraw` returns
    3. `kOfxActionInstanceChanged` action for parameter B returns
4. `kOfxActionInstanceChanged` action returns

The image effect actions which may trigger a recursive action call on a single instance are…

- `kOfxActionBeginInstanceChanged`
- `kOfxActionInstanceChanged`
- `kOfxActionEndInstanceChanged`
- `kOfxActionSyncPrivateData`

The interact actions which may trigger a recursive action to be called on the associated plugin instance are…

- `kOfxInteractActionGainFocus`
- `kOfxInteractActionKeyDown`

- *kOfxInteractActionKeyRepeat*
- *kOfxInteractActionKeyUp*
- *kOfxInteractActionLoseFocus*
- *kOfxInteractActionPenDown*
- *kOfxInteractActionPenMotion*
- *kOfxInteractActionPenUp*

The image effect actions which may be called recursively are…

- *kOfxActionBeginInstanceChanged*
- *kOfxActionInstanceChanged*
- *kOfxActionEndInstanceChanged*
- *kOfxImageEffectActionGetClipPreferences*
- *kOfxImageEffectActionGetRegionOfDefinition* (as a result of calling

  OfxImageEffectSuiteV1::clipGetImage()

  from *kOfxActionInstanceChanged*)
- *kOfxImageEffectActionGetRegionsOfInterest* (as a result of calling *OfxImageEffectSuiteV1::clipGetImage()* from *kOfxActionInstanceChanged*)

The interact actions which may be called recursively are…

- *kOfxInteractActionDraw*

1.10 Coordinate Systems

1.10.1 Spatial Coordinates

All OFX spatial coordinate systems have the positive Y axis pointing up, and the positive X axis pointing right.

As stated above, images are simply some rectangle in a potentially infinite plane of pixels. However, this is an idealisation of what really goes on, as images composed of real pixels have to take into account pixel aspect ratios and proxy render scales, as such they will *not* be in the same space as the image plane. To deal with this, OFX has three spatial coordinate systems

- The Canonical Coordinate System which describes the idealised image plane
- The Pixel Coordinate System which describes coordinates in addressable pixels
- The Normalised Canonical Coordinate System which allows for resolution independent description of parameters

Canonical Coordinates

The idealised image plane is always in a coordinate system of square unscaled pixels. For example a PAL D1 frame occupies (0,0) to (768,576). We call this the *Canonical Coordinate System*.

Many operations take place in canonical coordinates, parameter values are expressed in them while the and RoD and RoI actions report their values back in them.

The Canonical coordinate system is always referenced by double floating point values, generally via a `OfxRectD` structure:

struct **`OfxRectD`**

Defines two dimensional double region.

Regions are x1 <= x < x2

Infinite regions are flagged by setting

- x1 = *kOfxFlagInfiniteMin*
- y1 = *kOfxFlagInfiniteMin*
- x2 = *kOfxFlagInfiniteMax*
- y2 = *kOfxFlagInfiniteMax*

Public Members

double **`x1`**

double **`y1`**

double **`x2`**

double **`y2`**

Pixel Coordinates

Real images, where we have to deal with addressable pixels in memory, are in a coordinate system of non-square proxy scaled integer values. So a PAL D1 image, being renderred as a half resolution proxy would be (0,0) to (360, 288), which takes into account both the pixel aspect ratio of 1.067 and a scale factor of 0.5f. We call this the **Pixel Coordinate System**.

The Pixel coordinate system is always referenced by integer values, generally via a OfxRectI structure. It is used when referring to operations on actual pixels, and so is how the bounds of images are described and the render window passed to the render action.

Mapping Between The Spatial Coordinate Systems

To map between the two the pixel aspect ratio and the render scale need to be known, and it is a simple case of multiplication and rounding. More specifically, given…

- pixel aspect ratio, PAR , found on the image property *kOfxImagePropPixelAspectRatio*
- render scale in X SX , found on the first dimension of the effect property *kOfxImageEffectPropRenderScale*
- render scale in Y SY , found on the second dimension of the effect property *kOfxImageEffectPropRenderScale*
- field scale in Y FS , this is
  - 0.5 if the image property *kOfxImagePropField* is *kOfxImageFieldLower* or *kOfxImageFieldUpper*
  - 1.0 otherwise.

To map an X and Y coordinates from Pixel coordinates to Canonical coordinates, we perform the following multiplications…

```
X' = (X * PAR)/SX
Y' = Y/(SY * FS)
```

To map an X and Y coordinates from Canonical coordinates to Pixel coordinates, we perform the following multiplications…

```
X' = (X * SX)/PAR
Y' = Y * SY * FS
```

The Normalized Coordinate System

Note, normalised parameters and the normalised coordinate system are being deprecated in favour of spatial parameters which can handle the project rescaling without the problems of converting to/from normalised coordinates.

On most editing an compositing systems projects can be moved on resolutions, for example a project may be set up at high definition then have several versions rendered out at different sizes, say a PAL SD version, an NTSC SD version and an HD 720p version.

This causes problems with parameters that describe spatial coordinates. If they are expressed as absolute positions, the values will be incorrect as the project is moved from resolution to resolution. For example, a circle drawn at (384,288) in PAL SD canonical coordinates will be in the centre of the output. Re-render that at 2K film, it will be in the bottom left hand corner, which is probably not the correct spot.

To get around this, OFX allows parameters to be flagged as *normalised*, which is a resolution independent method of representing spatial coordinates. In this coordinate system, a point expressed as (0.5, 0.5) will appear in the centre of the screen, always.

To transform between normalised and canonical coordinates a simple linear equation is required. What that is requires a certain degree of explanation. It involves three two dimensional values…

- the project extent the resolution of the project, eg: PAL SD
- the project size how much of that is used by imagery, eg: the letter box area in a 16:9 PAL SD project
- the project offset the bottom left corner of the extent being used, eg: the BL corner of a 16:9 PAL SD project

As described above, the project extent is the section of the image plane that is covered by an image that is the desired output of the project, so for a PAL SD project you get an extent of 0,0 to 768,576. As the project is always rooted at the origin, so the extent is actually a size.

Project sizes and offsets are a bit less obvious. Consider a project that is going to be output as PAL D1 imagery, the extent will be 0,0 to 768,576. However our example is a letter box 16:9 project, which leaves a strip of black at bottom and top. The size of the letter box is 768 by 432, while the bottom left of the letter box is offset from the origin by 0,77. The ASCII art below shows the details…..

```
                                    (768,576)
        ---------------------------------------
        |                                     |
        |                BLACK                |
        |.....................................| (768, 504)
        |                                     |
        |                                     |
        |        LETTER BOXED IMAGERY         |
        |                                     |
        |                                     |
(0,72)  |.....................................|
        |                                     |
        |                BLACK                |
        |                                     |
        ---------------------------------------
 (0,0)
```

So in this example…

- the extent of the project is the full size of the output image, which is 768x576,
- the size of the project is the size of the letter box section, which is 768x432,
- the offset of the project is the bottom left corner of the project window, which is 0,72.

The properties on an effect instance handle allow you to fetch these values…

- *kOfxImageEffectPropProjectExtent* for the extent of the current project,
- *kOfxImageEffectPropProjectSize* for the size of the current project,
- *kOfxImageEffectPropProjectOffset* for the offset of the current project.

So to map from normalised coordinates to canonical coordinates, you use the project size and offset…

- for values that represent a size simply multiply the normalised coordinate by the project size
- for values that represent an absolute position, multiply the normalised coordinate by the project size then add the project origin

To flag to the host that a parameter as normalised, we use the kOfxParamPropDoubleType property. Parameters that are so flagged have values set and retrieved by an effect in normalized coordinates. However a host can choose to represent them to the user in whatever space it chooses. The values that this property can take are…

- **kOfxParamDoubleTypeX**

  value for the *kOfxParamPropDoubleType* property, indicating a size in canonical coords in the X dimension. See *kOfxParamPropDoubleType*.

  A size in the X dimension dimension (1D only), new for 1.2

- **kOfxParamDoubleTypeXAbsolute**

  value for the *kOfxParamPropDoubleType* property, indicating an absolute position in canonical coords in the X dimension. See *kOfxParamPropDoubleType*.

  A position in the X dimension (1D only), new for 1.2

- **`kOfxParamDoubleTypeY`**

  value for the *kOfxParamPropDoubleType* property, indicating a size in canonical coords in the Y dimension. See *kOfxParamPropDoubleType*.

  A size in the Y dimension dimension (1D only), new for 1.2

- **`kOfxParamDoubleTypeYAbsolute`**

  value for the *kOfxParamPropDoubleType* property, indicating an absolute position in canonical coords in the Y dimension. See *kOfxParamPropDoubleType*.

  A position in the X dimension (1D only), new for 1.2

- **`kOfxParamDoubleTypeXY`**

  value for the *kOfxParamPropDoubleType* property, indicating a 2D size in canonical coords. See *kOfxParamPropDoubleType*.

  A size in the X and Y dimension (2D only), new for 1.2

- **`kOfxParamDoubleTypeXYAbsolute`**

  value for the *kOfxParamPropDoubleType* property, indicating a 2D position in canonical coords. See *kOfxParamPropDoubleType*.

  A position in the X and Y dimension (2D only), new for 1.2

- **`kOfxParamDoubleTypeNormalisedX`**

  value for the *kOfxParamPropDoubleType* property, indicating a size normalised to the X dimension. See *kOfxParamPropDoubleType*. ofxParam.h

  Deprecated:

  – V1.3: Deprecated in favour of ::OfxParamDoubleTypeX V1.4: Removed

  Normalised size with respect to the project's X dimension (1D only), deprecated for 1.2

- **`kOfxParamDoubleTypeNormalisedXAbsolute`**

  value for the *kOfxParamPropDoubleType* property, indicating an absolute position normalised to the X dimension. See *kOfxParamPropDoubleType*. ofxParam.h

  Deprecated:

  – V1.3: Deprecated in favour of ::OfxParamDoubleTypeXAbsolute V1.4: Removed

  Normalised absolute position on the X axis (1D only), deprecated for 1.2

- **`kOfxParamDoubleTypeNormalisedY`**

  value for the *kOfxParamPropDoubleType* property, indicating a size normalised to the Y dimension. See *kOfxParamPropDoubleType*. ofxParam.h

  Deprecated:

  – V1.3: Deprecated in favour of ::OfxParamDoubleTypeY V1.4: Removed

Normalised size wrt to the project's Y dimension (1D only), deprecated for 1.2

- **`kOfxParamDoubleTypeNormalisedYAbsolute`**

  value for the *kOfxParamPropDoubleType* property, indicating an absolute position normalised to the Y dimension. See *kOfxParamPropDoubleType*. ofxParam.h

  Deprecated:

  - V1.3: Deprecated in favour of ::OfxParamDoubleTypeYAbsolute V1.4: Removed

  Normalised absolute position on the Y axis (1D only), deprecated for 1.2

- **`kOfxParamDoubleTypeNormalisedXY`**

  value for the *kOfxParamPropDoubleType* property, indicating normalisation to the X and Y dimension for 2D params. See *kOfxParamPropDoubleType*. ofxParam.h

  Deprecated:

  - V1.3: Deprecated in favour of ::OfxParamDoubleTypeXY V1.4: Removed

  Normalised to the project's X and Y size (2D only), deprecated for 1.2

- **`kOfxParamDoubleTypeNormalisedXYAbsolute`**

  value for the *kOfxParamPropDoubleType* property, indicating normalisation to the X and Y dimension for a 2D param that can be interpretted as an absolute spatial position. See *kOfxParamPropDoubleType*. ofxParam.h

  Deprecated:

  - V1.3: Deprecated in favour of *kOfxParamDoubleTypeXYAbsolute* V1.4: Removed

  Normalised to the projects X and Y size, and is an absolute position on the image plane, deprecated for 1.2.

For example, we have an effect that draws a circle. It has two parameters a 1D double radius parameter and a 2D double position parameter. It would flag the radius to be `kOfxParamDoubleTypeNormalisedX`, fetch the value and scale that by the project size before we render the circle. The host should present such normalised parameters to the user in a *sensible* range. So for a PAL project, it would be from 0..768, where the plug-in sees 0..1.

The position can be handled by the `kOfxParamDoubleTypeNormalisedXYAbsolute` case. In which case the plugin must scale the parameter's value by the project size and add in the project offset. This will allow the positional parameter to be moved between projects transparently.

1.10.2 Temporal Coordinates

Within OFX Image Effects, there is only one temporal coordinate system, this is in output frames referenced to the start of the effect (so the first affected frame = 0). All times within the API are in that coordinate system.

All clip instances have a property that indicates the frames for which they can generate image data. This is `kOfxImageEffectPropFrameRange`, a 2D double property, with the first dimension being the first, and the second being last the time at which the clip will generate data.

Consider the example below, it is showing an effect of 10 frames duration applied to a clip lasting 20 frames. The first frame of the effect is in fact the 5th frame of the clip. Both the input and output have the same frame rate.

```
Effect                 0   1   2   3   4   5   6   7   8   9
Source     0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
```

In this example, if the effect asks for the source image at time '4', the host will actually return the 9th image of that clip. When queried the output and source clip instances would report the following…

```
           range[0]     range[1]     FPS
Output     0            9            25
Source    -4            15           25
```

Consider the slightly more complex example below, where the output has a frame rate twice the input's

```
Effect         0   1   2   3   4   5   6   7   8   9
Source     0       1       2       3       4       5       6       7
```

When queried the output and source clips would report the following.

```
           range[0]     range[1]     FPS
Output     0            9            50
Source    -2            12           25
```

Using simple arithmetic, any effect that needs to access a specific frame of an input, can do so with the formula…

```
f' = (f - range[0]) * srcFPS/outFPS
```

1.11 Images and Clips

1.11.1 What Is An Image?

Image Effects process images (funny that), this chapter describes images and clips of images, how they behave and how to deal with them.

Firstly some definitions…

- an image is a rectangular array of addressable pixels,
- a clip is a contiguous sequence of images that vary over time.

Images and clips contain pixels, these pixels can currently be of the following types…

- a colour pixel with red, green, blue, alpha components
- a colour pixel with red, green and blue components
- single component 'alpha' images

The components of the pixels can be of the following types…

- 8 bit unsigned byte, with the nominal black and white points at 0 and 255 respectively,
- 16 bit unsigned short, with the nominal black and white points at 0 and 65535 respectively,
- 32 bit float, with the nominal black and white points at 0.0f and 1.0f respectively, component values are not clipped to 0.0f and 1.0f.

Components are packed per pixel in the following manner…

- RGBA pixels as R, G, B, A

- RGB pixels as R, G, B

There are several structs for pixel types in *ofxCore.h <https://github.com/ofxa/openfx/blob/master/include/ofxCore.h>* that can be used for raw pixels in OFX.

Images are always left to right, bottom to top, with the pixel data pointer being at the bottom left of the image. The pixels in a scan line are contiguously packed.

Scanlines need *not* be contiguously packed. The number of *bytes* between between a pixel in the same column, but separated by a scan line is known as the *rowbytes* of an image. Rowbytes can be negative, allowing for compositing systems with a native top to bottom scanline order to trivially support bottom to top images.

Clips and images also have a *pixel aspect ratio*, this is how much an actual addressable pixel must be stretched by in X to be square. For example PAL SD images have a pixel aspect ratio of 1.06666.

Images are rectangular, whose integral bounds are in *Pixel coordinates*, with the image being X1 <= X < X2 and Y1 <= Y < Y2, ie: exclusive on the top and right. The bounds represent the amount of data present in the image, which may be larger, smaller or equal to the Region of Definition of the image, depending on the architecture supported by the plugin. The `kOfxImagePropBounds` property on an image holds this information.

An image also contains it's RoD in image coordinates, in the `kOfxImagePropRegionOfDefinition` property. The RoD is the maximum area that an image may have pixels in, t he bounds are the actual addressable pixels present in an image. This allows for tiled rendering an so on.

Clips have a frame rate, which is the number of frames per second they are to be displayed at. Some clips may be continuously samplable (for example, if they are connected to animating geometry that can be rendered at arbitrary times), if this is so, the frame rate for these clips is set to 0.

Images may be composed of full frames, two fields or a single field, depending on its source and how the effect requests the image be processed. Clips are either full frame sequences or fielded sequences.

Images and clips also have a premultiplication state, this represents how the alpha component and the RGB/YUV components may have interacted.

1.11.2 Defining Clips

During an the effect's describe in context action an effect *must* define the clips mandated for that context, it can also define extra clips that it may need for that context. It does this using the :cpp:func`OfxImageEffectSuiteV1::clipDefine` function, the property handle returned by this function is purely for definition purposes only. It has not persistence outside the describe in context action and is distinct to the clip property handles used by instances. The *name* parameter is how you can later access that clip in a plugin instance via the :cpp:func`OfxImageEffectSuiteV1::clipGetHandle` function.

During the describe in context action, the plugin sets properties on a clip to control its use. The properties that can be set during a describe in context call are…

- `kOfxPropLabel` to give a user readable name to the clip (the host need not use this, for example in a transition it is redundant),
- `kOfxImageEffectPropSupportedComponents` to specify which components it is willing to accept on that clip,
- `kOfxImageClipPropOptional` to specify if the clip is optional,
- **`kOfxImageClipPropFieldExtraction`** specifies how to extract fielded images from a clip, see *this section*

    for more details on field and field rendering
- `kOfxImageEffectPropTemporalClipAccess` whether the effect wants to access images from the clip at times other that the frame being renderred.

Plugins *must* indicate which pixel depths they can process by setting the `kOfxImageEffectPropSupportedPixelDepths` on the plugin handle during the describe action.

Pixel Aspect Ratios, frame rates, fielding, components and pixel depths are constant for the duration of a clip, they cannot changed from frame to frame.

Note:

- it is an error not to set the `kOfxImageEffectPropSupportedPixelDepths` plugin property during its describe action
- it is an error not to define a mandated input clip during the describe in context action
- it is an error not to set the `kOfxImageEffectPropSupportedComponents` on an input clip during describe in context

1.11.3 Getting Images From Clips

Clips in instances are retrieved via the :cpp:func`OfxImageEffectSuiteV1::clipGetHandle` function. This returns a property handle for the clip in a specific instance. This handle is valid for the duration of the instance.

Images are fetched from a clip via the `OfxImageEffectSuiteV1::clipGetImage()` function. This takes a time and an optional region to extract an image at from a given clip. This returns, in a property handle, an image fetched from the clip at a specific time. The handle contains all the information relevant to dealing with that image.

Once fetched, an image must be released via the `OfxImageEffectSuiteV1::clipReleaseImage()` function. All images must be released within the action they were fetched in. You cannot retain an image after an action has returned.

Images may be fetched from an attached clip in the following situations…

- in the `kOfxImageEffectActionRender` action
- in the `kOfxActionInstanceChanged`

and `kOfxActionEndInstanceChanged` actions with a `kOfxPropChangeReason` of `kOfxChangeUserEdited`

A host may not be able to support random temporal access, it flags its ability to do so via the `kOfxImageEffectPropTemporalClipAccess` property. A plugin that wishes to perform random temporal access must set a property of that name on the plugin handle and the clip it wishes to perform random access from.

Note:

- it is an error for a plugin to attempt random temporal image access if the host does not support it
- it is an error for a plugin to attempt random temporal image access

if it has not flagged that it wishes to do so and the clip it wishes to do so from.

1.11.4 Premultiplication And Alpha

All images and clips have a premultiplication state. This is used to indicate how the image should interpret RGB (or YUV) pixels, with respect to alpha. The premultiplication state can be…

`kOfxImageOpaque`
Used to flag the alpha of an image as opaque

The image is opaque and so has no premultiplication state, but the alpha component in all pixels is set to the white point

`kOfxImagePreMultiplied`
Used to flag an image as premultiplied

The image is premultiplied by it's alpha

`kOfxImageUnPreMultiplied`
Used to flag an image as unpremultiplied

The image is unpremultiplied.

This document won't go into the details of premultiplication, but will simply state that OFX takes notice of it and flags images and clips accordingly.

The premultiplication state of a clip is constant over the entire duration of that clip.

1.11.5 Clips and Pixel Aspect Ratios

All clips and images have a pixel aspect ratio, this is how much a 'real' pixel must be stretched by in X to be square. For example PAL D1 images have a pixel aspect ratio of 1.06666.

The property *`kOfxImageEffectPropSupportsMultipleClipPARs`* is used to control how a plugin deals with pixel aspect ratios. This is both a host and plugin property. For a host it can be set to…

- 0 - the host only supports a single pixel aspect ratio for all clips, input or output, to an effect,
- 1 - the host can support differing pixel aspect ratios for inputs and outputs

For a plugin it can be set to…

- 0 - the plugin expects all pixel aspect ratios to be the same on all clips, input or output
- 1 - the plugin will accept clips of differing pixel aspect ratio.

If a plugin does not accept clips of differing PARs, then the host must resample all images fed to that effect to agree with the output's PAR.

If a plugin does accept clips of differing PARs, it will need to specify the output clip's PAR in the *`kOfxImageEffectActionGetClipPreferences`* action.

1.11.6 Allocating Your Own Images

Under OFX, the images you fetch from the host have already had their memory allocated. If a plug-in needs to define its owns temporary images buffers during processing, or to cache images between actions, then the plug-in should use the image memory allocation routines declared in *OfxImageEffectSuiteV1*. The reason for this is that many host have special purpose memory pools they manage to optimise memory usage as images can chew up memory very rapidly (eg: a 2K RGBA floating point film plate is 48 MBytes).

For general purpose (as in less than a megabyte) memory allocation, you should use the memory suite in ofxMemory.h

OFX provides four functions to deal with image memory. These are,

- *OfxImageEffectSuiteV1::imageMemoryAlloc()*
- *OfxImageEffectSuiteV1::imageMemoryFree()*
- *OfxImageEffectSuiteV1::imageMemoryLock()*
- *OfxImageEffectSuiteV1::imageMemoryUnlock()*

A host needs to be able defragment its image memory pool, potentially moving the contents of the memory you have allocated to another address, even saving it to disk under its own virtual memory caching scheme. Because of this when you request a block of memory, you are actually returned a handle to the memory, not the memory itself. To use the memory you must first lock the memory via the imageMemoryLock call, which will then return a pointer to the locked block of memory.

During an single action, there is generally no need to lock/unlock any temporary buffers you may have allocated via this mechanism. However image memory that is cached between actions should always be unlocked while it is not actually being used. This allows a host to do what it needs to do to optimise memory usage.

Note that locks and unlocks nest. This implies that there is a lock count kept on the memory handle, also not that this lock count cannot be negative. So unlocking a completely unlocked handle has no effect.

An example is below....

```
// get a memory handle
OfxImageMemoryHandle memHandle;
gEffectSuite->imageMemoryAlloc(0, imageSize, &memHandle);

// lock the handle and get a pointer
void *memPtr;
gEffectSuite->imageMemoryLock(memHandle, &memPtr);

... // do stuff with our pointer

// now unlock it
gEffectSuite->imageMemoryUnlock(memHandle);


// lock it again, note that this may give a completely different address to␣
↪the last lock
gEffectSuite->imageMemoryLock(memHandle, &memPtr);

... // do more stuff

// unlock it again
gEffectSuite->imageMemoryUnlock(memHandle);
```

(continues on next page)

(continued from previous page)

```
// delete it all
gEffectSuite->imageMemoryFree(memHandle);
```

1.12 Effect Parameters

1.12.1 Introduction

Nearly all plug-ins have some sort of parameters that control their behaviour, the radius of a circle drawer, the frequencies to filter out of an audio signal, the colour of a lens flare and so on.

Seeing as hosts already provide for the general management of their own native parameters (eg: persistence, interface, animation etc. . .), it would make no sense to force plug-ins to do this all themselves.

The OFX Parameters Suite is the means by which parameters are defined and used by the plug-in but maintained by the host. It is defined in the ofxParam.h header file.

Note that the entire state of the plugin is encoded in the value of its parameter set. If you need to persist some sort of private data, you must do so by setting param values in the effects. The *kOfxActionSyncPrivateData* is an action that tells you when to flush any values that need persisting out to the effects param set. You can reconstruct your private data during the *kOfxActionCreateInstance*.

1.12.2 Defining Parameters

A plugin needs to define it's parameters during a describe action. It does this with the *OfxParameterSuiteV1::paramDefine()* function, which returns a handle to a parameter *description*. Parameters cannot currently be defined outside of the plugins describe actions.

Parameters are uniquely labelled within a plugin with an ASCII null terminated C-string. This name is not necassarily meant to be end-user readable, various properties are provided to set the user visible labels on the param.

All parameters hold properties, though the exact set of properties on a parameter is dependent on the type of the parameter.

A parameter's handle comes in two slightly different flavours. The handle returned inside a plugin's describe action is not an actual instance of a parameter, it is there for the purpose description only. You can only set properties on that handle (eg: label, min/max value, default . . .), you cannot get values from it or set values in it. The parameters defined in the describe action will common to all instances of a plugin.

The handle returned by *OfxParameterSuiteV1::paramGetHandle()* outside of a describe action will be a working instance of a parameter, you can still set (some) properties of the parameter, and all the get/set value functions are now usable.

1.12.3 Parameter Types

There are seventeen types of parameter. These are

- **kOfxParamTypeInteger**

  String to identify a param as a single valued integer.

- **kOfxParamTypeInteger2D**

  String to identify a param as a Two dimensional integer point parameter.

- `kOfxParamTypeInteger3D`
  String to identify a param as a Three dimensional integer parameter.
- `kOfxParamTypeDouble`
  String to identify a param as a Single valued floating point parameter.
- `kOfxParamTypeDouble2D`
  String to identify a param as a Two dimensional floating point parameter.
- `kOfxParamTypeDouble3D`
  String to identify a param as a Three dimensional floating point parameter.
- `kOfxParamTypeRGB`
  String to identify a param as a Red, Green and Blue colour parameter.
- `kOfxParamTypeRGBA`
  String to identify a param as a Red, Green, Blue and Alpha colour parameter.
- `kOfxParamTypeBoolean`
  String to identify a param as a Single valued boolean parameter.
- `kOfxParamTypeChoice`
  String to identify a param as a Single valued, ‘one-of-many’ parameter.
- `kOfxParamTypeString`
  String to identify a param as a String (UTF8) parameter.
- `kOfxParamTypeCustom`
  String to identify a param as a Plug-in defined parameter.
- `kOfxParamTypePushButton`
  String to identify a param as a PushButton parameter.
- `kOfxParamTypeGroup`
  String to identify a param as a Grouping parameter.
- `kOfxParamTypePage`
  String to identify a param as a page parameter.
- `kOfxParamTypeParametric`
  String to identify a param as a single valued integer.

1.12.4 Multidimensional Parameters

Some parameter types are multi dimensional, these are…

- *kOfxParamTypeDouble2D*
- *kOfxParamTypeInteger2D*
- *kOfxParamTypeDouble3D*
- *kOfxParamTypeInteger3D*
- *kOfxParamTypeRGB*
- *kOfxParamTypeRGBA*
- *kOfxParamTypeParametric*

These parameters are treated in an atomic manner, so that all dimensions are set/retrieved simultaneously. This applies to keyframes as well.

The non colour parameters have an implicit 'X', 'Y' and 'Z' dimension, and any interface should display them with such labels.

1.12.5 Integer Parameters

These are typed by *kOfxParamTypeInteger*, *kOfxParamTypeInteger2D* and *kOfxParamTypeInteger3D*.

Integer parameters are of 1, 2 and 3D varieties and contain integer values, between INT_MIN and INT_MAX.

1.12.6 Double Parameters

These are typed by *kOfxParamTypeDouble*, *kOfxParamTypeDouble2D* and *kOfxParamTypeDouble3D*.

Double parameters are of 1, 2 and 3D varieties and contain double precision floating point values.

1.12.7 Colour Parameters

These are typed by *kOfxParamTypeRGB* and *kOfxParamTypeRGBA*.

Colour parameters are 3 or 4 dimensional double precision floating point parameters. They are displayed using the host's appropriate interface for a colour. Values are always normalised in the range [0 .. 1], with 0 being the nominal black point and 1 being the white point.

1.12.8 Boolean Parameters

This is typed by *kOfxParamTypeBoolean*.

Boolean parameters are integer values that can have only one of two values, 0 or 1.

1.12.9 Choice Parameters

This is typed by *kOfxParamTypeChoice*.

Choice parameters are integer values from 0 to N-1, which correspond to N labeled options.

Choice parameters have their individual options set via the *kOfxParamPropChoiceOption* property, for example

```
gPropHost->propSetString(myChoiceParam, kOfxParamPropChoiceOption, 0, "1st␣
↪Choice");
gPropHost->propSetString(myChoiceParam, kOfxParamPropChoiceOption, 1, "2nd␣
↪Choice");
gPropHost->propSetString(myChoiceParam, kOfxParamPropChoiceOption, 2, "3nd␣
↪Choice");
...
gPropHost->propSetString(myChoiceParam, kOfxParamPropChoiceOption, n, "nth␣
↪Choice");
```

It is an error to have gaps in the choices after the describe action has returned.

1.12.10 String Parameters

This is typed by *kOfxParamTypeString*.

String parameters contain null terminated `char *` UTF8 C strings. They can be of several different variants, which is controlled by the *kOfxParamPropStringMode* property, these are

- **kOfxParamStringIsSingleLine**
  Used to set a string parameter to be single line, value to be passed to a kOfxParamPropStringMode property.
- **kOfxParamStringIsMultiLine**
  Used to set a string parameter to be multiple line, value to be passed to a kOfxParamPropStringMode property.
- **kOfxParamStringIsFilePath**
  Used to set a string parameter to be a file path, value to be passed to a kOfxParamPropStringMode property.
- **kOfxParamStringIsDirectoryPath**
  Used to set a string parameter to be a directory path, value to be passed to a kOfxParamPropStringMode property.
- **kOfxParamStringIsLabel**
  Use to set a string parameter to be a simple label, value to be passed to a kOfxParamPropStringMode property.

1.12.11 Group Parameters

This is typed by `kOfxParamTypeGroup`.

Group parameters allow all parameters to be arranged in a tree hierarchy. They have no value, they are purely a grouping element.

All parameters have a `kOfxParamPropParent` property, which is a string property naming the group parameter which is its parent.

The empty string "" is used to label the root of the parameter hierarchy, which is the default parent for all parameters.

Parameters inside a group are ordered by their order of addition to that group, which implies parameters in the root group are added in order of definition.

Any host based hierarchical GUI should use this hierarchy to order parameters (eg: animation sheets).

1.12.12 Page Parameters

This is typed by `kOfxParamTypePage`.

Page parameters are covered in detail in their own *section*.

1.12.13 Custom Parameters

This is typed by `kOfxParamTypeCustom`.

Custom parameters contain null terminated char * C strings, and may animate. They are designed to provide plugins with a way of storing data that is too complicated or impossible to store in a set of ordinary parameters.

If a custom parameter animates, it must set its `kOfxParamPropCustomInterpCallbackV1` property, which points to a function with the following signature:

`OfxStatus() OfxCustomParamInterpFuncV1 (OfxParamSetHandle instance, OfxPropertySetHandle inArgs, OfxPropertySetHandle outArgs)`

Function prototype for custom parameter interpolation callback functions.

- instance the plugin instance that this parameter occurs in
- inArgs handle holding the following properties...
  - kOfxPropName - the name of the custom parameter to interpolate
  - kOfxPropTime - absolute time the interpolation is ocurring at
  - kOfxParamPropCustomValue - string property that gives the value of the two keyframes to interpolate, in this case 2D
  - kOfxParamPropInterpolationTime - 2D double property that gives the time of the two keyframes we are interpolating
  - kOfxParamPropInterpolationAmount - 1D double property indicating how much to interpolate between the two keyframes
- outArgs handle holding the following properties to be set
  - kOfxParamPropCustomValue - the value of the interpolated custom parameter, in this case 1D

This function allows custom parameters to animate by performing interpolation between keys.

The plugin needs to parse the two strings encoding keyframes on either side of the time we need a value for. It should then interpolate a new value for it, encode it into a string and set the *kOfxParamPropCustomValue* property with this on the outArgs handle.

The interp value is a linear interpolation amount, however his may be derived from a cubic (or other) curve.

This function is used to interpolate keyframes in custom params.

Custom parameters have no interface by default. However,

- if they animate, the host's animation sheet/editor should present a keyframe/curve representation to allow positioning of keys and control of interpolation. The 'normal' (ie: paged or hierarchical) interface should not show any gui.
- if the custom param sets its kOfxParamPropInteractV1 property, this should be used by the host in any normal (ie: paged or hierarchical) interface for the parameter.

Custom parameters are mandatory, as they are simply ASCII C strings. However, animation of custom parameters an support for an in editor interact is optional.

1.12.14 Push Button Parameters

This is typed by `kOfxParamTypePushButton`.

Push button parameters have no value, they are there so a plugin can detect if they have been pressed and perform some action. If pressed, a `kOfxActionInstanceChanged` action will be issued on the parameter with a `kOfxPropChangeReason` of `kOfxChangeUserEdited`.

1.12.15 Animation

By default the following parameter types animate…

- `kOfxParamTypeInteger`
- `kOfxParamTypeInteger2D`
- `kOfxParamTypeInteger3D`
- `kOfxParamTypeDouble`
- `kOfxParamTypeDouble2D`
- `kOfxParamTypeDouble3D`
- `kOfxParamTypeRGBA`
- `kOfxParamTypeRGB`

The following types cannot animate…

- `kOfxParamTypeGroup`
- `kOfxParamTypePage`
- `kOfxParamTypePushButton`

The following may animate, depending on the host. Properties exist on the host to check this. If the host does support animation on them, then they do **not** animate by default. They are…

- `kOfxParamTypeCustom`
- `kOfxParamTypeString`

- *kOfxParamTypeBoolean*
- *kOfxParamTypeChoice*

By default the *OfxParameterSuiteV1::paramGetValue()* will get the 'current' value of the parameter. To access values in a potentially animating parameter, use the *OfxParameterSuiteV1::paramGetValueAtTime()* function.

Keys can be manipulated in a parameter using a variety of functions, these are…

- *OfxParameterSuiteV1::paramSetValueAtTime()*
- *OfxParameterSuiteV1::paramGetNumKeys()*
- *OfxParameterSuiteV1::paramGetKeyTime()*
- *OfxParameterSuiteV1::paramGetKeyIndex()*
- *OfxParameterSuiteV1::paramDeleteKey()*
- *OfxParameterSuiteV1::paramDeleteAllKeys()*

1.12.16 Parameter Interfaces

Parameters will be presented to the user in some form of interface. Typically on most host systems, this comes in three varieties…

- a paged layout, with parameters spread over multiple controls pages (eg: the FLAME control pages)
- a hierarchical layout, with parameters presented in a grouped tree (eg: the After Effects 'effects' window)
- an animation sheet, showing animation curves and key frames. Typically this is hierarchical.

Most systems have an animation sheet and present one of either the paged or the hierarchical layouts.

Because a hierarchy of controls is explicitly set during plugin definition, the case of the animation sheet and hierarchial GUIs are taken care of explicitly.

1.12.17 Paged Parameter Editors

A paged layout of controls is difficult to standardise, as the size of the page and controls, how the controls are positioned on the page, how many controls appear on a page etc… depend very much upon the host implementation. A paged layout is ideally best described in the .XML resource supplied by the plugin, however a fallback page layout can be specified in OFX via the *kOfxParamTypePage* parameter type.

Several host properties are associated with paged layouts, these are…

- *kOfxParamHostPropMaxPages* The maximum number of pages you may use, 0 implies an unpaged layout
- *kOfxParamHostPropPageRowColumnCount* The number of rows and columns for parameters in the paged layout.

Each page parameter represents a page of controls. The controls in that page are set by the plugin using the *kOfxParamPropPageChild* multi-dimensional string. For example…

```
OfxParamHandle  page;
gHost->paramDefine(plugin, kOfxParamTypePage, "Main", &page);

propHost->propSetString(page, kOfxParamPropPageChild, 0, "size");      // add␣
↪the size parameter to the top left of the page
propHost->propSetString(page, kOfxParamPropPageChild, 1, kOfxParamPageSkipRow);
↪ // skip a row
```

(continues on next page)

(continued from previous page)

```
propHost->propSetString(page, kOfxParamPropPageChild, 2, "centre");    // add␣
↪the centre parameter
propHost->propSetString(page, kOfxParamPropPageChild, 3,␣
↪kOfxParamPageSkipColumn); // skip a column, we are now at the top of the␣
↪next column
propHost->propSetString(page, kOfxParamPropPageChild, 4, "colour"); // add the␣
↪colour parameter
```

The host then places the parameters on that page in the order they were added, starting at the top left and going down columns, then across rows as they fill.

Note that there are two pseudo parameters names used to help control layout:

kOfxParamPageSkipRow

Pseudo parameter name used to skip a row in a page layout.

Passed as a value to the *kOfxParamPropPageChild* property.

See ParametersInterfacesPagedLayouts for more details.

kOfxParamPageSkipColumn

Pseudo parameter name used to skip a row in a page layout.

Passed as a value to the *kOfxParamPropPageChild* property.

See ParametersInterfacesPagedLayouts for more details.

These will help control how parameters are added to a page, allowing vertical or horizontal slots to be skipped.

A host sets the order of pages by using the instance's *kOfxPluginPropParamPageOrder* property. Note that this property can vary from context to context, so you can exclude pages in contexts they are not useful in. For example…

```
OfxStatus describeInContext(OfxImageEffectHandle plugin)
{
...
    // order our pages of controls
    propHost->propSetString(paramSetProp, kOfxPluginPropParamPageOrder, 0,
↪"Main");
    propHost->propSetString(paramSetProp, kOfxPluginPropParamPageOrder, 1,
↪"Sampling");
    propHost->propSetString(paramSetProp, kOfxPluginPropParamPageOrder, 2,
↪"Colour Correction");
    if(isGeneralContext)
       propHost->propSetString(paramSetProp, kOfxPluginPropParamPageOrder, 3,
↪"Dance! Dance! Dance!");
...
}
```

Note: Parameters can be placed on more than a single page (this is often useful). Group parameters cannot be added to a page. Page parameters cannot be added to a page or group.

1.12.18 Instance changed callback

Whenever a parameter's value changes, the host is expected to issue a call to the *kOfxActionInstanceChanged* action with the name of the parameter that changed and a reason indicating who triggered the change:

kOfxChangeUserEdited
String used as a value to *kOfxPropChangeReason* to indicate a user has changed something.

kOfxChangePluginEdited
String used as a value to *kOfxPropChangeReason* to indicate the plug-in itself has changed something.

kOfxChangeTime
String used as a value to *kOfxPropChangeReason* to a time varying object has changed due to a time change.

1.12.19 Parameter Undo/Redo

Hosts usually retain an undo/redo stack, so users can undo changes they make to a parameter. Often undos and redos are grouped together into an undo/redo block, where multiple parameters are dealt with as a single undo/redo event. Plugins need to be able to deal with this cleanly.

Parameters can be excluded from being undone/redone if they set the *kOfxParamPropCanUndo* property to 0.

If the plugin changes parameters values by calling the get and set value functions, they will ordinarily be put on the undo stack, one event per parameter that is changed. If the plugin wants to group sets of parameter changes into a single undo block and label that block, it should use the *OfxParameterSuiteV1::paramEditBegin()* and *OfxParameterSuiteV1::paramEditEnd()* functions.

An example would be a 'preset' choice parameter in a sky simulation whose job is to set other parameters to values that achieve certain looks, eg "Dusk", "Midday", "Stormy", "Night" etc… This parameter has a value change callback which looks for *kOfxChangeUserEdited* then sets other parameters, sky colour, cloud density, sun position etc…. It also resets itself to the first choice, which says "Example Skys…".

Rather than have many undo events appear on the undo stack for each individual parameter change, the effect groups them via the paramEditBegin/paramEditEnd and gets a single undo event. The 'preset' parameter would also not want to be undoable as it such an event is redundant. Note that as the 'preset' has been changed it will be sent another instance changed action, however it will have a reason of *kOfxChangePluginEdited*, which it ignores and so stops an infinite loop occurring.

1.12.20 XML Resource Specification for Parameters

Parameters can have various properties overridden via a separate XML based resource file.

1.12.21 Parameter Persistence

All parameters flagged with the *kOfxParamPropPersistant* property will persist when an effect is saved. How the effect is saved is completely up to the host, it may be in a file, a data base, where ever. We call a saved set of parameters a *setup*. A host will need to save the major version number of the plugin, as well as the plugin's unique identifier, in any setup.

When an host loads a set up it should do so in the following manner…

1. examines the setup for the major version number.

2. find a matching plugin with that major version number, if multiple minor versions exist, the plugin with the largest minor version should be used.
3. creates an instance of that plugin with its set of parameters.
4. sets all those parameters to the defaults specified by the plugin.
5. examines the setup for any persistent parameters, then sets the instance's parameters to any found in it.
6. calls create instance on the plugin.

It is *not* an error for a parameter to exist in the plugin but not the setup, and vice versa. This allows a plugin developer to modify parameter sets between point releases, generally by adding new params. The developer should be sure that the default values of any new parameters yield the same behaviour as before they were added, otherwise it would be a breach of the 'major version means compatibility' rule.

1.12.22 Parameter Properties Whose Type Vary

Some properties type depends on the kind of the parameter, eg: `kOfxParamPropDefault` is an int for a integer parameter but a double X 2 for a `kOfxParamTypeDouble2D` parameter.

The variant property types are as follows....

- **`kOfxParamTypeInteger`** int X 1
- **`kOfxParamTypeDouble`** double X 1
- **`kOfxParamTypeBoolean`** int X 1
- **`kOfxParamTypeChoice`** int X 1
- **`kOfxParamTypeRGBA`** double X 4 (normalised to 0..1 range)
- **`kOfxParamTypeRGB`** double X 3 (normalised to 0..1 range)
- **`kOfxParamTypeDouble2D`** double X 2
- **`kOfxParamTypeInteger2D`** int X 2
- **`kOfxParamTypeDouble3D`** double X 3
- **`kOfxParamTypeInteger3D`** int X 3
- **`kOfxParamTypeString`** char * X 1
- **`kOfxParamTypeCustom`** char * X 1
- **`kOfxParamTypePushButton`** none

1.12.23 Types of Double Parameters

Double parameters can be used to represent a variety of data, by flagging what a double parameter is representing, a plug-in allows a host to represent to the user a more appropriate interface than a raw numerical value. Double parameters have the `kOfxParamPropDoubleType` property, which gives some meaning to the value. This can be one of...

- **`kOfxParamDoubleTypePlain`**

  value for the *kOfxParamPropDoubleType* property, indicating the parameter has no special interpretation and should be interpretted as a raw numeric value.

- **`kOfxParamDoubleTypeAngle`**
  value for the *kOfxParamDoubleTypeAngle* property, indicating the parameter is to be interpreted as an angle. See *kOfxParamPropDoubleType*.

- **`kOfxParamDoubleTypeScale`**
  value for the *kOfxParamPropDoubleType* property, indicating the parameter is to be interpreted as a scale factor. See *kOfxParamPropDoubleType*.

- **`kOfxParamDoubleTypeTime`**
  value for the *kOfxParamDoubleTypeAngle* property, indicating the parameter is to be interpreted as a time. See *kOfxParamPropDoubleType*.

- **`kOfxParamDoubleTypeAbsoluteTime`**
  value for the *kOfxParamDoubleTypeAngle* property, indicating the parameter is to be interpreted as an absolute time from the start of the effect. See *kOfxParamPropDoubleType*.

- **`kOfxParamDoubleTypeX`**
  value for the *kOfxParamPropDoubleType* property, indicating a size in canonical coords in the X dimension. See *kOfxParamPropDoubleType*.

- **`kOfxParamDoubleTypeXAbsolute`**
  value for the *kOfxParamPropDoubleType* property, indicating an absolute position in canonical coords in the X dimension. See *kOfxParamPropDoubleType*.

- **`kOfxParamDoubleTypeY`**
  value for the *kOfxParamPropDoubleType* property, indicating a size in canonical coords in the Y dimension. See *kOfxParamPropDoubleType*.

- **`kOfxParamDoubleTypeYAbsolute`**
  value for the *kOfxParamPropDoubleType* property, indicating an absolute position in canonical coords in the Y dimension. See *kOfxParamPropDoubleType*.

- **`kOfxParamDoubleTypeXY`**
  value for the *kOfxParamPropDoubleType* property, indicating a 2D size in canonical coords. See *kOfx-ParamPropDoubleType*.

- **`kOfxParamDoubleTypeXYAbsolute`**
  value for the *kOfxParamPropDoubleType* property, indicating a 2D position in canonical coords. See *kOfx-ParamPropDoubleType*.

- **`kOfxParamDoubleTypeNormalisedX`**
  value for the *kOfxParamPropDoubleType* property, indicating a size normalised to the X dimension. See *kOfxParamPropDoubleType*. ofxParam.h

  Deprecated:

  – V1.3: Deprecated in favour of ::OfxParamDoubleTypeX V1.4: Removed

- **`kOfxParamDoubleTypeNormalisedXAbsolute`**
  value for the *kOfxParamPropDoubleType* property, indicating an absolute position normalised to the X

dimension. See *kOfxParamPropDoubleType*. ofxParam.h

Deprecated:

- V1.3: Deprecated in favour of ::OfxParamDoubleTypeXAbsolute V1.4: Removed

- **`kOfxParamDoubleTypeNormalisedY`**
  value for the *kOfxParamPropDoubleType* property, indicating a size normalised to the Y dimension. See *kOfxParamPropDoubleType*. ofxParam.h

  Deprecated:

  - V1.3: Deprecated in favour of ::OfxParamDoubleTypeY V1.4: Removed

- **`kOfxParamDoubleTypeNormalisedYAbsolute`**
  value for the *kOfxParamPropDoubleType* property, indicating an absolute position normalised to the Y dimension. See *kOfxParamPropDoubleType*. ofxParam.h

  Deprecated:

  - V1.3: Deprecated in favour of ::OfxParamDoubleTypeYAbsolute V1.4: Removed

- **`kOfxParamDoubleTypeNormalisedXY`**
  value for the *kOfxParamPropDoubleType* property, indicating normalisation to the X and Y dimension for 2D params. See *kOfxParamPropDoubleType*. ofxParam.h

  Deprecated:

  - V1.3: Deprecated in favour of ::OfxParamDoubleTypeXY V1.4: Removed

- **`kOfxParamDoubleTypeNormalisedXYAbsolute`**
  value for the *kOfxParamPropDoubleType* property, indicating normalisation to the X and Y dimension for a 2D param that can be interpretted as an absolute spatial position. See *kOfxParamPropDoubleType*. ofxParam.h

  Deprecated:

  - V1.3: Deprecated in favour of *kOfxParamDoubleTypeXYAbsolute* V1.4: Removed

1.12.24 Plain Double Parameters

Double parameters with their `kOfxParamPropDoubleType` property set to `kOfxParamDoubleTypePlain` are uninterpreted. The values represented to the user are what is reported back to the effect when values are retrieved. 1, 2 and 3D parameters can be flagged as `kOfxParamDoubleTypePlain`, which is the default.

For example a physical simulation plugin might have a 'mass' double parameter, which is in kilograms, which should be displayed and used as a raw value.

1.12.25 Angle Double Parameters

Double parameters with their `kOfxParamPropDoubleType` property set to `kOfxParamDoubleTypeAngle` are interpreted as angles. The host could use some fancy angle widget in it's interface, representing degrees, angles mils whatever. However, the values returned to a plugin are always in degrees. Applicable to 1, 2 and 3D parameters.

For example a plugin that rotates an image in 3D would declare a 3D double parameter and flag that as an angle parameter and use the values as Euler angles for the rotation.

1.12.26 Scale Double Parameters

Double parameters with their `kOfxParamPropDoubleType` property set to `kOfxParamDoubleTypeScale` are interpreted as scale factors. The host can represent these as 1..100 percentages, 0..1 scale factors, fractions or whatever is appropriate for its interface. However, the plugin sees these as a straight scale factor, in the 0..1 range. Applicable to 1, 2 and 3D parameters.

For example a plugin that scales the size of an image would declare a 'image scale' parameter and use the raw value of that to scale the image.

1.12.27 Time Double Parameters

Double parameters with their `kOfxParamPropDoubleType` property set to `kOfxParamDoubleTypeTime` are interpreted as a time. The host can represent these as frames, seconds, milliseconds, millennia or whatever it feels is appropriate. However, a visual effect plugin sees such values in 'frames'. Applicable only to 1D double parameters. It is an error to set this on any other type of double parameter.

For example a plugin that does motion blur would have a 'shutter time' parameter and flags that as a time parameter. The value returned would be used as the length of the shutter, in frames.

1.12.28 Absolute Time Double Parameters

Double parameters with their `kOfxParamPropDoubleType` property set to `kOfxParamDoubleTypeAbsoluteTime` are interpreted as an absolute time from the beginning of the effect. The host can represent these as frames, seconds, milliseconds, millennia or whatever it feels is appropriate. However, a plugin sees such values in 'frames' from the beginning of a clip. Applicable only to 1D double parameters. It is an error to set this on any other type of double parameter.

For example a plugin that stabalises all the images in a clip to a specific frame would have a *reference frame* parameter and declare that as an absolute time parameter and use its value to fetch a frame to stablise against.

1.12.29 Spatial Parameters

Parameters that can represent a size or position are essential. To that end there are several values of the *kOfxParamPropDoubleType* that say it should be interpreted as a size or position, in either one or two dimensions.

The original OFX API only specified *normalised* parameters, this proved to be somewhat more of a problem than expected. With the 1.2 version of the API, *spatial* parameters were introduced. Ideally these should be used and the normalised parameter types should be deprecated.

Plugins can check *kOfxPropAPIVersion* to see if these new parameter types are supported, in hosts with version 1.2 or greater they will be.

See the section on coordinate systems to understand some of the terms being discussed.

Spatial Double Parameters

These parameter types represent a size or position in one or two dimensions in *Canonical Coordinate*. The host and plug-in get and set values in this coordinate system. Scaling to *Pixel Coordinate* is the responsibility of the effect.

The default value of a spatial parameter can be set in either a normalised coordinate system or the canonical coordinate system. This is controlled by the *kOfxParamPropDefaultCoordinateSystem* on the parameter descriptor with one of the following value:

kOfxParamCoordinatesCanonical
: Define the canonical coordinate system.

kOfxParamCoordinatesNormalised
: Define the normalised coordinate system.

Parameters can choose to be spatial in several ways…

- ***kOfxParamDoubleTypeX*** size in the X dimension, in canonical coords (1D double only),
- ***kOfxParamDoubleTypeXAbsolute*** positing in the X axis, in canonical coords (1D double only)
- ***kOfxParamDoubleTypeY*** size in the Y dimension, in canonical coords (1D double only),
- ***kOfxParamDoubleTypeYAbsolute*** positing in the Y axis, in canonical coords (1D double only)
- ***kOfxParamDoubleTypeXY*** 2D size, in canonical coords (2D double only),
- ***kOfxParamDoubleTypeXYAbsolute*** 2D position, in canonical coords. (2D double only).

Spatial Normalised Double Parameters

Ideally, normalised parameters should be deprecated and no longer used if spatial parameters are available.

There are several values of the *kOfxParamPropDoubleType* that say it should be interpreted as a size or position. These are expressed and proportional to the current project's size. This will allow the parameter to scale cleanly with project size changes and to be represented to the user in an appropriate range.

For example, the sensible X range of a visual effect plugin is the project's width, say 768 pixels for a PAL D1 definition video project. The user sees the parameter as 0..768, the effect sees it as 0..1. So if the plug-in wanted to set the default value of an effect to be the centre of the image, it would flag a 2D parameter as normalised and set the defaults to be 0.5. The user would see this in the centre of the image, no matter the resolution of the project in question. The plugin would retrieve the parameter as 0..1 and scale it up to the project size to size to use.

Parameters can choose to be normalised in several ways…

- ***kOfxParamDoubleTypeNormalisedX*** normalised size wrt to the project's X dimension (1D only),

- **`kOfxParamDoubleTypeNormalisedXAbsolute`** normalised absolute position on the X axis (1D only)
- **`kOfxParamDoubleTypeNormalisedY`** normalised size wrt to the project's Y dimension(1D only),
- **`kOfxParamDoubleTypeNormalisedYAbsolute`** normalised absolute position on the Y axis (1D only)
- **`kOfxParamDoubleTypeNormalisedXY`** normalised to the project's X and Y size (2D only),
- **`kOfxParamDoubleTypeNormalisedXYAbsolute`** normalised to the projects X and Y size, and is an absolute position on the image plane.

See the section on coordinate systems on how to scale between normalised, canonical and pixel coordinates.

1.12.30 Double Parameters Defaults, Increments, Mins and Maxs

In all cases double parameters' defaults, minimums and maximums are specified in the same space as the parameter, as is the increment in all cases but normalised parameters.

Normalised parameters specify their increments in canonical coordinates, rather than in normalised coordinates. So an increment of '1' means 1 pixel, not '1 project width', otherwise sliders would be a bit wild.

1.12.31 Parametric Parameters

Parametric params are new for 1.2 and are optionally supported by host applications. They are specified via the `kOfxParamTypeParametric` identifier passed into `OfxParameterSuiteV1::paramDefine()`

These parameters are somewhat more complex than normal parameters and require their own set of functions to manage and manipulate them. The new `OfxParametricParameterSuiteV1` is there to do that.

All the defines and suite definitions for parameteric parameters are defined in the file ofxParametricParam.h

Parametric parameters are in effect *functions* a plug-in can ask a host to arbitrarily evaluate for some value *x*. A classic use case would be for constructing look-up tables, a plug-in would ask the host to evaluate one at multiple values from 0 to 1 and use that to fill an array.

A host would probably represent this to a user as a cubic curve in a standard curve editor interface, or possibly through scripting. The user would then use this to define the 'shape' of the parameter.

The evaluation of such params is not the same as animation, they are returning values based on some arbitrary argument orthogonal to time, so to evaluate such a param, you need to pass a parametric position and time.

Often, you would want such a parametric parameter to be multi-dimensional, for example, a colour look-up table might want three values, one for red, green and blue. Rather than declare three separate parametric parameters, so a parametric parameter can be multi-dimensional.

Due to the nature of the underlying data, you *cannot* call certain functions in the ordinary parameter suite when manipulating a parametric parameter. All functions in the standard parameter suite are valid when called on a parametric parameter, with the exception of the following. . . .

- `OfxParameterSuiteV1::paramGetValue()`
- `OfxParameterSuiteV1::paramGetValueAtTime()`
- `OfxParameterSuiteV1::paramGetDerivative()`
- `OfxParameterSuiteV1::paramGetIntegral()`
- `OfxParameterSuiteV1::paramSetValue()`
- `OfxParameterSuiteV1::paramSetValueAtTime()`

Parametric parameters are defined using the standard parameter suite function `OfxParameterSuiteV1::paramDefine()`. The descriptor returned by this call have several non standard parameter properties available. These are

- **`kOfxParamPropParametricDimension`** the dimension of the parametric parameter,
- **`kOfxParamPropParametricUIColour`** the colour of the curves of a parametric parameter in any user

  interface
- **`kOfxParamPropParametricInteractBackground`** a pointer to an interact entry point, which will be used to draw a

  background under any user interface,
- **`kOfxParamPropParametricRange`** the min and max value that the parameter will be evaluated over.

Animation is an optional host feature for parametric parameters. Hosts flag whether they support this feature by setting the host descriptor property `kOfxParamHostPropSupportsParametricAnimation`.

Seeing as we need to pass in the parametric position and dimension to evaluate, parametric parameters need a new evaluation mechanism. They do this with the `OfxParametricParameterSuiteV1::parametricParamGetValue()` function. This function returns the value of the parameter at the given time, for the given dimension, adt the given parametric position,.

Parametric parameters are effectively interfaces to some sort of host based curve library. To get/set/delete points in the curve that represents a parameter, the new suite has several functions available to manipulate control points of the underlying curve.

To set the default value of a parametric parameter to anything but the identity, you use the control point setting functions in the new suite to set up a curve on the *descriptor* returned by `OfxParameterSuiteV1::paramDefine()`. Any instances later created, will have that curve as a default.

This simple example defines a colour lookup table, defines a default, and show how to evaluate the curve

```
// describe our parameter in
static OfxStatus
describeInContext( OfxImageEffectHandle  effect,  OfxPropertySetHandle inArgs)
{
  ....
  // define it
  OfxPropertySetHandle props;
  gParamHost->paramDefine(paramSet, kOfxParamTypeParametric, "lookupTable", &␣
↪props);

  // set standard names and labeles
  gPropHost->propSetString(props, kOfxParamPropHint, 0, "Colour lookup table");
  gPropHost->propSetString(props, kOfxParamPropScriptName, 0, "lookupTable");
  gPropHost->propSetString(props, kOfxPropLabel, 0, "Lookup Table");

  // define it as three dimensional
  gPropHost->propSetInt(props, kOfxParamPropParametricDimension, 0, 3);

  // label our dimensions are r/g/b
  gPropHost->propSetString(props, kOfxParamPropDimensionLabel, 0, "red");
  gPropHost->propSetString(props, kOfxParamPropDimensionLabel, 1, "green");
  gPropHost->propSetString(props, kOfxParamPropDimensionLabel, 2, "blue");

  // set the UI colour for each dimension
```

(continues on next page)

(continued from previous page)

```
  for(int component = 0; component < 3; ++component) {
     gPropHost->propSetDouble(props, kOfxParamPropParametricUIColour,␣
↪component * 3 + 0, component % 3 == 0 ? 1 : 0);
     gPropHost->propSetDouble(props, kOfxParamPropParametricUIColour,␣
↪component * 3 + 1, component % 3 == 1 ? 1 : 0);
     gPropHost->propSetDouble(props, kOfxParamPropParametricUIColour,␣
↪component * 3 + 2, component % 3 == 2 ? 1 : 0);
  }

  // set the min/max parametric range to 0..1
  gPropHost->propSetDouble(props, kOfxParamPropParametricRange, 0, 0.0);
  gPropHost->propSetDouble(props, kOfxParamPropParametricRange, 1, 1.0);

  // set a default curve, this example sets an invert
  OfxParamHandle descriptor;
  gParamHost->paramGetHandle(paramSet, "lookupTable", &descriptor, NULL);
  for(int component = 0; component < 3; ++component) {
    // add a control point at 0, value is 1
    gParametricParamHost->parametricParamAddControlPoint(descriptor,
                                                        component, // curve␣
↪to set
                                                        0.0,   // time,␣
↪ignored in this case, as we are not adding a ket
                                                        0.0,   // parametric␣
↪position, zero
                                                        1.0,   // value to␣
↪be, 1
                                                        false);   // don't␣
↪add a key
    // add a control point at 1, value is 0
    gParametricParamHost->parametricParamAddControlPoint(descriptor, component,
↪ 0.0, 1.0, 0.0, false);
  }

  ...
}

void render8Bits(double currentFrame, otherStuff...)
{
   ...

   // make three luts from our curves
   unsigned char lut[3][256];

  OfxParamHandle param;
  gParamHost->paramGetHandle(paramSet, "lookupTable", &param, NULL);
  for(int component = 0; component < 3; ++component) {
    for(int position = 0; position < 256; ++position) {
      // position to evaluate the param at
      float parametricPos = float(position)/255.0f;

      // evaluate the parametric param
```

(continues on next page)

(continued from previous page)

```
      float value;
      gParametricParamHost->parametricParamGetValue(param, component,␣
↪currentFrame, parametricPos, &value);
      value = value * 255;
      value = clamp(value, 0, 255);

      // set that in the lut
      lut[dimension][position] = (unsigned char)value;
    }
  }
  ...
}
```

1.12.32 Setting Parameters

Plugins are free to set parameters in limited set of circumstances, typically relating to user interaction. You can only set parameters in the following actions passed to the plug-in's main entry function…

- *kOfxActionCreateInstance*
- *kOfxActionBeginInstanceChanged*
- *kOfxActionInstanceChanged*
- *kOfxActionEndInstanceChanged*
- *kOfxActionSyncPrivateData*

Plugins can also set parameter values during the following actions passed to any of its interacts main entry function:

- *kOfxInteractActionPenDown*
- *kOfxInteractActionPenMotion*
- *kOfxInteractActionPenUp*
- *kOfxInteractActionKeyDown*
- *kOfxInteractActionKeyRepeat*
- *kOfxInteractActionKeyUp*
- *kOfxInteractActionLoseFocus*

1.13 Rendering

The *kOfxImageEffectActionRender* action is passed to plugins, when the host requires them to render an output frame.

All calls to the *kOfxImageEffectActionRender* are bracketed by a pair of *kOfxImageEffectActionBeginSequenceRender* and *kOfxImageEffectActionEndSequenceRender* actions. This is to allow plugins to prepare themselves for rendering long sequences by setting up any tables etc.. it may need.

The *kOfxImageEffectActionBeginSequenceRender* will indicate the frame range that is to be renderred, and whether this is purely a single frame render due to interactive feedback from a user in a GUI.

The render action is used in conjunction with the optional

1.13.1 Identity Effects

If an effect does nothing to its input clips (for example a blur with blur size set to '0') it can indicate that it is an identity function via the *kOfxImageEffectActionIsIdentity* action. The plugin indicates which input the host should use for the region in question. This allows a host to short circuit the processing of an effect.

1.13.2 Rendering and The Get Region Actions

Many hosts attempt to minimise the areas that they render by using regions of interest and regions of definition, while some of the simpler hosts do not attempt to do so. In general the order of actions, per frame rendered, is something along the lines of. . . .

- ask the effect for it's region of definition,
- clip the render window against that
- ask the effect for the regions of interest of each of it's inputs against the clipped render window,
- clip those regions of interest against the region of definition of each of those inputs,
- render and cache each of those inputs,
- render the effect against it's clipped render window.

A host can ask an effect to render an arbitrary window of pixels, generally these should be clipped to an effect's region of definition, however, depending on the host, they may not be. The actual region to render is indicated by the *kOfxImageEffectPropRenderWindow* render action argument. If an effect is asked to render outside of its region of definition, it should fill those pixels in with black transparent pixels.

Note thate *OfxImageEffectSuiteV1::clipGetImage()* function takes an optional *region* parameter. This is a region, in Canonical coordinates, that the effect would like on that input clip. If not used in a render action, then the image returned should be based on the previous get region of interest action. If used, then the image returned will be based on this (usually be clipped to the input's region of definition). Generally a plugin should not use the *region* parameter in the render action, but should leave it to the 'default' region.

1.13.3 Multi-threaded Rendering

Multiple render actions may be passed to an effect at the same time. A plug-in states it's level of render thread safety by setting the *kOfxImageEffectPluginRenderThreadSafety* string property. This can be set to one of three states. . . .

kOfxImageEffectRenderUnsafe

String used to label render threads as un thread safe, see, *kOfxImageEffectPluginRenderThreadSafety*.

Indicating that only a single 'render' action can be made at any time among all instances

kOfxImageEffectRenderInstanceSafe

String used to label render threads as instance thread safe, *kOfxImageEffectPluginRenderThreadSafety*.

Indicating that any instance can have a single 'render' action at any one time

kOfxImageEffectRenderFullySafe

String used to label render threads as fully thread safe, *kOfxImageEffectPluginRenderThreadSafety*.

Indicating that any instance of a plugin can have multiple renders running simultaneously

Rendering in a Symmetric Multi Processing Environment

When rendering on computers that have more that once CPU (or this new-fangled hyperthreading), hosts and effects will want to take advantage of all that extra CPU goodness to speed up rendering. This means multi-threading of the render function in some way.

If the plugin has set `kOfxImageEffectPluginRenderThreadSafety` to `kOfxImageEffectRenderFullySafe`, the host may choose to render a single frame across multiple CPUs by having each CPU render a different window. However, the plugin may wish to remain in charge of multithreading a single frame. The plugin set property `kOfxImageEffectPluginPropHostFrameThreading` informs the host as to whether the host should perform SMP on the effect. It can be set to either…

- 1, in which case the host will attempt to multithread an effect instance by calling it's render function called simultaneously, each call will be with a different renderWindow, but be at the same frame
- 0, in which case the host only ever calls the render function once per frame. If the effect wants to multithread it must use the OfxMultiThreadSuite API.

A host may have a render farm of computers. Depending exactly how the host works with it's render farm, it may have multiple copies on an instance spread over the farm rendering separate frame ranges, 1-100 on station A, 101 to 200 on station B and so on…

Rendering Sequential Effects

Some plugins need the output of the previous frame to render the next, typically they cache some information about the last render and use that somehow on the next frame. Some temporally averaging degraining algorithms work that way. Such effects cannot render correctly unless they are strictly rendered in order, from first to last frame, on a single instance.

Other plugins are able to render correctly when called in an arbitrary frame order, but render much more efficiently if rendered in order. For example a particle system which maintains the state of the particle system in an instance would simply increment the simulation by a frame if rendering in-order, but would need to restart the particle system from scratch if the frame jumped backwards.

Most plug-ins do not have any sequential dependence. For example, a simple gain operation has no dependence on the previous frame.

Similarly, host applications, due to their architectures, may or may not be able to guarantee that a plugin can be rendered strictly in-order. Node based applications typically have much more difficulty in guaranteeing such behaviour.

To indicate whether a plugin needs to be rendered in a strictly sequential order, and to indicate whether a host supports such behaviour we have a property, `kOfxImageEffectInstancePropSequentialRender`. For plug-ins this can be one of three values…

- 0, in which case the host can render an instance over arbitrary frame ranges on an arbitrary number of computers without any problem (default),
- 1, in which case the host must render an instance on a single computer over it's entire frame range, from first to last.
- 2, in which case the effect is more efficiently rendered in frame order, but can compute the correct result regardless of render order.

For hosts, this property takes three values…

- 0, which indicates thet the host can never guarantee sequential rendering,
- 1, which indicates thet the host can guarantee sequential rendering for plugins that request it,
- 2, which indicates thet the host can sometimes perform sequential rendering.

When rendering, a host will set the in args property on *kOfxImageEffectPropSequentialRenderStatus* to indicate whether the host is currently supporting sequential renders. This will be passed to the following actions,

- the begin sequence render action
- the sequence render action
- the end sequence render action

Hosts may still render sequential effects with random frame access in interactive sessions, for example when the user scrubs the current frame on the timeline and the host asks an effect to render a preview frame. In such cases, the plugin can detect that the instance is being interactively manipulated via the *kOfxImageEffectPropInteractiveRenderStatus* property and hack an approximation together for UI purposes. If eventually rendering the sequence, the host *must* ignore all frames rendered out of order and not cache them for use in the final result.

A host may set the in args property *kOfxImageEffectPropRenderQualityDraft* in :c:macro:kOfxImageEffectActionRender` to ask for a render in Draft/Preview mode. This is useful for applications that must support fast scrubbing. These allow a plug-in to take short-cuts for improved performance when the situation allows and it makes sense, for example to generate thumbnails with effects applied. For example switch to a cheaper interpolation type or rendering mode. A plugin should expect frames rendered in this manner that will not be stuck in host cache unless the cache is only used in the same draft situations.

1.13.4 OFX : Fields and Field Rendering

Fields are evil, but until the world decides to adopt sensible video standard and casts the current ones into the same pit as 2 inch video tape, we are stuck with them.

Before we start, some nomenclature. The Y-Axis is considerred to be up, so in a fielded image,

- even scan lines 0,2,4,6,… are collectively referred to as the lower field,
- odd scan lines 1,3,5,7… are collective referred to as the upper field.

We don't call them odd and even, so as to avoid confusion with video standard, which have scanline 0 at the top, and so have the opposite sense of our 'odd' and 'even'.

Clips and images from those clips are flagged as to whether they are fielded or not, and if so what is the spatial/temporal ordering of the fields in that image. The *kOfxImageClipPropFieldOrder* clip and image instance property can be…

kOfxImageFieldNone

String used to label imagery as having no fields

The material is unfielded

kOfxImageFieldLower

String used to label the lower field (scan lines 0,2,4…) of fielded imagery

The material is fielded, with scan line 0,2,4…. occurring first in a frame

kOfxImageFieldUpper

String used to label the upper field (scan lines 1,3,5…) of fielded imagery

The material is fielded, with scan line 1,3,5…. occurring first in a frame

Images extracted from a clip flag what their fieldedness is with the property *kOfxImagePropField*, this can be….

kOfxImageFieldNone

String used to label imagery as having no fields

The image is an unfielded frame

kOfxImageFieldBoth

String used to label both fields of fielded imagery, indicating interlaced footage

The image is fielded and contains both interlaced fields

kOfxImageFieldLower

String used to label the lower field (scan lines 0,2,4…) of fielded imagery

The image is fielded and contains a single field, being the lower field (lines 0,2,4…)

kOfxImageFieldUpper

String used to label the upper field (scan lines 1,3,5…) of fielded imagery

The image is fielded and contains a single field, being the upper field (lines 1,3,5…)

The plugin specifies how it deals with fielded imagery by setting the *kOfxImageEffectPluginPropFieldRenderTwiceAlways* property. This can be,

- 0 - the plugin is to have it's render function called twice only if there is animation in any of it's parameters
- 1 - the plugin is to have it's render function called twice always (default)

The reason for this is an optimisation. Imagine a text generator with no animation being asked to render into a fielded output clip, it can treat an interlaced fielded image as an unfielded frame. So the host can get the effect to render both fields in one hit and save on the overhead required to do the rendering in two passes.

If called twice per frame, the time passed to the render action will be frame and frame+0.5. So 0.0 0.5 1.0 1.5 etc…

When rendering unfielded footage, the host will only ever call the effect's render action once per frame, with the time being at the integers, 0.0, 1.0, 2.0 and so on.

The render action's argument property *kOfxImageEffectPropFieldToRender* tells the effect which field it should render, this can be one of…

- *kOfxImageFieldNone* - there are no fields to deal with, the image is full frame
- *kOfxImageFieldBoth* - the imagery is fielded and both scan lines should be renderred
- *kOfxImageFieldLower* - the lower field is being rendered (lines 0,2,4…)
- *kOfxImageFieldUpper* - the upper field is being rendered (lines 1,3,5…)

Note: *kOfxImageEffectPropFieldToRender* will be set to *kOfxImageFieldBoth* if *kOfxImageEffectPluginPropFieldRenderTwiceAlways* is set to 0 on the plugin

A plugin can specify how it wishes fielded footage to be fetched from a clip via the clip descriptor property *kOfxImageClipPropFieldExtraction*. This can be one of…

- *kOfxImageFieldBoth*

Fetch a full frame interlaced image

- `kOfxImageFieldSingle`

Fetch a single field, making a half height image

- `kOfxImageFieldDoubled`

Fetch a single field, but doubling each line and so making a full height image (default)

If fetching a single field, the actual field fetched from the source frame is…

- the first temporal field if the time passed to clipGetImage has a fractional part of 0.0 <= f < 0.5
- the second temporal field otherwise,

To illustrate this last behaviour, the two examples below show an output with twice the frame rate of the input and how clipGetImage maps to the input. The .0 and .5 mean first and second temporal fields.

```
Behaviour with unfielded footage

output 0       1       2       3
source 0       0       1       1
```

```
Behaviour with fielded footage

output 0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5
source 0.0 0.0 0.5 0.5 1.0 1.0 1.5 1.5
```

NOTE

- while some rarely used video standards can have odd number of scan-lines, under OFX, both fields always consist of the same number of lines. Pad with black where needed.
- host developers, for single field extracted images, you don't need to do any buffer copies, you just need to set the row bytes property of the returned image to twice the normal value, and maybe tweak the start address by a scanline.

1.13.5 Rendering In An Interactive Environment

Any host with an interface will most likely have an interactive thread and a rendering thread. This allows an effect to be manipulated while having renders batched off to a background thread. This will mean that some degree of locking will go on to prevent simultaneous read/writes occurring, see this section for more on thread safety.

A host may need to abort a backgrounded render, typically in response to a user changing a parameter value. An effect should occasionally poll the *`OfxImageEffectSuiteV1::abort()`* function to see if it should give up on rendering.

1.14 Interacts

When a host presents a graphical user interface to an image effect, it may optionally give it the chance to draw its own custom GUI tools and to be able to interact with pen and keyboard input. In OFX this is done via the OfxInteract suite, which is found in the file ofxInteract.h.

OFX interacts by default use openGL to perform all drawing in interacts, due to its portabilty, robustness and wide implementation. As of 2022, some systems are moving away from OpenGL support in favor of more modern graphics drawing APIs. So as of OFX 1.5, effects may use the *`OfxDrawSuiteV1`* instead of OpenGL if the host supports it.

Each object that can have their own interact a pointer property in it which should point to a separate *main entry point*. This entry point is *not* the same as the one in the OfxPlugin struct, as it needs to respond to a different set of actions to the effect.

There are two things in an image effect can have their own interact, these are…

- as on overlay on the image being currently viewed in any image viewer, set via the effect descriptor's *`kOfxImageEffectPluginPropOverlayInteractV1`* property

- as a replacement for any parameter's standard GUI object, set this via the parameter descriptor's

  `kOfxParamPropInteractV1`

  property

Hosts might not be able to support interacts, to indicate this, two properties exist on the host descriptor which an effect should examine at description time so as to determine its own behaviour. These are…

- `kOfxImageEffectPropSupportsOverlays`
- `kOfxParamHostPropSupportsCustomInteract`

Interacts are separate objects to the effect they are associated with, they have their own descriptor and instance handles passed into their separate *main entry point*.

An interact instance cannot exist without a plugin instance, an interact's instance, once created, is bound to a single instance of a plugin until the interact instance is destroyed.

All interacts of the same type share openGL display lists, even if they are in different openGL contexts.

All interacts of the same type will have the same pixel types (this is a side effect of the last point), this will always be double buffered with at least RGB components. Alpha and the exact bit depth is left to the implementation.

So for example, all image effect overlays share the same display lists and have the same pixel depth, and all custom parameter GUIs share the same display list and have the same pixel depth, but overlays and custom parameter GUIs do not necassarily share the same display list/pixel depths.

An interact instance may be used in more than one view. Consider an image effect overlay interact in a host that supports multiple viewers to an effect instance. The same interact instance will be used in all views, the relevant properties controlling the view being changed before any action is passed to the interact. In this example, the draw action would be called once for each view open on the instance, with the projection, viewport and pixel scale being set appropriately for the view before each action.

1.14.1 Overlay Interacts

Hosts will generally display images (both input and output) in user their interfaces. A plugin can put an interact in this display by setting the effect descriptor `kOfxImageEffectPluginPropOverlayInteractV1` property to point to a main entry.

The viewport for such interacts will depend completely on the host.

The `GL_PROJECTION` matrix will be set up so that it maps openGL coordinates to canonical image coordinates.

The `GL_MODELVIEW` matrix will be the identity matrix.

An overlay's interact draw action should assume that it is sharing the openGL context and viewport with other objects that belong to the host. It should not blank the background and it should never swap buffers, that is for the host to do.

1.14.2 Parameter Interacts

All parameters, except for custom parameters, have some default interface that the host creates for them. Be it a numeric slider, colour swatch etc… Effects can override the default interface (or set an interface for a custom parameter) by setting the `kOfxParamPropInteractV1`. This will completely replace the parameters default user interface in the 'paged' and *hierarchical* interfaces, but it will not replace the parameter's interface in any animation sheet.

Properties affecting custom interacts for parameters are…

- `kOfxParamPropInteractSizeAspect`
- `kOfxParamPropInteractMinimumSize`

- *kOfxParamPropInteractPreferedSize*

The viewport for such interacts will be dependent upon the various properties above, and possibly a per host override in any XML resource file.

The `GL_PROJECTION` matrix will be an orthographic 2D view with -0.5,-0.5 at the bottom left and viewport width-0.5, viewport height-0.5 at the top right.

The `GL_MODELVIEW` matrix will be the identity matrix.

The bit depth will be double buffered 24 bit RGB.

A parameter's interact draw function will have full responsibility for drawing the interact, including clearing the background and swapping buffers.

1.14.3 Interact Actions

The following actions are passed to any interact entry point in an image effect plug-in.

- The Generic Describe Action called to describe the specific interact ,
- The Create Instance Action called just after an instance of the interact is created,
- The Generic Destroy Instance Action called just before of the interact is destroyed,
- The Draw Action called to have the interact draw itself,
- *kOfxInteractActionPenMotion* called whenever the interact has the input focus and the pen has moved, regardless of whether the pen is up or down,
- *kOfxInteractActionPenDown* called whenever the interact has the input focus and the pen has changed state to 'down',
- *kOfxInteractActionPenUp* called whenever the interact has the input focus and the pen has changed state to 'up,
- *kOfxInteractActionKeyDown* called whenever the interact has the input focus and a key has gone down,
- *kOfxInteractActionKeyUp* called whenever the interact has the input focus and a key has gone up,
- *kOfxInteractActionKeyRepeat* called whenever the interact has the input focus and a key has gone down and a repeat key sequence has been sent,
- *kOfxInteractActionGainFocus* called whenever the interact gains input focus,
- *kOfxInteractActionLoseFocus* called whenever the interact loses input focus,

An interact cannot be described until an effect has been described.

An interact instance must always be associated with an effect instance. So it gets created after an effect and destroyed before one.

An interact instance should be issued a gain focus action before any key or pen actions are issued, and a lose focus action when it goes.

1.15 Image Effect Clip Preferences

The *kOfxImageEffectActionGetClipPreferences* action is passed to an effect to allow a plugin to specify how it wishes to deal with its input clips and to set properties in its output clip. This is especially important when there are multiple inputs which may have differing properties such as pixel depth and number of channels.

More specifically, there are six properties that can be set during the clip preferences action, some on the input clip, some on the output clip, some on both. These are:

- the depth of a clip's pixels, input or output clip
- the components of a clip's pixels, input or output clip
- the pixel aspect ratio of a clip, input or output clip
- the frame rate of the output clip
- the fielding of the output clip
- the premultiplication state of the output clip
- whether the output clip varys from frame to frame, even if no parameters or input images change over time
- whether the output clip can be sampled at sub-frame times and produce different images

The behaviour specified by OFX means that a host may need to cast images from their native data format into one suitable for the plugin. It is better that the host do any of this pixel shuffling because:

- the behaviour is orthogonal for all plugins on that host
- the code is not replicated in all plugins
- the host can optimise the pixel shuffling in one pass with any other data grooming it may need to do

A plugin gets to assert its clip preferences in several situations. Firstly whenever a clip is attached to a plugin, secondly whenever one of the parameters in the plugin property *kOfxImageEffectPropClipPreferencesSlaveParam* has its value changed. The clip preferences action is never called until all non-optional clips have been attached to the plugin.

Note:

- these properties cannot animate over the duration of an effect
- that the ability to set input and output clip preferences is restricted by the context of an effect
- optional input clips do not have any context specific restrictions on plugin set preferences

1.15.1 Frame Varying Effects

Some plugins can generate differing output frames at different times, even if no parameters animate or no input images change. The *kOfxImageEffectFrameVarying* property set in the clip preferences action is used to flag this.

A counterexample is a solid colour generator. If it has no animating parameters, the image generated at frame 0 will be the same as the image generated at any other frame. Intelligent hosts can render a single frame and cache that for use at all other times.

On the other hand, a plugin that generates random noise at each frame and seeds its random number generator with the render time will create different images at different times. The host cannot render a single frame and cache that for use at subsequent times.

To differentiate between these two cases the *kOfxImageEffectFrameVarying* is used. If set to 1, it indicates that the effect will need to be rendered at each frame, even if no input images or parameters are varying. If set to 0, then a single frame can be rendered and used for all times if no input images or parameters vary. The default value is 0.

1.15.2 Continuously Sampled Effects

Some effects can generate images at non frame-time boundaries, even if the inputs to the effect are frame based and there is no animation.

For example a fractal cloud generator whose pattern evolves with a speed parameter can be rendered at arbitrary times, not just on frame boundaries. Hosts that are interested in sub-frame rendering can determine that the plugin supports this behaviour by examining the *kOfxImageClipPropContinuousSamples* property set in the clip preferences action. By default this is `false`.

Note: Implicitly, all retimers effects can be continuously sampled.

1.15.3 Specifying Pixel Depths

Hosts and plugins flag whether whether they can deal with input/output clips of differing pixel depths via the *kOfxImageEffectPropSupportsMultipleClipDepths* property.

If the host sets this to 0, then all effect's input and output clips will always have the same component depth, and the plugin may not remap them.

If the plugin sets this to 0, then the host will transparently map all of an effect's input and output clips to a single depth, even if the actual clips are of differing depths. In the above two cases, the common component depth chosen will be the deepest depth of any input clip mapped to a depth the plugin supports that loses the least precision. E.g.: if a plugin supported 8 bit and float images, but the deepest clip attached to it was 16 bit, the host would transparently map all clips to float.

If both the plugin and host set this to 1, then the plugin can, during the *kOfxImageEffectActionGetClipPreferences*, specify how the host is to map each clip, including the output clip. Note that this is the only case where a plugin may set the output depth.

The bitdepth must be one of:

- **`kOfxBitDepthByte`**
  String used to label unsigned 8 bit integer samples.
- **`kOfxBitDepthShort`**
  String used to label unsigned 16 bit integer samples.
- **`kOfxBitDepthHalf`**
  String used to label half-float (16 bit floating point) samples.

  Version Added in Version 1.4. Was in ofxOpenGLRender.h before.
- **`kOfxBitDepthFloat`**
  String used to label signed 32 bit floating point samples.

- `kOfxBitDepthNone`
  String used to label unset bitdepths.

1.15.4 Specifying Pixel Components

A plugin specifies what components it is willing to accept on a clip via the *kOfxImageEffectPropSupportedComponents* on the clip's descriptor during the *kOfxImageEffectActionDescribeInContext* This is one or more of:

- `kOfxImageComponentRGBA`
  String to label images with RGBA components.
- `kOfxImageComponentRGB`
  String to label images with RGB components.
- `kOfxImageComponentAlpha`
  String to label images with only Alpha components.
- `kOfxImageComponentNone`
  String to label something with unset components.

If an effect has multiple inputs, and each can be a range of component types, the effect may end up with component types that are incompatible for its purposes. In this case the effect will want to have the host remap the components of the inputs and to specify the components in the output.

For example, a general effect that blends two images will have have two inputs, each of which may be RGBA or A. In operation, if presented with RGBA on one and A on the other, it will most likely request that the A clip be mapped to RGBA by the host and the output be RGBA as well.

In all contexts, except for the general context, mandated input clips cannot have their component types remapped, nor can the output. Optional input clips can always have their component types remapped.

In the general context, all input clips may be remapped, as can the output clip. The output clip has its default components set to be:

- RGBA if any of the inputs is RGBA
- otherwise A if the effect has any inputs
- otherwise RGBA if there are no inputs.

Note: It is a host implementation detail as to how a host actually attaches real clips to a plugin. However it must map the clip to RGBA in a manner that is transparent to the plugin. Similarly for any other component types that the plugin does not support on an input.

1.15.5 Specifying Pixel Aspect Ratios

Hosts and plugins flag whether whether they can deal with input/output clips of differing pixel aspect ratios via the `kOfxImageEffectPropSupportsMultipleClipPARs` property.

If the host sets this to 0, then all effect's input and output clips will always have the same pixel aspect ratio, and the plugin may not remap them.

If the plugin sets this to 0, then the host will transparently map all of an effect's input and output clips to a single pixel aspect ratio, even if the actual clips are of differring PARs.

In the above two cases, the common pixel aspect ratio chosen will be the smallest on all the inputs, as this preserves image data.

If *both* the plugin and host set this to 1, then the plugin can, during `kOfxImageEffectActionGetClipPreferences`, specify how the host is to map each clip, including the output clip.

1.15.6 Specifying Fielding

The `kOfxImageEffectPropSetableFielding` host property indicates if a plugin is able to change the fielding of the output clip from the default.

The default value of the output clip's fielding is host dependent, but in general,

- if any of the input clips are fielded, so will the output clip
- the output clip may be fielded irregardless of the input clips (for example, in a fielded project).

If the host allows a plugin to specify the fielding of the output clip, then a plugin may do so during the `kOfxImageEffectActionGetClipPreferences` by setting the property `kOfxImageClipPropFieldOrder` in the out args argument of the action. For example a defielding plugin will want to indicate that the output is frame based rather than fielded.

1.15.7 Specifying Frame Rates

The `kOfxImageEffectPropSetableFrameRate` host property indicates if a plugin is able to change the frame rate of the output clip from the default.

The default value of the output clip's frame rate is host dependent, but in general, it will be based on the input clips' frame rates.

If the host allows a plugin to specify the frame rate of the output clip, then a plugin may do so during the `kOfxImageEffectActionGetClipPreferences`. For example a deinterlace plugin that separates both fields from fielded footage will want to double the frame rate of the output clip.

If a plugin changes the frame rate, it is effectively changing the number of frames in the output clip. If our hypothetical deinterlace plugin doubles the frame rate of the output clip, it will be doubling the number of frames in that clip. The timing diagram below should help, showing how our fielded input has been turned into twice the number of frames on output.

```
FIELDED SOURCE       0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 ....
DEINTERLACED OUTPUT  0   1   2   3   4   5   6   7   8   9
```

The mapping of the number of output frames is as follows:

```
nFrames' = nFrames * FPS' / FPS
```

- `nFrames` is the default number of frames,

- `nFrames'` is the new number of output frames,
- `FPS` is the default frame rate,
- `FPS'` is the new frame rate specified by a plugin.

1.15.8 Specifying Premultiplication

All clips have a premultiplication state (see this for a nice explanation). An effect cannot map the premultiplication state of the input clips, but it can specify the premultiplication state of the output clip via *`kOfxImageEffectPropPreMultiplication`*, setting that to *`kOfxImagePreMultiplied`* or *`kOfxImageUnPreMultiplied`*.

The output's default premultiplication state is...

- premultiplied if any of the inputs are premultiplied
- otherwise unpremultiplied if any of the inputs are unpremultiplied
- otherwise opaque

1.16 Actions Passed to An Image Effect

Actions passed to an OFX Image Effect's plug-in main entry point are from two categories...

- actions that could potentially be issued to any kind of plug in, not just image effects, known as generic actions, found in

  ofxCore.h

- actions that are only applicable purely to image effects, found in ofxImageEffect.h

For generic actions, the `handle` passed to to main entry point will depend on the API being implemented, for all generic actions passed to an OFX Image Effect plug-in, it will nearly always be an *`OfxImageEffectHandle`*.

Because interacts are a special case, they are dealt with in a separate chapter, this chapter will deal with actions issued to an image effect plug-ins main entry point.

`kOfxActionLoad`

This action is the first action passed to a plug-in after the binary containing the plug-in has been loaded. It is there to allow a plug-in to create any global data structures it may need and is also when the plug-in should fetch suites from the host.

The handle, inArgs and outArgs arguments to the mainEntry are redundant and should be set to NULL.

Pre

- The plugin's OfxPlugin::setHost function has been called

Post This action will not be called again while the binary containing the plug-in remains loaded.

Returns

- *kOfxStatOK*, the action was trapped and all was well,
- *kOfxStatReplyDefault*, the action was ignored,
- *kOfxStatFailed*, the load action failed, no further actions will be passed to the plug-in. Interpret if possible kOfxStatFailed as plug-in indicating it does not want to load Do not create an

entry in the host's UI for plug-in then. Plug-in also has the option to return 0 for OfxGetNumberOfPlugins or kOfxStatFailed if host supports OfxSetHost in which case kOfxActionLoad will never be called.

- *kOfxStatErrFatal*, fatal error in the plug-in.

`kOfxActionUnload`

This action is the last action passed to the plug-in before the binary containing the plug-in is unloaded. It is there to allow a plug-in to destroy any global data structures it may have created.

The handle, inArgs and outArgs arguments to the main entry are redundant and should be set to NULL.

- the *kOfxActionLoad* action has been called
- all instances of a plugin have been destroyed

Post

- No other actions will be called.

Returns

- *kOfxStatOK*, the action was trapped all was well
- *kOfxStatReplyDefault*, the action was ignored
- *kOfxStatErrFatal*, in which case we the program will be forced to quit

`kOfxActionDescribe`

The kOfxActionDescribe is the second action passed to a plug-in. It is where a plugin defines how it behaves and the resources it needs to function.

Note that the handle passed in acts as a descriptor for, rather than an instance of the plugin. The handle is global and unique. The plug-in is at liberty to cache the handle away for future reference until the plug-in is unloaded.

Most importantly, the effect must set what image effect contexts it is capable of working in.

This action *must* be trapped, it is not optional.

Parameters

- **`handle`** – handle to the plug-in descriptor, cast to an *OfxImageEffectHandle*
- **`inArgs`** – is redundant and is set to NULL
- **`outArgs`** – is redundant and is set to NULL

Pre

- *kOfxActionLoad* has been called

Post

- *kOfxActionDescribe* will not be called again, unless it fails and returns one of the error codes where the host is allowed to attempt the action again
- the handle argument, being the global plug-in description handle, is a valid handle from the end of a sucessful describe action until the end of the *kOfxActionUnload* action (ie: the plug-in can cache it away without worrying about it changing between actions).

- *kOfxImageEffectActionDescribeInContext* will be called once for each context that the host and plug-in mutually support. If a plug-in does not report to support any context supported by host, host should not enable the plug-in.

Returns

- *kOfxStatOK*, the action was trapped and all was well
- *kOfxStatErrMissingHostFeature*, in which the plugin will be unloaded and ignored, plugin may post message
- *kOfxStatErrMemory*, in which case describe may be called again after a memory purge
- *kOfxStatFailed*, something wrong, but no error code appropriate, plugin to post message
- *kOfxStatErrFatal*

`kOfxActionCreateInstance`

This action is the first action passed to a plug-in's instance after its creation. It is there to allow a plugin to create any per-instance data structures it may need.

- *kOfxActionDescribe* has been called
- the instance is fully constructed, with all objects requested in the describe actions (eg, parameters and clips) have been constructed and have had their initial values set. This means that if the values are being loaded from an old setup, that load should have taken place before the create instance action is called.

Parameters

- **`handle`** – handle to the plug-in instance, cast to an *OfxImageEffectHandle*
- **`inArgs`** – is redundant and is set to NULL
- **`outArgs`** – is redundant and is set to NULL

Post

- the instance pointer will be valid until the *kOfxActionDestroyInstance* action is passed to the plug-in with the same instance handle

Returns

- *kOfxStatOK*, the action was trapped and all was well
- *kOfxStatReplyDefault*, the action was ignored, but all was well anyway
- *kOfxStatErrFatal*
- *kOfxStatErrMemory*, in which case this may be called again after a memory purge
- *kOfxStatFailed*, something went wrong, but no error code appropriate, the plugin should to post a message if possible and the host should destroy the instanace handle and not attempt to proceed further

`kOfxActionDestroyInstance`

This action is the last passed to a plug-in's instance before its destruction. It is there to allow a plugin to destroy any per-instance data structures it may have created.

- *kOfxStatOK*, the action was trapped and all was well,

- *kOfxStatReplyDefault*, the action was ignored as the effect had nothing to do,
- *kOfxStatErrFatal*,
- *kOfxStatFailed*, something went wrong, but no error code appropriate, the plugin should to post a message.

Parameters

- `handle` – handle to the plug-in instance, cast to an *OfxImageEffectHandle*
- `inArgs` – is redundant and is set to NULL
- `outArgs` – is redundant and is set to NULL

Pre

- *kOfxActionCreateInstance* has been called on the handle,
- the instance has not had any of its members destroyed yet,

Post

- the instance pointer is no longer valid and any operation on it will be undefined

Returns To some extent, what is returned is moot, a bit like throwing an exception in a C++ destructor, so the host should continue destruction of the instance regardless.

`kOfxActionBeginInstanceChanged`

The *kOfxActionBeginInstanceChanged* and *kOfxActionEndInstanceChanged* actions are used to bracket all *kOfxActionInstanceChanged* actions, whether a single change or multiple changes. Some changes to a plugin instance can be grouped logically (eg: a 'reset all' button resetting all the instance's parameters), the begin/end instance changed actions allow a plugin to respond appropriately to a large set of changes. For example, a plugin that maintains a complex internal state can delay any changes to that state until all parameter changes have completed.

Parameters

- `handle` – handle to the plug-in instance, cast to an *OfxImageEffectHandle*
- `inArgs` – has the following properties
  - *kOfxPropChangeReason* what triggered the change, which will be one of…
  - *kOfxChangeUserEdited* - the user or host changed the instance somehow and caused a change to something, this includes undo/redos, resets and loading values from files or presets,
  - *kOfxChangePluginEdited* - the plugin itself has changed the value of the instance in some action
  - *kOfxChangeTime* - the time has changed and this has affected the value of the object because it varies over time
- `outArgs` – is redundant and is set to NULL

Post

- For *kOfxActionBeginInstanceChanged* , *kOfxActionCreateInstance* has been called on the instance handle.
- For *kOfxActionEndInstanceChanged* , *kOfxActionBeginInstanceChanged* has been called on the instance handle.
- *kOfxActionCreateInstance* has been called on the instance handle.

Post

- For *kOfxActionBeginInstanceChanged*, *kOfxActionInstanceChanged* will be called at least once on the instance handle.
- *kOfxActionEndInstanceChanged* will be called on the instance handle.

Returns

- *kOfxStatOK*, the action was trapped and all was well
- *kOfxStatReplyDefault*, the action was ignored
- *kOfxStatErrFatal*,
- *kOfxStatFailed*, something went wrong, but no error code appropriate, the plugin should to post a message

`kOfxActionEndInstanceChanged`

Action called after the end of a set of *kOfxActionEndInstanceChanged* actions, used with *kOfxActionBeginInstanceChanged* to bracket a grouped set of changes, see *kOfxActionBeginInstanceChanged*.

`kOfxActionInstanceChanged`

This action signals that something has changed in a plugin's instance, either by user action, the host or the plugin itself. All change actions are bracketed by a pair of *kOfxActionBeginInstanceChanged* and *kOfxActionEndInstanceChanged* actions. The `inArgs` property set is used to determine what was the thing inside the instance that was changed.

Parameters

- **`handle`** – handle to the plug-in instance, cast to an *OfxImageEffectHandle*
- **`inArgs`** – has the following properties
  - *kOfxPropType* The type of the thing that changed which will be one of..
  - kOfxTypeParameter Indicating a parameter's value has changed in some way
  - kOfxTypeClip A clip to an image effect has changed in some way (for Image Effect Plugins only)
  - *kOfxPropName* the name of the thing that was changed in the instance
  - *kOfxPropChangeReason* what triggered the change, which will be one of…
  - *kOfxChangeUserEdited* - the user or host changed the instance somehow and caused a change to something, this includes undo/redos, resets and loading values from files or presets,
  - *kOfxChangePluginEdited* - the plugin itself has changed the value of the instance in some action
  - *kOfxChangeTime* - the time has changed and this has affected the value of the object because it varies over time
  - *kOfxPropTime*
  - the effect time at which the chang occured (for Image Effect Plugins only)
  - *kOfxImageEffectPropRenderScale*
  - the render scale currently being applied to any image fetched from a clip (for Image Effect Plugins only)
- **`outArgs`** – is redundant and is set to NULL

Pre

- *kOfxActionCreateInstance* has been called on the instance handle,
- *kOfxActionBeginInstanceChanged* has been called on the instance handle.

Post

- *kOfxActionEndInstanceChanged* will be called on the instance handle.

Returns

- *kOfxStatOK*, the action was trapped and all was well
- *kOfxStatReplyDefault*, the action was ignored
- *kOfxStatErrFatal*,
- *kOfxStatFailed*, something went wrong, but no error code appropriate, the plugin should to post a message

`kOfxActionPurgeCaches`

This action is an action that may be passed to a plug-in instance from time to time in low memory situations. Instances recieving this action should destroy any data structures they may have and release the associated memory, they can later reconstruct this from the effect's parameter set and associated information.

For Image Effects, it is generally a bad idea to call this after each render, but rather it should be called after *kOfxImageEffectActionEndSequenceRender* Some effects, typically those flagged with the *kOfxImageEffectInstancePropSequentialRender* property, may need to cache information from previously rendered frames to function correctly, or have data structures that are expensive to reconstruct at each frame (eg: a particle system). Ideally, such effect should free such structures during the *kOfxImageEffectActionEndSequenceRender* action.

Parameters

- **`handle`** – handle to the plug-in instance, cast to an *OfxImageEffectHandle*
- **`inArgs`** – is redundant and is set to NULL
- **`outArgs`** – is redundant and is set to NULL

Pre

- *kOfxActionCreateInstance* has been called on the instance handle,

Returns

- *kOfxStatOK*, the action was trapped and all was well
- *kOfxStatReplyDefault*, the action was ignored
- *kOfxStatErrFatal*,
- *kOfxStatFailed*, something went wrong, but no error code appropriate, the plugin should to post a message

`kOfxActionSyncPrivateData`

This action is called when a plugin should synchronise any private data structures to its parameter set. This generally occurs when an effect is about to be saved or copied, but it could occur in other situations as well.

Parameters

- **`handle`** – handle to the plug-in instance, cast to an *OfxImageEffectHandle*
- **`inArgs`** – is redundant and is set to NULL
- **`outArgs`** – is redundant and is set to NULL

Pre

- *kOfxActionCreateInstance* has been called on the instance handle,

Post

- Any private state data can be reconstructed from the parameter set,

Returns

- *kOfxStatOK*, the action was trapped and all was well
- *kOfxStatReplyDefault*, the action was ignored
- *kOfxStatErrFatal*,
- *kOfxStatFailed*, something went wrong, but no error code appropriate, the plugin should to post a message

`kOfxActionBeginInstanceEdit`

This is called when an instance is *first* actively edited by a user, ie: and interface is open and parameter values and input clips can be modified. It is there so that effects can create private user interface structures when necassary. Note that some hosts can have multiple editors open on the same effect instance simulateously.

Parameters

- **`handle`** – handle to the plug-in instance, cast to an *OfxImageEffectHandle*
- **`inArgs`** – is redundant and is set to NULL
- **`outArgs`** – is redundant and is set to NULL

Pre

- *kOfxActionCreateInstance* has been called on the instance handle,

Post

- *kOfxActionEndInstanceEdit* will be called when the last editor is closed on the instance

Returns

- *kOfxStatOK*, the action was trapped and all was well
- *kOfxStatReplyDefault*, the action was ignored
- *kOfxStatErrFatal*,
- *kOfxStatFailed*, something went wrong, but no error code appropriate, the plugin should to post a message

`kOfxActionEndInstanceEdit`

This is called when the *last* user interface on an instance closed. It is there so that effects can destroy private user interface structures when necassary. Note that some hosts can have multiple editors open on the same effect instance simulateously, this will only be called when the last of those editors are closed.

Parameters

- **`handle`** – handle to the plug-in instance, cast to an *OfxImageEffectHandle*
- **`inArgs`** – is redundant and is set to NULL
- **`outArgs`** – is redundant and is set to NULL

Pre

- *kOfxActionBeginInstanceEdit* has been called on the instance handle,

Post

- no user interface is open on the instance

Returns

- *kOfxStatOK*, the action was trapped and all was well
- *kOfxStatReplyDefault*, the action was ignored
- *kOfxStatErrFatal*,
- *kOfxStatFailed*, something went wrong, but no error code appropriate, the plugin should to post a message

`kOfxImageEffectActionDescribeInContext`

This action is unique to OFX Image Effect plug-ins. Because a plugin is able to exhibit different behaviour depending on the context of use, each separate context will need to be described individually. It is within this action that image effects describe which parameters and input clips it requires.

This action will be called multiple times, one for each of the contexts the plugin says it is capable of implementing. If a host does not support a certain context, then it need not call *kOfxImageEffectActionDescribeInContext* for that context.

This action *must* be trapped, it is not optional.

Parameters

- **`handle`** – handle to the context descriptor, cast to an *OfxImageEffectHandle* this may or may not be the same as passed to *kOfxActionDescribe*
- **`inArgs`** – has the following property:
  - *kOfxImageEffectPropContext* the context being described
- **`outArgs`** – is redundant and is set to NULL

Pre

- *kOfxActionDescribe* has been called on the descriptor handle,
- *kOfxActionCreateInstance* has not been called

Returns

- *kOfxStatOK*, the action was trapped and all was well
- *kOfxStatErrMissingHostFeature*, in which the context will be ignored by the host, the plugin may post a message
- *kOfxStatErrMemory*, in which case the action may be called again after a memory purge
- *kOfxStatFailed*, something wrong, but no error code appropriate, plugin to post message
- *kOfxStatErrFatal*

`kOfxImageEffectActionGetRegionOfDefinition`

The region of definition for an image effect is the rectangular section of the 2D image plane that it is capable of filling, given the state of its input clips and parameters. This action is used to calculate the RoD for a plugin instance at a given frame. For more details on regions of definition see Image Effect Architectures.

Note that hosts that have constant sized imagery need not call this action, only hosts that allow image sizes to vary need call this.

If the effect did not trap this, it means the host should use the default RoD instead, which depends on the context. This is…

- generator context - defaults to the project window,
- filter and paint contexts - defaults to the RoD of the 'Source' input clip at the given time,
- transition context - defaults to the union of the RoDs of the 'SourceFrom' and 'SourceTo' input clips at the given time,
- general context - defaults to the union of the RoDs of all the non optional input clips and the 'Source' input clip (if it exists and it is connected) at the given time, if none exist, then it is the project window
- retimer context - defaults to the union of the RoD of the 'Source' input clip at the frame directly preceding the value of the 'SourceTime' double parameter and the frame directly after it

Parameters

- **`handle`** – handle to the instance, cast to an *OfxImageEffectHandle*
- **`inArgs`** – has the following properties
  - *kOfxPropTime* the effect time for which a region of definition is being requested
  - *kOfxImageEffectPropRenderScale* the render scale that should be used in any calculations in this action
- **`outArgs`** – has the following property which the plug-in may set
  - *kOfxImageEffectPropRegionOfDefinition* the calculated region of definition, initially set by the host to the default RoD (see below), in Canonical Coordinates.

Returns

- *kOfxStatOK* the action was trapped and the RoD was set in the outArgs property set
- *kOfxStatReplyDefault*, the action was not trapped and the host should use the default values
- *kOfxStatErrMemory*, in which case the action may be called again after a memory purge
- *kOfxStatFailed*, something wrong, but no error code appropriate, plugin to post message
- *kOfxStatErrFatal*

`kOfxImageEffectActionGetRegionsOfInterest`

This action allows a host to ask an effect, given a region I want to render, what region do you need from each of your input clips. In that way, depending on the host architecture, a host can fetch the minimal amount of the image needed as input. Note that there is a region of interest to be set in `outArgs` for each input clip that exists on the effect. For more details see Image EffectArchitectures".

The default RoI is simply the value passed in on the *kOfxImageEffectPropRegionOfInterest* `inArgs` property set. All the RoIs in the `outArgs` property set must initialised to this value before the action is called.

Parameters

- **`handle`** – handle to the instance, cast to an *OfxImageEffectHandle*
- **`inArgs`** – has the following properties
  - *kOfxPropTime* the effect time for which a region of definition is being requested
  - *kOfxImageEffectPropRenderScale* the render scale that should be used in any calculations in this action

  - *kOfxImageEffectPropRegionOfInterest* the region to be rendered in the output image, in Canonical Coordinates.
- **outArgs** – has a set of 4 dimensional double properties, one for each of the input clips to the effect. The properties are each named `OfxImageClipPropRoI_` with the clip name post pended, for example `OfxImageClipPropRoI_Source`. These are initialised to the default RoI.

Returns

- *kOfxStatOK*, the action was trapped and at least one RoI was set in the outArgs property set
- *kOfxStatReplyDefault*, the action was not trapped and the host should use the default values
- *kOfxStatErrMemory*, in which case the action may be called again after a memory purge
- *kOfxStatFailed*, something wrong, but no error code appropriate, plugin to post message
- *kOfxStatErrFatal*

`kOfxImageEffectActionGetFramesNeeded`

This action lets the host ask the effect what frames are needed from each input clip to process a given frame. For example a temporal based degrainer may need several frames around the frame to render to do its work.

This action need only ever be called if the plugin has set the *kOfxImageEffectPropTemporalClipAccess* property on the plugin descriptor to be true. Otherwise the host assumes that the only frame needed from the inputs is the current one and this action is not called.

Note that each clip can have it's required frame range specified, and that you can specify discontinuous sets of ranges for each clip, for example

```
// The effect always needs the initial frame of the source as well as the previous␣
↪and current frame
double rangeSource[4];

// required ranges on the source
rangeSource[0] = 0; // we always need frame 0 of the source
rangeSource[1] = 0;
rangeSource[2] = currentFrame - 1; // we also need the previous and current frame␣
↪on the source
rangeSource[3] = currentFrame;

gPropHost->propSetDoubleN(outArgs, "OfxImageClipPropFrameRange_Source", 4,␣
↪rangeSource);
```

```
Which sets two discontinuous range of frames from the 'Source' clip
required as input.
```

The default frame range is simply the single frame, kOfxPropTime..kOfxPropTime, found on the `inArgs` property set. All the frame ranges in the `outArgs` property set must initialised to this value before the action is called.

Parameters

- **handle** – handle to the instance, cast to an *OfxImageEffectHandle*
- **inArgs** – has the following property
  - *kOfxPropTime* the effect time for which we need to calculate the frames needed on input

  - outArgs has a set of properties, one for each input clip, named `OfxImageClipPropFrameRange_` with the name of the clip post-pended. For example `OfxImageClipPropFrameRange_Source`. All these properties are multi-dimensional doubles, with the dimension is a multiple of two. Each pair of values indicates a continuous range of frames that is needed on the given input. They are all initalised to the default value.

Returns

- *kOfxStatOK*, the action was trapped and at least one frame range in the outArgs property set
- *kOfxStatReplyDefault*, the action was not trapped and the host should use the default values
- *kOfxStatErrMemory*, in which case the action may be called again after a memory purge
- *kOfxStatFailed*, something wrong, but no error code appropriate, plugin to post message
- *kOfxStatErrFatal*

`kOfxImageEffectActionIsIdentity`

Sometimes an effect can pass through an input uprocessed, for example a blur effect with a blur size of 0. This action can be called by a host before it attempts to render an effect to determine if it can simply copy input directly to output without having to call the render action on the effect.

If the effect does not need to process any pixels, it should set the value of the *kOfxPropName* to the clip that the host should us as the output instead, and the *kOfxPropTime* property on `outArgs` to be the time at which the frame should be fetched from a clip.

The default action is to call the render action on the effect.

Parameters

- **`handle`** – handle to the instance, cast to an *OfxImageEffectHandle*
- **`inArgs`** – has the following properties
  - *kOfxPropTime* the time at which to test for identity
  - *kOfxImageEffectPropFieldToRender* the field to test for identity
  - *kOfxImageEffectPropRenderWindow* the window (in \ref PixelCoordinates) to test for identity under
  - *kOfxImageEffectPropRenderScale* the scale factor being applied to the images being renderred
- **`outArgs`** – has the following properties which the plugin can set
  - *kOfxPropName* this to the name of the clip that should be used if the effect is an identity transform, defaults to the empty string
  - *kOfxPropTime* the time to use from the indicated source clip as an identity image (allowing time slips to happen), defaults to the value in *kOfxPropTime* in inArgs

Returns

- *kOfxStatOK*, the action was trapped and the effect should not have its render action called, the values in outArgs indicate what frame from which clip to use instead
- *kOfxStatReplyDefault*, the action was not trapped and the host should call the render action
- *kOfxStatErrMemory*, in which case the action may be called again after a memory purge
- *kOfxStatFailed*, something wrong, but no error code appropriate, plugin to post message
- *kOfxStatErrFatal*

`kOfxImageEffectActionRender`

This action is where an effect gets to push pixels and turn its input clips and parameter set into an output image. This is possibly quite complicated and covered in the Rendering Image Effects chapter.

The render action *must* be trapped by the plug-in, it cannot return *kOfxStatReplyDefault*. The pixels needs be pushed I'm afraid.

Parameters

- **`handle`** – handle to the instance, cast to an *OfxImageEffectHandle*
- **`inArgs`** – has the following properties
  - *kOfxPropTime* the time at which to render
  - *kOfxImageEffectPropFieldToRender* the field to render
  - *kOfxImageEffectPropRenderWindow* the window (in \ref PixelCoordinates) to render
  - *kOfxImageEffectPropRenderScale* the scale factor being applied to the images being renderred
  - *kOfxImageEffectPropSequentialRenderStatus* whether the effect is currently being rendered in strict frame order on a single instance
  - *kOfxImageEffectPropInteractiveRenderStatus* if the render is in response to a user modifying the effect in an interactive session
  - *kOfxImageEffectPropRenderQualityDraft* if the render should be done in draft mode (e.g. for faster scrubbing)
- **`outArgs`** – is redundant and should be set to NULL

Pre

- *kOfxActionCreateInstance* has been called on the instance
- *kOfxImageEffectActionBeginSequenceRender* has been called on the instance

Post

- *kOfxImageEffectActionEndSequenceRender* action will be called on the instance

Returns

- *kOfxStatOK*, the effect rendered happily
- *kOfxStatErrMemory*, in which case the action may be called again after a memory purge
- *kOfxStatFailed*, something wrong, but no error code appropriate, plugin to post message
- *kOfxStatErrFatal*

`kOfxImageEffectActionBeginSequenceRender`

This action is passed to an image effect before it renders a range of frames. It is there to allow an effect to set things up for a long sequence of frames. Note that this is still called, even if only a single frame is being rendered in an interactive environment.

Parameters

- **`handle`** – handle to the instance, cast to an *OfxImageEffectHandle*
- **`inArgs`** – has the following properties
  - *kOfxImageEffectPropFrameRange* the range of frames (inclusive) that will be renderred

  - *kOfxImageEffectPropFrameStep* what is the step between frames, generally set to 1 (for full frame renders) or 0.5 (for fielded renders)
  - *kOfxPropIsInteractive* is this a single frame render due to user interaction in a GUI, or a proper full sequence render.
  - *kOfxImageEffectPropRenderScale* the scale factor to apply to images for this call
  - *kOfxImageEffectPropSequentialRenderStatus* whether the effect is currently being rendered in strict frame order on a single instance
  - *kOfxImageEffectPropInteractiveRenderStatus* if the render is in response to a user modifying the effect in an interactive session
- **outArgs** – is redundant and is set to NULL

Pre

- *kOfxActionCreateInstance* has been called on the instance

Post

- *kOfxImageEffectActionRender* action will be called at least once on the instance
- *kOfxImageEffectActionEndSequenceRender* action will be called on the instance

Returns

- *kOfxStatOK*, the action was trapped and handled cleanly by the effect,
- *kOfxStatReplyDefault*, the action was not trapped, but all is well anyway,
- *kOfxStatErrMemory*, in which case the action may be called again after a memory purge,
- *kOfxStatFailed*, something wrong, but no error code appropriate, plugin to post message,
- *kOfxStatErrFatal*

kOfxImageEffectActionEndSequenceRender

This action is passed to an image effect after is has rendered a range of frames. It is there to allow an effect to free resources after a long sequence of frame renders. Note that this is still called, even if only a single frame is being rendered in an interactive environment.

Parameters

- **handle** – handle to the instance, cast to an *OfxImageEffectHandle*
- **inArgs** – has the following properties
  - *kOfxImageEffectPropFrameRange* the range of frames (inclusive) that will be rendered
  - *kOfxImageEffectPropFrameStep* what is the step between frames, generally set to 1 (for full frame renders) or 0.5 (for fielded renders),
  - *kOfxPropIsInteractive*
  - is this a single frame render due to user interaction in a GUI, or a proper full sequence render.
  - *kOfxImageEffectPropRenderScale*
  - the scale factor to apply to images for this call
  - *kOfxImageEffectPropSequentialRenderStatus*
  - whether the effect is currently being rendered in strict frame order on a single instance
  - *kOfxImageEffectPropInteractiveRenderStatus*

  - if the render is in response to a user modifying the effect in an interactive session
- **outArgs** – is redundant and is set to NULL

Pre

- *kOfxActionCreateInstance* has been called on the instance
- *kOfxImageEffectActionEndSequenceRender* action was called on the instance
- *kOfxImageEffectActionRender* action was called at least once on the instance

Returns

- *kOfxStatOK*, the action was trapped and handled cleanly by the effect,
- *kOfxStatReplyDefault*, the action was not trapped, but all is well anyway,
- *kOfxStatErrMemory*, in which case the action may be called again after a memory purge,
- *kOfxStatFailed*, something wrong, but no error code appropriate, plugin to post message,
- *kOfxStatErrFatal*

kOfxImageEffectActionGetClipPreferences

This action allows a plugin to dynamically specify its preferences for input and output clips. Please see Image Effect Clip Preferences for more details on the behaviour. Clip preferences are constant for the duration of an effect, so this action need only be called once per clip, not once per frame.

This should be called once after creation of an instance, each time an input clip is changed, and whenever a parameter named in the *kOfxImageEffectPropClipPreferencesSlaveParam* has its value changed.

Parameters

- **handle** – handle to the instance, cast to an *OfxImageEffectHandle*
- **inArgs** – is redundant and is set to NULL
- **outArgs** – has the following properties which the plugin can set
  - a set of char * X 1 properties, one for each of the input clips currently attached and the output clip, labelled with `OfxImageClipPropComponents_` post pended with the clip's name. This must be set to one of the component types which the host supports and the effect stated it can accept on that input
  - a set of char * X 1 properties, one for each of the input clips currently attached and the output clip, labelled with `OfxImageClipPropDepth_` post pended with the clip's name. This must be set to one of the pixel depths both the host and plugin supports
  - a set of double X 1 properties, one for each of the input clips currently attached and the output clip, labelled with `OfxImageClipPropPAR_` post pended with the clip's name. This is the pixel aspect ratio of the input and output clips. This must be set to a positive non zero double value,
  - *kOfxImageEffectPropFrameRate* the frame rate of the output clip, this must be set to a positive non zero double value
  - *kOfxImageClipPropFieldOrder* the fielding of the output clip
  - *kOfxImageEffectPropPreMultiplication* the premultiplication of the output clip
  - *kOfxImageClipPropContinuousSamples* whether the output clip can produce different images at non-frame intervals, defaults to false,
  - *kOfxImageEffectFrameVarying* whether the output clip can produces different images at different times, even if all parameters and inputs are constant, defaults to false.

Returns

- *kOfxStatOK*, the action was trapped and at least one of the properties in the outArgs was changed from its default value
- *kOfxStatReplyDefault*, the action was not trapped and the host should use the default values
- *kOfxStatErrMemory*, in which case the action may be called again after a memory purge
- *kOfxStatFailed*, something wrong, but no error code appropriate, plugin to post message
- *kOfxStatErrFatal*

`kOfxImageEffectActionGetTimeDomain`

This action allows a host to ask an effect what range of frames it can produce images over. Only effects instantiated in the GeneralContext" can have this called on them. In all other the host is in strict control over the temporal duration of the effect.

The default is:

- the union of all the frame ranges of the non optional input clips,
- infinite if there are no non optional input clips.

Parameters

- **`handle`** – handle to the instance, cast to an *OfxImageEffectHandle*
- **`inArgs`** – is redundant and is null
- **`outArgs`** – has the following property
  - *kOfxImageEffectPropFrameRange* the frame range an effect can produce images for

Pre

- *kOfxActionCreateInstance* has been called on the instance
- the effect instance has been created in the general effect context

Returns

- *kOfxStatOK*, the action was trapped and the *kOfxImageEffectPropFrameRange* was set in the outArgs property set
- *kOfxStatReplyDefault*, the action was not trapped and the host should use the default value
- *kOfxStatErrMemory*, in which case the action may be called again after a memory purge
- *kOfxStatFailed*, something wrong, but no error code appropriate, plugin to post message
- *kOfxStatErrFatal*

1.17 Actions Passed to an Interact

This chapter describes the actions that can be issued to an interact's main entry point. Interact actions are also generic in character, they could be issued to other plug-in types rather than just Image Effects, however they are not issued directly to an effect's main entry point, they are rather issued to separate entry points which exist on specific 'interact' objects that a plugin may create.

For nearly all the actions the `handle` passed to to main entry point for an interact will be either NULL, or a value that should be cast to an `OfxInteractHandle`.

`kOfxActionDescribeInteract`

This action is the first action passed to an interact. It is where an interact defines how it behaves and the resources it needs to function. If not trapped, the default action is for the host to carry on as normal Note that the handle passed in acts as a descriptor for, rather than an instance of the interact.

Parameters

- **`handle`** – handle to the interact descriptor, cast to an *OfxInteractHandle*
- **`inArgs`** – is redundant and is set to NULL
- **`outArgs`** – is redundant and is set to NULL

Pre

- The plugin has been loaded and the effect described.

Returns

- *kOfxStatOK* the action was trapped and all was well
- *kOfxStatErrMemory* in which case describe may be called again after a memory purge
- *kOfxStatFailed* something was wrong, the host should ignore the interact
- *kOfxStatErrFatal*

`kOfxActionCreateInstanceInteract`

This action is the first action passed to an interact instance after its creation. It is there to allow a plugin to create any per-instance data structures it may need.

Parameters

- **`handle`** – handle to the interact instance, cast to an *OfxInteractHandle*
- **`inArgs`** – is redundant and is set to NULL
- **`outArgs`** – is redundant and is set to NULL

Pre

- *kOfxActionDescribe* has been called on this interact

Post

- the instance pointer will be valid until the *kOfxActionDestroyInstance* action is passed to the plug-in with the same instance handle

Returns

- *kOfxStatOK* the action was trapped and all was well
- *kOfxStatReplyDefault* the action was ignored, but all was well anyway
- *kOfxStatErrFatal*

- *kOfxStatErrMemory* in which case this may be called again after a memory purge
- *kOfxStatFailed* in which case the host should ignore this interact

`kOfxActionDestroyInstanceInteract`

This action is the last passed to an interact's instance before its destruction. It is there to allow a plugin to destroy any per-instance data structures it may have created.

Parameters

- **`handle`** – handle to the interact instance, cast to an *OfxInteractHandle*
- **`inArgs`** – is redundant and is set to NULL
- **`outArgs`** – is redundant and is set to NULL

Pre

- *kOfxActionCreateInstance* has been called on the handle,
- the instance has not had any of its members destroyed yet

Post

- the instance pointer is no longer valid and any operation on it will be undefined

Returns To some extent, what is returned is moot, a bit like throwing an exception in a C++ destructor, so the host should continue destruction of the instance regardless

- *kOfxStatOK* the action was trapped and all was well
- *kOfxStatReplyDefault* the action was ignored as the effect had nothing to do
- *kOfxStatErrFatal*
- *kOfxStatFailed* something went wrong, but no error code appropriate.

`kOfxInteractActionDraw`

This action is issued to an interact whenever the host needs the plugin to redraw the given interact. The interact should issue any openGL calls it needs at this point.

Note that the interact may (in the case of custom parameter GUIS) or may not (in the case of image effect overlays) be required to swap buffers, that is up to the kind of interact.

Parameters

- **`handle`** – handle to an interact instance, cast to an *OfxInteractHandle*
- **`inArgs`** – has the following properties on an image effect plugin
  - *kOfxPropEffectInstance* a handle to the effect for which the interact has been,
  - *kOfxInteractPropPixelScale* the scale factor to convert cannonical pixels to screen pixels
  - *kOfxInteractPropBackgroundColour* the background colour of the application behind the current view
  - *kOfxPropTime* the effect time at which changed occured
  - *kOfxImageEffectPropRenderScale* the render scale applied to any image fetched
- **`outArgs`** – is redundant and is set to NULL

Pre

- *kOfxActionCreateInstance* has been called on the instance handle

- the openGL context for this interact has been set
- the projection matrix will correspond to the interact's cannonical view

Returns

- *kOfxStatOK* the action was trapped and all was well
- *kOfxStatReplyDefault* the action was ignored
- *kOfxStatErrFatal*
- *kOfxStatFailed* something went wrong, the host should ignore this interact in future

`kOfxInteractActionPenMotion`

This action is issued whenever the pen moves an the interact's has focus. It should be issued whether the pen is currently up or down. No openGL calls should be issued by the plug-in during this action.

Parameters

- **`handle`** – handle to an interact instance, cast to an *OfxInteractHandle*
- **`inArgs`** – has the following properties on an image effect plugin
  - *kOfxPropEffectInstance* a handle to the effect for which the interact has been,
  - *kOfxInteractPropPixelScale* the scale factor to convert cannonical pixels to screen pixels
  - *kOfxInteractPropBackgroundColour* the background colour of the application behind the current view
  - *kOfxPropTime* the effect time at which changed occured
  - *kOfxImageEffectPropRenderScale* the render scale applied to any image fetched
  - *kOfxInteractPropPenPosition* postion of the pen in,
  - *kOfxInteractPropPenViewportPosition* position of the pen in,
  - *kOfxInteractPropPenPressure* the pressure of the pen,
- **`outArgs`** – is redundant and is set to NULL

Pre

- *kOfxActionCreateInstance* has been called on the instance handle
- the current instance handle has had *kOfxInteractActionGainFocus* called on it

Post

- if the instance returns *kOfxStatOK* the host should not pass the pen motion to any other interactive object it may own that shares the same view.

Returns

- *kOfxStatOK* the action was trapped and the host should not pass the event to other objects it may own
- *kOfxStatReplyDefault* the action was not trapped and the host can deal with it if it wants

`kOfxInteractActionPenDown`

This action is issued when a pen transitions for the 'up' to the 'down' state. No openGL calls should be issued by the plug-in during this action.

Parameters

- **handle** – handle to an interact instance, cast to an *OfxInteractHandle*
- **inArgs** – has the following properties on an image effect plugin,
  - *kOfxPropEffectInstance* a handle to the effect for which the interact has been,
  - *kOfxInteractPropPixelScale* the scale factor to convert cannonical pixels to screen pixels
  - *kOfxInteractPropBackgroundColour* the background colour of the application behind the current view
  - *kOfxPropTime* the effect time at which changed occured
  - *kOfxImageEffectPropRenderScale* the render scale applied to any image fetched
  - *kOfxInteractPropPenPosition* position of the pen in
  - *kOfxInteractPropPenViewportPosition* position of the pen in
  - *kOfxInteractPropPenPressure* the pressure of the pen
- **outArgs** – is redundant and is set to NULL

Pre

- *kOfxActionCreateInstance* has been called on the instance handle,
- the current instance handle has had *kOfxInteractActionGainFocus* called on it

Post

- if the instance returns *kOfxStatOK*, the host should not pass the pen motion to any other interactive object it may own that shares the same view.

Returns

- *kOfxStatOK*, the action was trapped and the host should not pass the event to other objects it may own
- *kOfxStatReplyDefault* , the action was not trapped and the host can deal with it if it wants

kOfxInteractActionPenUp

This action is issued when a pen transitions for the 'down' to the 'up' state. No openGL calls should be issued by the plug-in during this action.

Parameters

- **handle** – handle to an interact instance, cast to an *OfxInteractHandle*
- **inArgs** – has the following properties on an image effect plugin,
  - *kOfxPropEffectInstance* a handle to the effect for which the interact has been,
  - *kOfxInteractPropPixelScale* the scale factor to convert cannonical pixels to screen pixels
  - *kOfxInteractPropBackgroundColour* the background colour of the application behind the current view
  - *kOfxPropTime* the effect time at which changed occured
  - *kOfxImageEffectPropRenderScale* the render scale applied to any image fetched
  - *kOfxInteractPropPenPosition* position of the pen in
  - *kOfxInteractPropPenViewportPosition* position of the pen in
  - *kOfxInteractPropPenPressure* the pressure of the pen
- **outArgs** – is redundant and is set to NULL

Pre

- *kOfxActionCreateInstance* has been called on the instance handle,
- the current instance handle has had *kOfxInteractActionGainFocus* called on it

Post

- if the instance returns *kOfxStatOK*, the host should not pass the pen motion to any other interactive object it may own that shares the same view.

Returns

- *kOfxStatOK*, the action was trapped and the host should not pass the event to other objects it may own
- *kOfxStatReplyDefault* , the action was not trapped and the host can deal with it if it wants

`kOfxInteractActionKeyDown`

This action is issued when a key on the keyboard is depressed. No openGL calls should be issued by the plug-in during this action.

Parameters

- **`handle`** – handle to an interact instance, cast to an *OfxInteractHandle*
- **`inArgs`** – has the following properties on an image effect plugin
  - *kOfxPropEffectInstance* a handle to the effect for which the interact has been,
  - *kOfxPropKeySym* single integer value representing the key that was manipulated, this may not have a UTF8 representation (eg: a return key)
  - *kOfxPropKeyString* UTF8 string representing a character key that was pressed, some keys have no UTF8 encoding, in which case this is “”
  - *kOfxPropTime* the effect time at which changed occured
  - *kOfxImageEffectPropRenderScale* the render scale applied to any image fetched
- **`outArgs`** – is redundant and is set to NULL

Pre

- *kOfxActionCreateInstance* has been called on the instance handle,
- the current instance handle has had *kOfxInteractActionGainFocus* called on it

Post

- if the instance returns *kOfxStatOK*, the host should not pass the pen motion to any other interactive object it may own that shares the same focus.

Returns

- *kOfxStatOK* , the action was trapped and the host should not pass the event to other objects it may own
- *kOfxStatReplyDefault* , the action was not trapped and the host can deal with it if it wants

`kOfxInteractActionKeyUp`

This action is issued when a key on the keyboard is released. No openGL calls should be issued by the plug-in during this action.

Parameters

- **handle** – handle to an interact instance, cast to an *OfxInteractHandle*
- **inArgs** – has the following properties on an image effect plugin
  - *kOfxPropEffectInstance* a handle to the effect for which the interact has been,
  - *kOfxPropKeySym* single integer value representing the key that was manipulated, this may not have a UTF8 representation (eg: a return key)
  - *kOfxPropKeyString* UTF8 string representing a character key that was pressed, some keys have no UTF8 encoding, in which case this is “”
  - *kOfxPropTime* the effect time at which changed occured
  - *kOfxImageEffectPropRenderScale* the render scale applied to any image fetched
- **outArgs** – is redundant and is set to NULL

Pre

- *kOfxActionCreateInstance* has been called on the instance handle,
- the current instance handle has had *kOfxInteractActionGainFocus* called on it

Post

- if the instance returns *kOfxStatOK*, the host should not pass the pen motion to any other interactive object it may own that shares the same focus.

Returns

- *kOfxStatOK* , the action was trapped and the host should not pass the event to other objects it may own
- *kOfxStatReplyDefault* , the action was not trapped and the host can deal with it if it wants

kOfxInteractActionKeyRepeat

This action is issued when a key on the keyboard is repeated. No openGL calls should be issued by the plug-in during this action.

Parameters

- **handle** – handle to an interact instance, cast to an *OfxInteractHandle*
- **inArgs** – has the following properties on an image effect plugin
  - *kOfxPropEffectInstance* a handle to the effect for which the interact has been,
  - *kOfxPropKeySym* single integer value representing the key that was manipulated, this may not have a UTF8 representation (eg: a return key)
  - *kOfxPropKeyString* UTF8 string representing a character key that was pressed, some keys have no UTF8 encoding, in which case this is “”
  - *kOfxPropTime* the effect time at which changed occured
  - *kOfxImageEffectPropRenderScale* the render scale applied to any image fetched
- **outArgs** – is redundant and is set to NULL

Pre

- *kOfxActionCreateInstance* has been called on the instance handle,
- the current instance handle has had *kOfxInteractActionGainFocus* called on it

Post

- if the instance returns *kOfxStatOK*, the host should not pass the pen motion to any other interactive object it may own that shares the same focus.

Returns

- *kOfxStatOK* , the action was trapped and the host should not pass the event to other objects it may own
- *kOfxStatReplyDefault* , the action was not trapped and the host can deal with it if it wants

`kOfxInteractActionGainFocus`

This action is issued when an interact gains input focus. No openGL calls should be issued by the plug-in during this action.

Parameters

- **`handle`** – handle to an interact instance, cast to an *OfxInteractHandle*
- **`inArgs`** – has the following properties on an image effect plugin
  - *kOfxPropEffectInstance* a handle to the effect for which the interact is being used on,
  - *kOfxInteractPropPixelScale* the scale factor to convert cannonical pixels to screen pixels,
  - *kOfxInteractPropBackgroundColour* the background colour of the application behind the current view
  - *kOfxPropTime* the effect time at which changed occured
  - *kOfxImageEffectPropRenderScale* the render scale applied to any image fetched
- **`outArgs`** – is redundant and is set to NULL

Pre

- *kOfxActionCreateInstance* has been called on the instance handle,

Returns

- *kOfxStatOK* the action was trapped
- *kOfxStatReplyDefault* the action was not trapped

`kOfxInteractActionLoseFocus`

This action is issued when an interact loses input focus. No openGL calls should be issued by the plug-in during this action.

Parameters

- **`handle`** – handle to an interact instance, cast to an *OfxInteractHandle*
- **`inArgs`** – has the following properties on an image effect plugin
  - *kOfxPropEffectInstance* a handle to the effect for which the interact is being used on,
  - *kOfxInteractPropPixelScale* the scale factor to convert cannonical pixels to screen pixels,
  - *kOfxInteractPropBackgroundColour* the background colour of the application behind the current view
  - *kOfxPropTime* the effect time at which changed occured
  - *kOfxImageEffectPropRenderScale* the render scale applied to any image fetched
- **`outArgs`** – is redundant and is set to NULL

Pre

- *kOfxActionCreateInstance* has been called on the instance handle,

Returns

- *kOfxStatOK* the action was trapped
- *kOfxStatReplyDefault* the action was not trapped

1.18 OpenFX suites reference

This table list all suites available in the OpenFX standard

1.18.1 OfxPropertySuiteV1

The files `ofxCore.h` and `ofxProperty.h` contain the basic definitions for the property suite.

The property suite is the most basic and important suite in OFX, it is used to get and set the values of various objects defined by other suites.

A property is a named value of a specific data type, such values can be multi-dimensional, but is typically of one dimension. The name is a 'C' string literal, typically #defined in one of the various OFX header files. For example, the property labeled by the string literal `"OfxPropName"` is a 'C' string which holds the name of some object.

Properties are not accessed in isolation, but are grouped and accessed through a property set handle. The number and types of properties on a specific property set handle are currently strictly defined by the API that the properties are being used for. There is no scope to add new properties.

There is a naming convention for property labels and the macros #defined to them. The scheme is,

- generic properties names start with "OfxProp" + name of the property, e.g. "OfxPropTime".
- properties pertaining to a specific object with "Ofx" + object name + "Prop" + name of the property, e.g. "OfxParamPropAnimates".
- the C preprocessor #define used to define the string literal is the same as the string literal, but with "k" prepended to the name. For example, #define kOfxPropLabel "OfxPropLabel"

OfxPropertySetHandle OfxPropertySetHandle Blind data type used to hold sets of properties

```
#include "ofxCore.h"
typedef struct OfxPropertySetStruct *OfxPropertySetHandle;
```

Description

Properties are not accessed on their own, nor do they exist on their own. They are grouped and manipulated via an OfxPropertySetHandle.

Any object that has properties can be made to return it's property set handle via some call on the relevant suite. Individual properties are then manipulated with the property suite through that handle.

struct **`OfxPropertySuiteV1`**
    The OFX suite used to access properties on OFX objects.

Public Members

OfxStatus (***propSetPointer**)(*OfxPropertySetHandle* properties, const char *property, int index, void *value)

Set a single value in a pointer property.

- properties is the handle of the thing holding the property
- property is the string labelling the property
- index is for multidimenstional properties and is dimension of the one we are setting
- value is the value of the property we are setting

Return

- *kOfxStatOK*
- *kOfxStatErrBadHandle*
- *kOfxStatErrUnknown*
- *kOfxStatErrBadIndex*
- *kOfxStatErrValue*

OfxStatus (***propSetString**)(*OfxPropertySetHandle* properties, const char *property, int index, const char *value)

Set a single value in a string property.

- properties is the handle of the thing holding the property
- property is the string labelling the property
- index is for multidimenstional properties and is dimension of the one we are setting
- value is the value of the property we are setting

Return

- *kOfxStatOK*
- *kOfxStatErrBadHandle*
- *kOfxStatErrUnknown*
- *kOfxStatErrBadIndex*
- *kOfxStatErrValue*

OfxStatus (***propSetDouble**)(*OfxPropertySetHandle* properties, const char *property, int index, double value)

Set a single value in a double property.

- properties is the handle of the thing holding the property
- property is the string labelling the property
- index is for multidimenstional properties and is dimension of the one we are setting

- value is the value of the property we are setting

Return

- *kOfxStatOK*
- *kOfxStatErrBadHandle*
- *kOfxStatErrUnknown*
- *kOfxStatErrBadIndex*
- *kOfxStatErrValue*

OfxStatus (***propSetInt**)(*OfxPropertySetHandle* properties, const char *property, int index, int value)
Set a single value in an int property.

- properties is the handle of the thing holding the property
- property is the string labelling the property
- index is for multidimenstional properties and is dimension of the one we are setting
- value is the value of the property we are setting

Return

- *kOfxStatOK*
- *kOfxStatErrBadHandle*
- *kOfxStatErrUnknown*
- *kOfxStatErrBadIndex*
- *kOfxStatErrValue*

OfxStatus (***propSetPointerN**)(*OfxPropertySetHandle* properties, const char *property, int count, void *const *value)
Set multiple values of the pointer property.

- properties is the handle of the thing holding the property
- property is the string labelling the property
- count is the number of values we are setting in that property (ie: indicies 0..count-1)
- value is a pointer to an array of property values

Return

- *kOfxStatOK*
- *kOfxStatErrBadHandle*
- *kOfxStatErrUnknown*
- *kOfxStatErrBadIndex*
- *kOfxStatErrValue*

OfxStatus (***propSetStringN**)(*OfxPropertySetHandle* properties, const char *property, int count, const char *const *value)

Set multiple values of a string property.

- properties is the handle of the thing holding the property
- property is the string labelling the property
- count is the number of values we are setting in that property (ie: indicies 0..count-1)
- value is a pointer to an array of property values

Return

- *kOfxStatOK*
- *kOfxStatErrBadHandle*
- *kOfxStatErrUnknown*
- *kOfxStatErrBadIndex*
- *kOfxStatErrValue*

OfxStatus (***propSetDoubleN**)(*OfxPropertySetHandle* properties, const char *property, int count, const double *value)

Set multiple values of a double property.

- properties is the handle of the thing holding the property
- property is the string labelling the property
- count is the number of values we are setting in that property (ie: indicies 0..count-1)
- value is a pointer to an array of property values

Return

- *kOfxStatOK*
- *kOfxStatErrBadHandle*
- *kOfxStatErrUnknown*
- *kOfxStatErrBadIndex*
- *kOfxStatErrValue*

OfxStatus (***propSetIntN**)(*OfxPropertySetHandle* properties, const char *property, int count, const int *value)

Set multiple values of an int property.

- properties is the handle of the thing holding the property
- property is the string labelling the property
- count is the number of values we are setting in that property (ie: indicies 0..count-1)
- value is a pointer to an array of property values

Return

- *kOfxStatOK*
- *kOfxStatErrBadHandle*
- *kOfxStatErrUnknown*
- *kOfxStatErrBadIndex*
- *kOfxStatErrValue*

OfxStatus (***propGetPointer**)(*OfxPropertySetHandle* properties, const char *property, int index, void **value)

Get a single value from a pointer property.

- properties is the handle of the thing holding the property
- property is the string labelling the property
- index refers to the index of a multi-dimensional property
- value is a pointer the return location

Return

- *kOfxStatOK*
- *kOfxStatErrBadHandle*
- *kOfxStatErrUnknown*
- *kOfxStatErrBadIndex*

OfxStatus (***propGetString**)(*OfxPropertySetHandle* properties, const char *property, int index, char **value)

Get a single value of a string property.

- properties is the handle of the thing holding the property
- property is the string labelling the property
- index refers to the index of a multi-dimensional property
- value is a pointer the return location

Return

- *kOfxStatOK*
- *kOfxStatErrBadHandle*
- *kOfxStatErrUnknown*
- *kOfxStatErrBadIndex*

OfxStatus (***propGetDouble**)(*OfxPropertySetHandle* properties, const char *property, int index, double *value)

Get a single value of a double property.

- properties is the handle of the thing holding the property
- property is the string labelling the property
- index refers to the index of a multi-dimensional property
- value is a pointer the return location

See the note ArchitectureStrings for how to deal with strings.

Return

- *kOfxStatOK*
- *kOfxStatErrBadHandle*
- *kOfxStatErrUnknown*
- *kOfxStatErrBadIndex*

OfxStatus (***propGetInt**)(*OfxPropertySetHandle* properties, const char *property, int index, int *value)

Get a single value of an int property.

- properties is the handle of the thing holding the property
- property is the string labelling the property
- index refers to the index of a multi-dimensional property
- value is a pointer the return location

Return

- *kOfxStatOK*
- *kOfxStatErrBadHandle*
- *kOfxStatErrUnknown*
- *kOfxStatErrBadIndex*

OfxStatus (***propGetPointerN**)(*OfxPropertySetHandle* properties, const char *property, int count, void **value)

Get multiple values of a pointer property.

- properties is the handle of the thing holding the property
- property is the string labelling the property
- count is the number of values we are getting of that property (ie: indicies 0..count-1)
- value is a pointer to an array of where we will return the property values

Return

- *kOfxStatOK*
- *kOfxStatErrBadHandle*
- *kOfxStatErrUnknown*
- *kOfxStatErrBadIndex*

OfxStatus (***propGetStringN**)(*OfxPropertySetHandle* properties, const char *property, int count, char **value)

Get multiple values of a string property.

- properties is the handle of the thing holding the property
- property is the string labelling the property
- count is the number of values we are getting of that property (ie: indicies 0..count-1)
- value is a pointer to an array of where we will return the property values

See the note ArchitectureStrings for how to deal with strings.

Return

- *kOfxStatOK*
- *kOfxStatErrBadHandle*
- *kOfxStatErrUnknown*
- *kOfxStatErrBadIndex*

OfxStatus (***propGetDoubleN**)(*OfxPropertySetHandle* properties, const char *property, int count, double *value)

Get multiple values of a double property.

- properties is the handle of the thing holding the property
- property is the string labelling the property
- count is the number of values we are getting of that property (ie: indicies 0..count-1)
- value is a pointer to an array of where we will return the property values

Return

- *kOfxStatOK*
- *kOfxStatErrBadHandle*
- *kOfxStatErrUnknown*
- *kOfxStatErrBadIndex*

OfxStatus (***propGetIntN**)(*OfxPropertySetHandle* properties, const char *property, int count, int *value)

Get multiple values of an int property.

- properties is the handle of the thing holding the property
- property is the string labelling the property
- count is the number of values we are getting of that property (ie: indicies 0..count-1)
- value is a pointer to an array of where we will return the property values

Return

- *kOfxStatOK*

- *kOfxStatErrBadHandle*
- *kOfxStatErrUnknown*
- *kOfxStatErrBadIndex*

OfxStatus (***propReset**)(*OfxPropertySetHandle* properties, const char *property)

Resets all dimensions of a property to its default value.

- properties is the handle of the thing holding the property
- property is the string labelling the property we are resetting

Return

- *kOfxStatOK*
- *kOfxStatErrBadHandle*
- *kOfxStatErrUnknown*

OfxStatus (***propGetDimension**)(*OfxPropertySetHandle* properties, const char *property, int *count)

Gets the dimension of the property.

- properties is the handle of the thing holding the property
- property is the string labelling the property we are resetting
- count is a pointer to an integer where the value is returned

Return

- *kOfxStatOK*
- *kOfxStatErrBadHandle*
- *kOfxStatErrUnknown*

1.18.2 OfxImageEffectSuiteV1

struct **OfxImageEffectSuiteV1**

The OFX suite for image effects.

This suite provides the functions needed by a plugin to defined and use an image effect plugin.

Public Members

OfxStatus (***getPropertySet**)(*OfxImageEffectHandle* imageEffect, *OfxPropertySetHandle* *propHandle)

Retrieves the property set for the given image effect.

- imageEffect image effect to get the property set for
- propHandle pointer to a the property set pointer, value is returned here

The property handle is for the duration of the image effect handle.

Return

- *kOfxStatOK* - the property set was found and returned
- *kOfxStatErrBadHandle* - if the paramter handle was invalid
- *kOfxStatErrUnknown* - if the type is unknown

OfxStatus (***getParamSet**)(*OfxImageEffectHandle* imageEffect, OfxParamSetHandle *paramSet)

Retrieves the parameter set for the given image effect.

- imageEffect image effect to get the property set for
- paramSet pointer to a the parameter set, value is returned here

The param set handle is valid for the lifetime of the image effect handle.

Return

- *kOfxStatOK* - the property set was found and returned
- *kOfxStatErrBadHandle* - if the paramter handle was invalid
- *kOfxStatErrUnknown* - if the type is unknown

OfxStatus (***clipDefine**)(*OfxImageEffectHandle* imageEffect, const char *name, *OfxPropertySetHandle* *propertySet)

Define a clip to the effect.

- pluginHandle - the handle passed into 'describeInContext' action
- name - unique name of the clip to define
- propertySet - a property handle for the clip descriptor will be returned here

This function defines a clip to a host, the returned property set is used to describe various aspects of the clip to the host. Note that this does not create a clip instance.

Pre

- we are inside the describe in context action.

Return

OfxStatus (***clipGetHandle**)(*OfxImageEffectHandle* imageEffect, const char *name, *OfxImageClipHandle* *clip, *OfxPropertySetHandle* *propertySet)

Get the propery handle of the named input clip in the given instance.

- imageEffect - an instance handle to the plugin
- name - name of the clip, previously used in a clip define call
- clip - where to return the clip
- propertySet if not null, the descriptor handle for a parameter's property set will be placed here.

The propertySet will have the same value as would be returned by *OfxImageEffectSuiteV1::clipGetPropertySet*

```
This return a clip handle for the given instance, note that this will \em not␣
↪be the same as the
clip handle returned by clipDefine and will be distanct to clip handles in any␣
↪other instance
of the plugin.

Not a valid call in any of the describe actions.
```

Pre

- create instance action called,
- *name* passed to clipDefine for this context,
- not inside describe or describe in context actions.

Post

- handle will be valid for the life time of the instance.

OfxStatus (***clipGetPropertySet**)(*OfxImageClipHandle* clip, *OfxPropertySetHandle* *propHandle)
Retrieves the property set for a given clip.

- clip clip effect to get the property set for
- propHandle pointer to a the property set handle, value is returedn her

The property handle is valid for the lifetime of the clip, which is generally the lifetime of the instance.

Return

- *kOfxStatOK* - the property set was found and returned
- *kOfxStatErrBadHandle* - if the paramter handle was invalid
- *kOfxStatErrUnknown* - if the type is unknown

OfxStatus (***clipGetImage**)(*OfxImageClipHandle* clip, OfxTime time, const *OfxRectD* *region, *OfxPropertySetHandle* *imageHandle)
Get a handle for an image in a clip at the indicated time and indicated region.

- clip - the clip to extract the image from
- time - time to fetch the image at
- region - region to fetch the image from (optional, set to NULL to get a 'default' region) this is in the CanonicalCoordinates.

- imageHandle - property set containing the image's data

An image is fetched from a clip at the indicated time for the given region and returned in the imageHandle.

If the *region* parameter is not set to NULL, then it will be clipped to the clip's Region of Definition for the given time. The returned image will be *at least* as big as this region. If the region parameter is not set, then the region fetched will be at least the Region of Interest the effect has previously specified, clipped the clip's Region of Definition.

If clipGetImage is called twice with the same parameters, then two separate image handles will be returned, each of which must be release. The underlying implementation could share image data pointers and use reference counting to maintain them.

Pre

- clip was returned by clipGetHandle

Post

- image handle is only valid for the duration of the action clipGetImage is called in
- image handle to be disposed of by clipReleaseImage before the action returns

Return

- *kOfxStatOK* - the image was successfully fetched and returned in the handle,
- *kOfxStatFailed* - the image could not be fetched because it does not exist in the clip at the indicated time and/or region, the plugin should continue operation, but assume the image was black and transparent.
- *kOfxStatErrBadHandle* - the clip handle was invalid,
- *kOfxStatErrMemory* - the host had not enough memory to complete the operation, plugin should abort whatever it was doing.

OfxStatus (*`clipReleaseImage`)(*OfxPropertySetHandle* imageHandle)
Releases the image handle previously returned by clipGetImage.

Pre

- imageHandle was returned by clipGetImage

Post

- all operations on imageHandle will be invalid

Return

- *kOfxStatOK* - the image was successfully fetched and returned in the handle,
- *kOfxStatErrBadHandle* - the image handle was invalid,

OfxStatus (*`clipGetRegionOfDefinition`)(*OfxImageClipHandle* clip, OfxTime time, *OfxRectD* *bounds)
Returns the spatial region of definition of the clip at the given time.

- clipHandle - the clip to extract the image from
- time - time to fetch the image at
- region - region to fetch the image from (optional, set to NULL to get a 'default' region) this is in the CanonicalCoordinates.
- imageHandle - handle where the image is returned

An image is fetched from a clip at the indicated time for the given region and returned in the imageHandle.

If the *region* parameter is not set to NULL, then it will be clipped to the clip's Region of Definition for the given time. The returned image will be *at least* as big as this region. If the region parameter is not set, then the region fetched will be at least the Region of Interest the effect has previously specified, clipped the clip's Region of Definition.

Pre

- clipHandle was returned by clipGetHandle

Post

- bounds will be filled the RoD of the clip at the indicated time

Return

- *kOfxStatOK* - the image was successfully fetched and returned in the handle,
- *kOfxStatFailed* - the image could not be fetched because it does not exist in the clip at the indicated time, the plugin should continue operation, but assume the image was black and transparent.
- *kOfxStatErrBadHandle* - the clip handle was invalid,
- *kOfxStatErrMemory* - the host had not enough memory to complete the operation, plugin should abort whatever it was doing.

int (***abort**)(*OfxImageEffectHandle* imageEffect)

Returns whether to abort processing or not.

- imageEffect - instance of the image effect

A host may want to signal to a plugin that it should stop whatever rendering it is doing and start again. Generally this is done in interactive threads in response to users tweaking some parameter.

This function indicates whether a plugin should stop whatever processing it is doing.

Return

- 0 if the effect should continue whatever processing it is doing
- 1 if the effect should abort whatever processing it is doing

OfxStatus (***imageMemoryAlloc**)(*OfxImageEffectHandle* instanceHandle, size_t nBytes, OfxImageMemoryHandle *memoryHandle)

Allocate memory from the host's image memory pool.

- instanceHandle - effect instance to associate with this memory allocation, may be NULL.
- nBytes - the number of bytes to allocate
- memoryHandle - pointer to the memory handle where a return value is placed

Memory handles allocated by this should be freed by *OfxImageEffectSuiteV1::imageMemoryFree*. To access the memory behind the handle you need to call *OfxImageEffectSuiteV1::imageMemoryLock*.

See ImageEffectsMemoryAllocation.

Return

- kOfxStatOK if all went well, a valid memory handle is placed in *memoryHandle*

- kOfxStatErrBadHandle if instanceHandle is not valid, memoryHandle is set to NULL
- kOfxStatErrMemory if there was not enough memory to satisfy the call, memoryHandle is set to NULL

OfxStatus (***imageMemoryFree**)(OfxImageMemoryHandle memoryHandle)

Frees a memory handle and associated memory.

- memoryHandle - memory handle returned by imageMemoryAlloc

This function frees a memory handle and associated memory that was previously allocated via *OfxImageEffectSuiteV1::imageMemoryAlloc*

If there are outstanding locks, these are ignored and the handle and memory are freed anyway.

See ImageEffectsMemoryAllocation.

Return

- kOfxStatOK if the memory was cleanly deleted
- kOfxStatErrBadHandle if the value of *memoryHandle* was not a valid pointer returned by *OfxImageEffectSuiteV1::imageMemoryAlloc*

OfxStatus (***imageMemoryLock**)(OfxImageMemoryHandle memoryHandle, void **returnedPtr)

Lock the memory associated with a memory handle and make it available for use.

- memoryHandle - memory handle returned by imageMemoryAlloc
- returnedPtr - where to the pointer to the locked memory

This function locks them memory associated with a memory handle and returns a pointer to it. The memory will be 16 byte aligned, to allow use of vector operations.

Note that memory locks and unlocks nest.

After the first lock call, the contents of the memory pointer to by *returnedPtr* is undefined. All subsequent calls to lock will return memory with the same contents as the previous call.

Also, if unlocked, then relocked, the memory associated with a memory handle may be at a different address.

See also *OfxImageEffectSuiteV1::imageMemoryUnlock* and ImageEffectsMemoryAllocation.

Return

- kOfxStatOK if the memory was locked, a pointer is placed in *returnedPtr*
- kOfxStatErrBadHandle if the value of *memoryHandle* was not a valid pointer returned by *OfxImageEffectSuiteV1::imageMemoryAlloc*, null is placed in **returnedPtr*
- kOfxStatErrMemory if there was not enough memory to satisfy the call, **returnedPtr* is set to NULL

OfxStatus (***imageMemoryUnlock**)(OfxImageMemoryHandle memoryHandle)

Unlock allocated image data.

- allocatedData - pointer to memory previously returned by OfxImageEffectSuiteV1::imageAlloc

This function unlocks a previously locked memory handle. Once completely unlocked, memory associated with a memoryHandle is no longer available for use. Attempting to use it results in undefined behaviour.

Note that locks and unlocks nest, and to fully unlock memory you need to match the count of locks placed upon it.

Also note, if you unlock a completely unlocked handle, it has no effect (ie: the lock count can't be negative).

If unlocked, then relocked, the memory associated with a memory handle may be at a different address, however the contents will remain the same.

See also *OfxImageEffectSuiteV1::imageMemoryLock* and ImageEffectsMemoryAllocation.

Return

- kOfxStatOK if the memory was unlocked cleanly,
- kOfxStatErrBadHandle if the value of *memoryHandle* was not a valid pointer returned by *OfxImageEffectSuiteV1::imageMemoryAlloc*, null is placed in **returnedPtr*

1.18.3 OfxProgressSuiteV1

struct **`OfxProgressSuiteV1`**

A suite that provides progress feedback from a plugin to an application.

A plugin instance can initiate, update and close a progress indicator with this suite.

This is an optional suite in the Image Effect API.

API V1.4: Amends the documentation of progress suite V1 so that it is expected that it can be raised in a modal manner and have a "cancel" button when invoked in instanceChanged. Plugins that perform analysis post an appropriate message, raise the progress monitor in a modal manner and should poll to see if processing has been aborted. Any cancellation should be handled gracefully by the plugin (eg: reset analysis parameters to default values), clear allocated memory…

Many hosts already operate as described above. kOfxStatReplyNo should be returned to the plugin during progressUpdate when the user presses cancel.

Suite V2: Adds an ID that can be looked up for internationalisation and so on. When a new version is introduced, because plug-ins need to support old versions, and plug-in's new releases are not necessary in synch with hosts (or users don't immediately update), best practice is to support the 2 suite versions. That is, the plugin should check if V2 exists; if not then check if V1 exists. This way a graceful transition is guaranteed. So plugin should fetchSuite passing 2, (OfxProgressSuiteV2*) fetchSuite(mHost->mHost->host, kOfxProgressSuite,2); and if no success pass (OfxProgressSuiteV1*) fetchSuite(mHost->mHost->host, kOfxProgressSuite,1);

Public Members

OfxStatus (***`progressStart`**)(void *effectInstance, const char *label)

Initiate a progress bar display.

Call this to initiate the display of a progress bar.

- *effectInstance* - the instance of the plugin this progress bar is associated with. It cannot be NULL.
- *label* - a text label to display in any message portion of the progress object's user interface. A UTF8 string.

Pre - There is no currently ongoing progress display for this instance.

Return

- *kOfxStatOK* - the handle is now valid for use
- *kOfxStatFailed* - the progress object failed for some reason
- *kOfxStatErrBadHandle* - effectInstance was invalid

OfxStatus (***progressUpdate**)(void *effectInstance, double progress)
Indicate how much of the processing task has been completed and reports on any abort status.

- *effectInstance* - the instance of the plugin this progress bar is associated with. It cannot be NULL.
- *progress* - a number between 0.0 and 1.0 indicating what proportion of the current task has been processed.

Return

- *kOfxStatOK* - the progress object was successfully updated and the task should continue
- *kOfxStatReplyNo* - the progress object was successfully updated and the task should abort
- *kOfxStatErrBadHandle* - the progress handle was invalid,

OfxStatus (***progressEnd**)(void *effectInstance)
Signal that we are finished with the progress meter.

Call this when you are done with the progress meter and no longer need it displayed.

- *effectInstance* - the instance of the plugin this progress bar is associated with. It cannot be NULL.

Post - you can no longer call progressUpdate on the instance

Return

- *kOfxStatOK* - the progress object was successfully closed
- *kOfxStatErrBadHandle* - the progress handle was invalid,

1.18.4 OfxTimeLineSuiteV1

struct **OfxImageEffectSuiteV1**
The OFX suite for image effects.

This suite provides the functions needed by a plugin to defined and use an image effect plugin.

Public Members

OfxStatus (***getPropertySet**)(*OfxImageEffectHandle* imageEffect, *OfxPropertySetHandle* *propHandle)
Retrieves the property set for the given image effect.

- imageEffect image effect to get the property set for
- propHandle pointer to a the property set pointer, value is returned here

The property handle is for the duration of the image effect handle.

Return

- *kOfxStatOK* - the property set was found and returned
- *kOfxStatErrBadHandle* - if the paramter handle was invalid
- *kOfxStatErrUnknown* - if the type is unknown

OfxStatus (***getParamSet**)(*OfxImageEffectHandle* imageEffect, OfxParamSetHandle *paramSet)
Retrieves the parameter set for the given image effect.

- imageEffect image effect to get the property set for
- paramSet pointer to a the parameter set, value is returned here

The param set handle is valid for the lifetime of the image effect handle.

Return

- *kOfxStatOK* - the property set was found and returned
- *kOfxStatErrBadHandle* - if the paramter handle was invalid
- *kOfxStatErrUnknown* - if the type is unknown

OfxStatus (***clipDefine**)(*OfxImageEffectHandle* imageEffect, const char *name, *OfxPropertySetHandle* *propertySet)
Define a clip to the effect.

- pluginHandle - the handle passed into 'describeInContext' action
- name - unique name of the clip to define
- propertySet - a property handle for the clip descriptor will be returned here

This function defines a clip to a host, the returned property set is used to describe various aspects of the clip to the host. Note that this does not create a clip instance.

Pre

- we are inside the describe in context action.

Return

OfxStatus (***clipGetHandle**)(*OfxImageEffectHandle* imageEffect, const char *name, *OfxImageClipHandle* *clip, *OfxPropertySetHandle* *propertySet)
Get the propery handle of the named input clip in the given instance.

- imageEffect - an instance handle to the plugin
- name - name of the clip, previously used in a clip define call
- clip - where to return the clip
- propertySet if not null, the descriptor handle for a parameter's property set will be placed here.

The propertySet will have the same value as would be returned by *OfxImageEffectSuiteV1::clipGetPropertySet*

```
This return a clip handle for the given instance, note that this will \em not␣
↪be the same as the
clip handle returned by clipDefine and will be distanct to clip handles in any␣
↪other instance
of the plugin.

Not a valid call in any of the describe actions.
```

Pre

- create instance action called,
- *name* passed to clipDefine for this context,
- not inside describe or describe in context actions.

Post

- handle will be valid for the life time of the instance.

OfxStatus (***clipGetPropertySet**)(*OfxImageClipHandle* clip, *OfxPropertySetHandle* *propHandle)

Retrieves the property set for a given clip.

- clip clip effect to get the property set for
- propHandle pointer to a the property set handle, value is returedn her

The property handle is valid for the lifetime of the clip, which is generally the lifetime of the instance.

Return

- *kOfxStatOK* - the property set was found and returned
- *kOfxStatErrBadHandle* - if the paramter handle was invalid
- *kOfxStatErrUnknown* - if the type is unknown

OfxStatus (***clipGetImage**)(*OfxImageClipHandle* clip, OfxTime time, const *OfxRectD* *region, *OfxPropertySetHandle* *imageHandle)

Get a handle for an image in a clip at the indicated time and indicated region.

- clip - the clip to extract the image from
- time - time to fetch the image at
- region - region to fetch the image from (optional, set to NULL to get a 'default' region) this is in the CanonicalCoordinates.

- imageHandle - property set containing the image's data

An image is fetched from a clip at the indicated time for the given region and returned in the imageHandle.

If the *region* parameter is not set to NULL, then it will be clipped to the clip's Region of Definition for the given time. The returned image will be *at least* as big as this region. If the region parameter is not set, then the region fetched will be at least the Region of Interest the effect has previously specified, clipped the clip's Region of Definition.

If clipGetImage is called twice with the same parameters, then two separate image handles will be returned, each of which must be release. The underlying implementation could share image data pointers and use reference counting to maintain them.

Pre

- clip was returned by clipGetHandle

Post

- image handle is only valid for the duration of the action clipGetImage is called in
- image handle to be disposed of by clipReleaseImage before the action returns

Return

- *kOfxStatOK* - the image was successfully fetched and returned in the handle,
- *kOfxStatFailed* - the image could not be fetched because it does not exist in the clip at the indicated time and/or region, the plugin should continue operation, but assume the image was black and transparent.
- *kOfxStatErrBadHandle* - the clip handle was invalid,
- *kOfxStatErrMemory* - the host had not enough memory to complete the operation, plugin should abort whatever it was doing.

OfxStatus (*`clipReleaseImage`)(*OfxPropertySetHandle* imageHandle)
Releases the image handle previously returned by clipGetImage.

Pre

- imageHandle was returned by clipGetImage

Post

- all operations on imageHandle will be invalid

Return

- *kOfxStatOK* - the image was successfully fetched and returned in the handle,
- *kOfxStatErrBadHandle* - the image handle was invalid,

OfxStatus (*`clipGetRegionOfDefinition`)(*OfxImageClipHandle* clip, OfxTime time, *OfxRectD* *bounds)
Returns the spatial region of definition of the clip at the given time.

- clipHandle - the clip to extract the image from
- time - time to fetch the image at
- region - region to fetch the image from (optional, set to NULL to get a 'default' region) this is in the CanonicalCoordinates.
- imageHandle - handle where the image is returned

An image is fetched from a clip at the indicated time for the given region and returned in the imageHandle.

If the *region* parameter is not set to NULL, then it will be clipped to the clip's Region of Definition for the given time. The returned image will be *at least* as big as this region. If the region parameter is not set, then the region fetched will be at least the Region of Interest the effect has previously specified, clipped the clip's Region of Definition.

Pre

- clipHandle was returned by clipGetHandle

Post

- bounds will be filled the RoD of the clip at the indicated time

Return

- *kOfxStatOK* - the image was successfully fetched and returned in the handle,
- *kOfxStatFailed* - the image could not be fetched because it does not exist in the clip at the indicated time, the plugin should continue operation, but assume the image was black and transparent.
- *kOfxStatErrBadHandle* - the clip handle was invalid,
- *kOfxStatErrMemory* - the host had not enough memory to complete the operation, plugin should abort whatever it was doing.

int (***abort**)(*OfxImageEffectHandle* imageEffect)

Returns whether to abort processing or not.

- imageEffect - instance of the image effect

A host may want to signal to a plugin that it should stop whatever rendering it is doing and start again. Generally this is done in interactive threads in response to users tweaking some parameter.

This function indicates whether a plugin should stop whatever processing it is doing.

Return

- 0 if the effect should continue whatever processing it is doing
- 1 if the effect should abort whatever processing it is doing

OfxStatus (***imageMemoryAlloc**)(*OfxImageEffectHandle* instanceHandle, size_t nBytes, OfxImageMemoryHandle *memoryHandle)

Allocate memory from the host's image memory pool.

- instanceHandle - effect instance to associate with this memory allocation, may be NULL.
- nBytes - the number of bytes to allocate
- memoryHandle - pointer to the memory handle where a return value is placed

Memory handles allocated by this should be freed by *OfxImageEffectSuiteV1::imageMemoryFree*. To access the memory behind the handle you need to call *OfxImageEffectSuiteV1::imageMemoryLock*.

See ImageEffectsMemoryAllocation.

Return

- kOfxStatOK if all went well, a valid memory handle is placed in *memoryHandle*

- kOfxStatErrBadHandle if instanceHandle is not valid, memoryHandle is set to NULL
- kOfxStatErrMemory if there was not enough memory to satisfy the call, memoryHandle is set to NULL

OfxStatus (***imageMemoryFree**)(OfxImageMemoryHandle memoryHandle)

Frees a memory handle and associated memory.

- memoryHandle - memory handle returned by imageMemoryAlloc

This function frees a memory handle and associated memory that was previously allocated via *OfxImageEffectSuiteV1::imageMemoryAlloc*

If there are outstanding locks, these are ignored and the handle and memory are freed anyway.

See ImageEffectsMemoryAllocation.

Return

- kOfxStatOK if the memory was cleanly deleted
- kOfxStatErrBadHandle if the value of *memoryHandle* was not a valid pointer returned by *OfxImageEffectSuiteV1::imageMemoryAlloc*

OfxStatus (***imageMemoryLock**)(OfxImageMemoryHandle memoryHandle, void **returnedPtr)

Lock the memory associated with a memory handle and make it available for use.

- memoryHandle - memory handle returned by imageMemoryAlloc
- returnedPtr - where to the pointer to the locked memory

This function locks them memory associated with a memory handle and returns a pointer to it. The memory will be 16 byte aligned, to allow use of vector operations.

Note that memory locks and unlocks nest.

After the first lock call, the contents of the memory pointer to by *returnedPtr* is undefined. All subsequent calls to lock will return memory with the same contents as the previous call.

Also, if unlocked, then relocked, the memory associated with a memory handle may be at a different address.

See also *OfxImageEffectSuiteV1::imageMemoryUnlock* and ImageEffectsMemoryAllocation.

Return

- kOfxStatOK if the memory was locked, a pointer is placed in *returnedPtr*
- kOfxStatErrBadHandle if the value of *memoryHandle* was not a valid pointer returned by *OfxImageEffectSuiteV1::imageMemoryAlloc*, null is placed in **returnedPtr*
- kOfxStatErrMemory if there was not enough memory to satisfy the call, **returnedPtr* is set to NULL

OfxStatus (***imageMemoryUnlock**)(OfxImageMemoryHandle memoryHandle)

Unlock allocated image data.

- allocatedData - pointer to memory previously returned by OfxImageEffectSuiteV1::imageAlloc

This function unlocks a previously locked memory handle. Once completely unlocked, memory associated with a memoryHandle is no longer available for use. Attempting to use it results in undefined behaviour.

Note that locks and unlocks nest, and to fully unlock memory you need to match the count of locks placed upon it.

Also note, if you unlock a completely unlocked handle, it has no effect (ie: the lock count can't be negative).

If unlocked, then relocked, the memory associated with a memory handle may be at a different address, however the contents will remain the same.

See also *OfxImageEffectSuiteV1::imageMemoryLock* and ImageEffectsMemoryAllocation.

Return

- kOfxStatOK if the memory was unlocked cleanly,
- kOfxStatErrBadHandle if the value of *memoryHandle* was not a valid pointer returned by *OfxImageEffectSuiteV1::imageMemoryAlloc*, null is placed in **returnedPtr*

1.18.5 OfxParameterSuiteV1

struct **OfxParameterSuiteV1**

The OFX suite used to define and manipulate user visible parameters.

Keyframe Handling

These functions allow the plug-in to delete and get information about keyframes.

To set keyframes, use *paramSetValueAtTime()*.

paramGetKeyTime and paramGetKeyIndex use indices to refer to keyframes. Keyframes are stored by the host in increasing time order, so time(kf[i]) < time(kf[i+1]). Keyframe indices will change whenever keyframes are added, deleted, or moved in time, whether by the host or by the plug-in. They may vary between actions if the user changes a keyframe. The keyframe indices will not change within a single action.

OfxStatus (***paramGetNumKeys**)(*OfxParamHandle* paramHandle, unsigned int *numberOfKeys)

Returns the number of keyframes in the parameter.

- paramHandle parameter handle to interogate
- numberOfKeys pointer to integer where the return value is placed

V1.3: This function can be called the *kOfxActionInstanceChanged* action and during image effect analysis render passes. V1.4: This function can be called the *kOfxActionInstanceChanged* action

Returns the number of keyframes in the parameter.

Return

- *kOfxStatOK* - all was OK
- *kOfxStatErrBadHandle* - if the parameter handle was invalid

OfxStatus (***paramGetKeyTime**)(*OfxParamHandle* paramHandle, unsigned int nthKey, OfxTime *time)

Returns the time of the nth key.

- paramHandle parameter handle to interogate
- nthKey which key to ask about (0 to paramGetNumKeys -1), ordered by time
- time pointer to OfxTime where the return value is placed

Return

- *kOfxStatOK* - all was OK
- *kOfxStatErrBadHandle* - if the parameter handle was invalid
- *kOfxStatErrBadIndex* - the nthKey does not exist

OfxStatus (***paramGetKeyIndex**)(*OfxParamHandle* paramHandle, OfxTime time, int direction, int *index)
Finds the index of a keyframe at/before/after a specified time.

- paramHandle parameter handle to search
- time what time to search from
- direction
  - == 0 indicates search for a key at the indicated time (some small delta)
  - > 0 indicates search for the next key after the indicated time
  - < 0 indicates search for the previous key before the indicated time
- index pointer to an integer which in which the index is returned set to -1 if no key was found

Return

- *kOfxStatOK* - all was OK
- *kOfxStatFailed* - if the search failed to find a key
- *kOfxStatErrBadHandle* - if the parameter handle was invalid

OfxStatus (***paramDeleteKey**)(*OfxParamHandle* paramHandle, OfxTime time)
Deletes a keyframe if one exists at the given time.

- paramHandle parameter handle to delete the key from
- time time at which a keyframe is

Return

- *kOfxStatOK* - all was OK
- *kOfxStatErrBadHandle* - if the parameter handle was invalid
- *kOfxStatErrBadIndex* - no key at the given time

OfxStatus (***paramDeleteAllKeys**)(*OfxParamHandle* paramHandle)
Deletes all keyframes from a parameter.

- paramHandle parameter handle to delete the keys from

- name parameter to delete the keyframes frome is

V1.3: This function can be called the *kOfxActionInstanceChanged* action and during image effect analysis render passes. V1.4: This function can be called the *kOfxActionInstanceChanged* action

Return

- *kOfxStatOK* - all was OK
- *kOfxStatErrBadHandle* - if the parameter handle was invalid

Public Members

OfxStatus (***paramDefine**)(OfxParamSetHandle paramSet, const char *paramType, const char *name, *OfxPropertySetHandle* *propertySet)

Defines a new parameter of the given type in a describe action.

- paramSet handle to the parameter set descriptor that will hold this parameter
- paramType type of the parameter to create, one of the kOfxParamType* #defines
- name unique name of the parameter
- propertySet if not null, a pointer to the parameter descriptor's property set will be placed here.

This function defines a parameter in a parameter set and returns a property set which is used to describe that parameter.

This function does not actually create a parameter, it only says that one should exist in any subsequent instances. To fetch an parameter instance paramGetHandle must be called on an instance.

This function can always be called in one of a plug-in's "describe" functions which defines the parameter sets common to all instances of a plugin.

Return

- *kOfxStatOK* - the parameter was created correctly
- *kOfxStatErrBadHandle* - if the plugin handle was invalid
- *kOfxStatErrExists* - if a parameter of that name exists already in this plugin
- *kOfxStatErrUnknown* - if the type is unknown
- *kOfxStatErrUnsupported* - if the type is known but unsupported

OfxStatus (***paramGetHandle**)(OfxParamSetHandle paramSet, const char *name, *OfxParamHandle* *param, *OfxPropertySetHandle* *propertySet)

Retrieves the handle for a parameter in a given parameter set.

- paramSet instance of the plug-in to fetch the property handle from
- name parameter to ask about
- param pointer to a param handle, the value is returned here
- propertySet if not null, a pointer to the parameter's property set will be placed here.

Parameter handles retrieved from an instance are always distinct in each instance. The paramter handle is valid for the life-time of the instance. Parameter handles in instances are distinct from paramter handles in plugins. You cannot call this in a plugin's describe function, as it needs an instance to work on.

Return

- *kOfxStatOK* - the parameter was found and returned
- *kOfxStatErrBadHandle* - if the plugin handle was invalid
- *kOfxStatErrUnknown* - if the type is unknown

OfxStatus (***paramSetGetPropertySet**)(OfxParamSetHandle paramSet, *OfxPropertySetHandle* *propHandle)

Retrieves the property set handle for the given parameter set.

- paramSet parameter set to get the property set for
- propHandle pointer to a the property set handle, value is returedn her

Note: The property handle belonging to a parameter set is the same as the property handle belonging to the plugin instance.

Return

- *kOfxStatOK* - the property set was found and returned
- *kOfxStatErrBadHandle* - if the paramter handle was invalid
- *kOfxStatErrUnknown* - if the type is unknown

OfxStatus (***paramGetPropertySet**)(*OfxParamHandle* param, *OfxPropertySetHandle* *propHandle)

Retrieves the property set handle for the given parameter.

- param parameter to get the property set for
- propHandle pointer to a the property set handle, value is returedn her

The property handle is valid for the lifetime of the parameter, which is the lifetime of the instance that owns the parameter

Return

- *kOfxStatOK* - the property set was found and returned
- *kOfxStatErrBadHandle* - if the paramter handle was invalid
- *kOfxStatErrUnknown* - if the type is unknown

OfxStatus (***paramGetValue**)(*OfxParamHandle* paramHandle, ...)

Gets the current value of a parameter,.

- paramHandle parameter handle to fetch value from
- … one or more pointers to variables of the relevant type to hold the parameter's value

This gets the current value of a parameter. The varargs … argument needs to be pointer to C variables of the relevant type for this parameter. Note that params with multiple values (eg Colour) take multiple args here. For example…

```
OfxParamHandle myDoubleParam, *myColourParam;
ofxHost->paramGetHandle(instance, "myDoubleParam", &myDoubleParam);
double myDoubleValue;
ofxHost->paramGetValue(myDoubleParam, &myDoubleValue);
ofxHost->paramGetHandle(instance, "myColourParam", &myColourParam);
double myR, myG, myB;
ofxHost->paramGetValue(myColourParam, &myR, &myG, &myB);
```

Note: paramGetValue should only be called from within a *kOfxActionInstanceChanged* or interact action and never from the render actions (which should always use paramGetValueAtTime).

Return

- *kOfxStatOK* - all was OK
- *kOfxStatErrBadHandle* - if the parameter handle was invalid

OfxStatus (***paramGetValueAtTime**)(*OfxParamHandle* paramHandle, OfxTime time, ...)

Gets the value of a parameter at a specific time.

- paramHandle parameter handle to fetch value from
- time at what point in time to look up the parameter
- ... one or more pointers to variables of the relevant type to hold the parameter's value

This gets the current value of a parameter. The varargs needs to be pointer to C variables of the relevant type for this parameter. See *OfxParameterSuiteV1::paramGetValue* for notes on the varags list

Return

- *kOfxStatOK* - all was OK
- *kOfxStatErrBadHandle* - if the parameter handle was invalid

OfxStatus (***paramGetDerivative**)(*OfxParamHandle* paramHandle, OfxTime time, ...)

Gets the derivative of a parameter at a specific time.

- paramHandle parameter handle to fetch value from
- time at what point in time to look up the parameter
- ... one or more pointers to variables of the relevant type to hold the parameter's derivative

This gets the derivative of the parameter at the indicated time.

The varargs needs to be pointer to C variables of the relevant type for this parameter. See *OfxParameter-SuiteV1::paramGetValue* for notes on the varags list.

Only double and colour params can have their derivatives found.

Return

- *kOfxStatOK* - all was OK

- *kOfxStatErrBadHandle* - if the parameter handle was invalid

OfxStatus (***paramGetIntegral**)(*OfxParamHandle* paramHandle, OfxTime time1, OfxTime time2, ...)

Gets the integral of a parameter over a specific time range,.

- paramHandle parameter handle to fetch integral from
- time1 where to start evaluating the integral
- time2 where to stop evaluating the integral
- … one or more pointers to variables of the relevant type to hold the parameter's integral

This gets the integral of the parameter over the specified time range.

The varargs needs to be pointer to C variables of the relevant type for this parameter. See *OfxParameterSuiteV1::paramGetValue* for notes on the varags list.

Only double and colour params can be integrated.

Return

- *kOfxStatOK* - all was OK
- *kOfxStatErrBadHandle* - if the parameter handle was invalid

OfxStatus (***paramSetValue**)(*OfxParamHandle* paramHandle, ...)

Sets the current value of a parameter.

- paramHandle parameter handle to set value in
- … one or more variables of the relevant type to hold the parameter's value

This sets the current value of a parameter. The varargs … argument needs to be values of the relevant type for this parameter. Note that params with multiple values (eg Colour) take multiple args here. For example…

```
ofxHost->paramSetValue(instance, "myDoubleParam", double(10));
ofxHost->paramSetValue(instance, "myColourParam", double(pix.r), double(pix.
↪g), double(pix.b));
```

Note: paramSetValue should only be called from within a *kOfxActionInstanceChanged* or interact action.

Return

- *kOfxStatOK* - all was OK
- *kOfxStatErrBadHandle* - if the parameter handle was invalid

OfxStatus (***paramSetValueAtTime**)(*OfxParamHandle* paramHandle, OfxTime time, ...)

Keyframes the value of a parameter at a specific time.

- paramHandle parameter handle to set value in
- time at what point in time to set the keyframe

- … one or more variables of the relevant type to hold the parameter's value

This sets a keyframe in the parameter at the indicated time to have the indicated value. The varargs … argument needs to be values of the relevant type for this parameter. See the note on *OfxParameterSuiteV1::paramSetValue* for more detail

V1.3: This function can be called the *kOfxActionInstanceChanged* action and during image effect analysis render passes. V1.4: This function can be called the *kOfxActionInstanceChanged* action

Note: paramSetValueAtTime should only be called from within a *kOfxActionInstanceChanged* or interact action.

Return

- *kOfxStatOK* - all was OK
- *kOfxStatErrBadHandle* - if the parameter handle was invalid

OfxStatus (***paramCopy**)(*OfxParamHandle* paramTo, *OfxParamHandle* paramFrom, OfxTime dstOffset, const OfxRangeD *frameRange)

Copies one parameter to another, including any animation etc…

- paramTo parameter to set
- paramFrom parameter to copy from
- dstOffset temporal offset to apply to keys when writing to the paramTo
- frameRange if paramFrom has animation, and frameRange is not null, only this range of keys will be copied

This copies the value of *paramFrom* to *paramTo*, including any animation it may have. All the previous values in *paramTo* will be lost.

To choose all animation in *paramFrom* set *frameRange* to [0, 0]

V1.3: This function can be called the *kOfxActionInstanceChanged* action and during image effect analysis render passes. V1.4: This function can be called the *kOfxActionInstanceChanged* action

Pre

- Both parameters must be of the same type.

Return

- *kOfxStatOK* - all was OK
- *kOfxStatErrBadHandle* - if the parameter handle was invalid

OfxStatus (***paramEditBegin**)(OfxParamSetHandle paramSet, const char *name)

Used to group any parameter changes for undo/redo purposes.

- paramSet the parameter set in which this is happening
- name label to attach to any undo/redo string UTF8

If a plugin calls paramSetValue/paramSetValueAtTime on one or more parameters, either from custom GUI interaction or some analysis of imagery etc.. this is used to indicate the start of a set of a parameter changes that should be considered part of a single undo/redo block.

See also *OfxParameterSuiteV1::paramEditEnd*

Note: paramEditBegin should only be called from within a *kOfxActionInstanceChanged* or interact action.

Return

- *kOfxStatOK* - all was OK
- *kOfxStatErrBadHandle* - if the instance handle was invalid

OfxStatus (***paramEditEnd**)(OfxParamSetHandle paramSet)
Used to group any parameter changes for undo/redo purposes.

- paramSet the parameter set in which this is happening

If a plugin calls paramSetValue/paramSetValueAtTime on one or more parameters, either from custom GUI interaction or some analysis of imagery etc.. this is used to indicate the end of a set of parameter changes that should be considerred part of a single undo/redo block

See also *OfxParameterSuiteV1::paramEditBegin*

Note: paramEditEnd should only be called from within a *kOfxActionInstanceChanged* or interact action.

Return

- *kOfxStatOK* - all was OK
- *kOfxStatErrBadHandle* - if the instance handle was invalid

1.18.6 OfxParametricParameterSuiteV1

struct **OfxParametricParameterSuiteV1**
The OFX suite used to define and manipulate 'parametric' parameters.

This is an optional suite.

Parametric parameters are in effect 'functions' a plug-in can ask a host to arbitrarily evaluate for some value 'x'. A classic use case would be for constructing look-up tables, a plug-in would ask the host to evaluate one at multiple values from 0 to 1 and use that to fill an array.

A host would probably represent this to a user as a cubic curve in a standard curve editor interface, or possibly through scripting. The user would then use this to define the 'shape' of the parameter.

The evaluation of such params is not the same as animation, they are returning values based on some arbitrary argument orthogonal to time, so to evaluate such a param, you need to pass a parametric position and time.

Often, you would want such a parametric parameter to be multi-dimensional, for example, a colour look-up table might want three values, one for red, green and blue. Rather than declare three separate parametric parameters, it would be better to have one such parameter with multiple values in it.

The major complication with these parameters is how to allow a plug-in to set values, and defaults. The default default value of a parametric curve is to be an identity lookup. If a plugin wishes to set a different default value for a curve, it can use the suite to set key/value pairs on the *descriptor* of the param. When a new instance is made, it will have these curve values as a default.

Public Members

OfxStatus (***parametricParamGetValue**)(*OfxParamHandle* param, int curveIndex, OfxTime time, double parametricPosition, double *returnValue)

Evaluates a parametric parameter.

- param handle to the parametric parameter
- curveIndex which dimension to evaluate
- time the time to evaluate to the parametric param at
- parametricPosition the position to evaluate the parametric param at
- returnValue pointer to a double where a value is returned

Return

- *kOfxStatOK* - all was fine
- *kOfxStatErrBadHandle* - if the paramter handle was invalid
- *kOfxStatErrBadIndex* - the curve index was invalid

OfxStatus (***parametricParamGetNControlPoints**)(*OfxParamHandle* param, int curveIndex, double time, int *returnValue)

Returns the number of control points in the parametric param.

- param handle to the parametric parameter
- curveIndex which dimension to check
- time the time to check
- returnValue pointer to an integer where the value is returned.

Return

- *kOfxStatOK* - all was fine
- *kOfxStatErrBadHandle* - if the paramter handle was invalid
- *kOfxStatErrBadIndex* - the curve index was invalid

OfxStatus (***parametricParamGetNthControlPoint**)(*OfxParamHandle* param, int curveIndex, double time, int nthCtl, double *key, double *value)

Returns the key/value pair of the nth control point.

- param handle to the parametric parameter
- curveIndex which dimension to check
- time the time to check
- nthCtl the nth control point to get the value of
- key pointer to a double where the key will be returned
- value pointer to a double where the value will be returned

Return

- *kOfxStatOK* - all was fine
- *kOfxStatErrBadHandle* - if the paramter handle was invalid
- *kOfxStatErrUnknown* - if the type is unknown

OfxStatus (*`parametricParamSetNthControlPoint`)(*OfxParamHandle* param, int curveIndex, double time, int nthCtl, double key, double value, bool addAnimationKey)

Modifies an existing control point on a curve.

- param handle to the parametric parameter
- curveIndex which dimension to set
- time the time to set the value at
- nthCtl the control point to modify
- key key of the control point
- value value of the control point
- addAnimationKey if the param is an animatable, setting this to true will force an animation keyframe to be set as well as a curve key, otherwise if false, a key will only be added if the curve is already animating.

This modifies an existing control point. Note that by changing key, the order of the control point may be modified (as you may move it before or after anther point). So be careful when iterating over a curves control points and you change a key.

Return

- *kOfxStatOK* - all was fine
- *kOfxStatErrBadHandle* - if the paramter handle was invalid
- *kOfxStatErrUnknown* - if the type is unknown

OfxStatus (*`parametricParamAddControlPoint`)(*OfxParamHandle* param, int curveIndex, double time, double key, double value, bool addAnimationKey)

Adds a control point to the curve.

- param handle to the parametric parameter
- curveIndex which dimension to set

- time the time to set the value at
- key key of the control point
- value value of the control point
- addAnimationKey if the param is an animatable, setting this to true will force an animation keyframe to be set as well as a curve key, otherwise if false, a key will only be added if the curve is already animating.

This will add a new control point to the given dimension of a parametric parameter. If a key exists sufficiently close to 'key', then it will be set to the indicated control point.

Return

- *kOfxStatOK* - all was fine
- *kOfxStatErrBadHandle* - if the paramter handle was invalid
- *kOfxStatErrUnknown* - if the type is unknown

OfxStatus (***parametricParamDeleteControlPoint**)(*OfxParamHandle* param, int curveIndex, int nthCtl)

Deletes the nth control point from a parametric param.

- param handle to the parametric parameter
- curveIndex which dimension to delete
- nthCtl the control point to delete

OfxStatus (***parametricParamDeleteAllControlPoints**)(*OfxParamHandle* param, int curveIndex)

Delete all curve control points on the given param.

- param handle to the parametric parameter
- curveIndex which dimension to clear

1.18.7 OfxMemorySuiteV1

struct **OfxMemorySuiteV1**

The OFX suite that implements general purpose memory management.

Use this suite for ordinary memory management functions, where you would normally use malloc/free or new/delete on ordinary objects.

For images, you should use the memory allocation functions in the image effect suite, as many hosts have specific image memory pools.

Note: C++ plugin developers will need to redefine new and delete as skins ontop of this suite.

Public Members

OfxStatus (***memoryAlloc**)(void *handle, size_t nBytes, void **allocatedData)

Allocate memory.

- handle - effect instance to assosciate with this memory allocation, or NULL.
- nBytes - the number of bytes to allocate
- allocatedData - a pointer to the return value. Allocated memory will be alligned for any use.

This function has the host allocate memory using its own memory resources and returns that to the plugin.

Return

- *kOfxStatOK* the memory was sucessfully allocated
- *kOfxStatErrMemory* the request could not be met and no memory was allocated

OfxStatus (***memoryFree**)(void *allocatedData)

Frees memory.

- allocatedData - pointer to memory previously returned by *OfxMemorySuiteV1::memoryAlloc*

This function frees any memory that was previously allocated via *OfxMemorySuiteV1::memoryAlloc*.

Return

- *kOfxStatOK* the memory was sucessfully freed
- *kOfxStatErrBadHandle* *allocatedData* was not a valid pointer returned by *OfxMemory-SuiteV1::memoryAlloc*

1.18.8 OfxMultiThreadSuiteV1

struct **OfxMultiThreadSuiteV1**

OFX suite that provides simple SMP style multi-processing.

Public Members

OfxStatus (***multiThread**)(OfxThreadFunctionV1 func, unsigned int nThreads, void *customArg)

Function to spawn SMP threads.

- func The function to call in each thread.
- nThreads The number of threads to launch
- customArg The paramter to pass to customArg of func in each thread.

This function will spawn nThreads separate threads of computation (typically one per CPU) to allow something to perform symmetric multi processing. Each thread will call 'func' passing in the index of the thread and the number of threads actually launched.

multiThread will not return until all the spawned threads have returned. It is up to the host how it waits for all the threads to return (busy wait, blocking, whatever).

nThreads can be more than the value returned by multiThreadNumCPUs, however the threads will be limitted to the number of CPUs returned by multiThreadNumCPUs.

This function cannot be called recursively.

Return

- *kOfxStatOK*, the function func has executed and returned sucessfully
- *kOfxStatFailed*, the threading function failed to launch
- *kOfxStatErrExists*, failed in an attempt to call multiThread recursively,

OfxStatus (***multiThreadNumCPUs**)(unsigned int *nCPUs)
Function which indicates the number of CPUs available for SMP processing.

- nCPUs pointer to an integer where the result is returned

This value may be less than the actual number of CPUs on a machine, as the host may reserve other CPUs for itself.

Return

- *kOfxStatOK*, all was OK and the maximum number of threads is in nThreads.
- *kOfxStatFailed*, the function failed to get the number of CPUs

OfxStatus (***multiThreadIndex**)(unsigned int *threadIndex)
Function which indicates the index of the current thread.

- threadIndex pointer to an integer where the result is returned

This function returns the thread index, which is the same as the *threadIndex* argument passed to the OfxThreadFunctionV1.

If there are no threads currently spawned, then this function will set threadIndex to 0

Return

- *kOfxStatOK*, all was OK and the maximum number of threads is in nThreads.
- *kOfxStatFailed*, the function failed to return an index

int (***multiThreadIsSpawnedThread**)(void)
Function to enquire if the calling thread was spawned by multiThread.

Return

- 0 if the thread is not one spawned by multiThread
- 1 if the thread was spawned by multiThread

OfxStatus (***mutexCreate**)(OfxMutexHandle *mutex, int lockCount)
Create a mutex.

- mutex - where the new handle is returned

- count - initial lock count on the mutex. This can be negative.

Creates a new mutex with lockCount locks on the mutex intially set.

Return

- kOfxStatOK - mutex is now valid and ready to go

OfxStatus (***mutexDestroy**)(const OfxMutexHandle mutex)

Destroy a mutex.

Destroys a mutex intially created by mutexCreate.

Return

- kOfxStatOK - if it destroyed the mutex
- kOfxStatErrBadHandle - if the handle was bad

OfxStatus (***mutexLock**)(const OfxMutexHandle mutex)

Blocking lock on the mutex.

This trys to lock a mutex and blocks the thread it is in until the lock suceeds.

A sucessful lock causes the mutex's lock count to be increased by one and to block any other calls to lock the mutex until it is unlocked.

Return

- kOfxStatOK - if it got the lock
- kOfxStatErrBadHandle - if the handle was bad

OfxStatus (***mutexUnLock**)(const OfxMutexHandle mutex)

Unlock the mutex.

This unlocks a mutex. Unlocking a mutex decreases its lock count by one.

Return

- kOfxStatOK if it released the lock
- kOfxStatErrBadHandle if the handle was bad

OfxStatus (***mutexTryLock**)(const OfxMutexHandle mutex)

Non blocking attempt to lock the mutex.

This attempts to lock a mutex, if it cannot, it returns and says so, rather than blocking.

A sucessful lock causes the mutex's lock count to be increased by one, if the lock did not suceed, the call returns immediately and the lock count remains unchanged.

Return

- kOfxStatOK - if it got the lock
- kOfxStatFailed - if it did not get the lock
- kOfxStatErrBadHandle - if the handle was bad

1.18.9 OfxInteractSuiteV1

struct **`OfxInteractSuiteV1`**

OFX suite that allows an effect to interact with an openGL window so as to provide custom interfaces.

Public Members

OfxStatus (***`interactSwapBuffers`**)(*OfxInteractHandle* interactInstance)

Requests an openGL buffer swap on the interact instance.

OfxStatus (***`interactRedraw`**)(*OfxInteractHandle* interactInstance)

Requests a redraw of the interact instance.

OfxStatus (***`interactGetPropertySet`**)(*OfxInteractHandle* interactInstance, *OfxPropertySetHandle* *property)

Gets the property set handle for this interact handle.

1.18.10 OfxMessageSuiteV1

struct **`OfxMessageSuiteV1`**

The OFX suite that allows a plug-in to pass messages back to a user. The V2 suite extends on this in a backwards compatible manner.

Public Members

OfxStatus (***`message`**)(void *handle, const char *messageType, const char *messageId, const char *format, ...)

Post a message on the host, using printf style varargs.

- handle - effect handle (descriptor or instance) the message should be associated with, may be null
- messageType - string describing the kind of message to post, one of the kOfxMessageType* constants
- messageId - plugin specified id to associate with this message. If overriding the message in XML resource, the message is identified with this, this may be NULL, or “”, in which case no override will occur,
- format - printf style format string
- … - printf style varargs list to print

Return

- *kOfxStatOK* - if the message was sucessfully posted
- *kOfxStatReplyYes* - if the message was of type kOfxMessageQuestion and the user reply yes
- *kOfxStatReplyNo* - if the message was of type kOfxMessageQuestion and the user reply no
- *kOfxStatFailed* - if the message could not be posted for some reason

1.18.11 OfxMessageSuiteV2

struct **`OfxMessageSuiteV2`**

The OFX suite that allows a plug-in to pass messages back to a user.

This extends *OfxMessageSuiteV1*, and should be considered a replacement to version 1.

Note that this suite has been extended in backwards compatible manner, so that a host can return this struct for both V1 and V2.

Public Members

OfxStatus (***`message`**)(void *handle, const char *messageType, const char *messageId, const char *format, ...)

Post a transient message on the host, using printf style varargs. Same as the V1 message suite call.

- handle - effect handle (descriptor or instance) the message should be associated with, may be null
- messageType - string describing the kind of message to post, one of the kOfxMessageType* constants
- messageId - plugin specified id to associate with this message. If overriding the message in XML resource, the message is identified with this, this may be NULL, or "", in which case no override will occur,
- format - printf style format string
- ... - printf style varargs list to print

Return

- *kOfxStatOK* - if the message was sucessfully posted
- *kOfxStatReplyYes* - if the message was of type kOfxMessageQuestion and the user reply yes
- *kOfxStatReplyNo* - if the message was of type kOfxMessageQuestion and the user reply no
- *kOfxStatFailed* - if the message could not be posted for some reason

OfxStatus (***`setPersistentMessage`**)(void *handle, const char *messageType, const char *messageId, const char *format, ...)

Post a persistent message on an effect, using printf style varargs, and set error states. New for V2 message suite.

- handle - effect instance handle the message should be associated with, may NOT be null,
- messageType - string describing the kind of message to post, should be one of...
  - kOfxMessageError
  - kOfxMessageWarning
  - kOfxMessageMessage
- messageId - plugin specified id to associate with this message. If overriding the message in XML resource, the message is identified with this, this may be NULL, or "", in which case no override will occur,

- format - printf style format string
- … - printf style varargs list to print

Persistent messages are associated with an effect handle until explicitly cleared by an effect. So if an error message is posted the error state, and associated message will persist and be displayed on the effect appropriately. (eg: draw a node in red on a node based compostor and display the message when clicked on).

If *messageType* is error or warning, associated error states should be flagged on host applications. Posting an error message implies that the host cannot proceeed, a warning allows the host to proceed, whilst a simple message should have no stop anything.

Return

- *kOfxStatOK* - if the message was sucessfully posted
- *kOfxStatErrBadHandle* - the handle was rubbish
- *kOfxStatFailed* - if the message could not be posted for some reason

OfxStatus (*`clearPersistentMessage`)(void *handle)

Clears any persistent message on an effect handle that was set by *OfxMessageSuiteV2::setPersistentMessage*. New for V2 message suite.

- handle - effect instance handle messages should be cleared from.
- handle - effect handle (descriptor or instance)

Clearing a message will clear any associated error state.

Return

- *kOfxStatOK* - if the message was sucessfully cleared
- *kOfxStatErrBadHandle* - the handle was rubbish
- *kOfxStatFailed* - if the message could not be cleared for some reason

1.18.12 OfxImageEffectOpenGLRenderSuiteV1

struct `OfxImageEffectOpenGLRenderSuiteV1`

OFX suite that provides image to texture conversion for OpenGL processing.

Public Members

OfxStatus (*`clipLoadTexture`)(*OfxImageClipHandle* clip, OfxTime time, const char *format, const *OfxRectD* *region, *OfxPropertySetHandle* *textureHandle)

loads an image from an OFX clip as a texture into OpenGL

- clip - the clip to load the image from
- time - effect time to load the image from
- format - the requested texture format (As in none,byte,word,half,float, etc..) When set to NULL, the host decides the format based on the plug-in's *kOfxOpenGLPropPixelDepth* setting.

- region - region of the image to load (optional, set to NULL to get a 'default' region) this is in the CanonicalCoordinates.
- textureHandle - a property set containing information about the texture

An image is fetched from a clip at the indicated time for the given region and loaded into an OpenGL texture. When a specific format is requested, the host ensures it gives the requested format. When the clip specified is the "Output" clip, the format is ignored and the host must bind the resulting texture as the current color buffer (render target). This may also be done prior to calling the *kOfxImageEffectActionRender* action. If the *region* parameter is set to non-NULL, then it will be clipped to the clip's Region of Definition for the given time. The returned image will be *at least* as big as this region. If the region parameter is not set or is NULL, then the region fetched will be at least the Region of Interest the effect has previously specified, clipped to the clip's Region of Definition. Information about the texture, including the texture index, is returned in the *textureHandle* argument. The properties on this handle will be…

- *kOfxImageEffectPropOpenGLTextureIndex*
- *kOfxImageEffectPropOpenGLTextureTarget*
- *kOfxImageEffectPropPixelDepth*
- *kOfxImageEffectPropComponents*
- *kOfxImageEffectPropPreMultiplication*
- *kOfxImageEffectPropRenderScale*
- *kOfxImagePropPixelAspectRatio*
- *kOfxImagePropBounds*
- *kOfxImagePropRegionOfDefinition*
- *kOfxImagePropRowBytes*
- *kOfxImagePropField*
- *kOfxImagePropUniqueIdentifier*

With the exception of the OpenGL specifics, these properties are the same as the properties in an image handle returned by clipGetImage in the image effect suite.

Note:

- this is the OpenGL equivalent of clipGetImage from *OfxImageEffectSuiteV1*

Pre

- clip was returned by clipGetHandle
- Format property in the texture handle

Post

- texture handle to be disposed of by clipFreeTexture before the action returns
- when the clip specified is the "Output" clip, the format is ignored and the host must bind the resulting texture as the current color buffer (render target). This may also be done prior to calling the render action.

Return

- *kOfxStatOK* - the image was successfully fetched and returned in the handle,

- *kOfxStatFailed* - the image could not be fetched because it does not exist in the clip at the indicated time and/or region, the plugin should continue operation, but assume the image was black and transparent.
- *kOfxStatErrBadHandle* - the clip handle was invalid,
- *kOfxStatErrMemory* - not enough OpenGL memory was available for the effect to load the texture. The plugin should abort the GL render and return *kOfxStatErrMemory*, after which the host can decide to retry the operation with CPU based processing.

OfxStatus (***`clipFreeTexture`**)(*OfxPropertySetHandle* textureHandle)

Releases the texture handle previously returned by clipLoadTexture.

For input clips, this also deletes the texture from OpenGL. This should also be called on the output clip; for the Output clip, it just releases the handle but does not delete the texture (since the host will need to read it).

Pre

- textureHandle was returned by clipGetImage

Post

- all operations on textureHandle will be invalid, and the OpenGL texture it referred to has been deleted (for source clips)

Return

- *kOfxStatOK* - the image was successfully fetched and returned in the handle,
- *kOfxStatFailed* - general failure for some reason,
- *kOfxStatErrBadHandle* - the image handle was invalid,

OfxStatus (***`flushResources`**)()

Request the host to minimize its GPU resource load.

When a plugin fails to allocate GPU resources, it can call this function to request the host to flush it's GPU resources if it holds any. After the function the plugin can try again to allocate resources which then might succeed if the host actually has released anything.

Pre

Post

- No changes to the plugin GL state should have been made.

Return

- *kOfxStatOK* - the host has actually released some resources,
- *kOfxStatReplyDefault* - nothing the host could do..

1.18.13 OfxDrawSuiteV1: Drawing Overlays

Added for OFX v1.5, Jan 2022.

See the source at ofxDrawSuite.h

struct **OfxDrawSuiteV1**

OFX suite that allows an effect to draw to a host-defined display context.

Public Members

OfxStatus (***getColour**)(OfxDrawContextHandle context, *OfxStandardColour* std_colour, OfxRGBAColourF *colour)

Retrieves the host's desired draw colour for.

- context - the draw context
- std_colour - the desired colour type
- colour - the returned RGBA colour

Return

- *kOfxStatOK* - the colour was returned

OfxStatus (***setColour**)(OfxDrawContextHandle context, const OfxRGBAColourF *colour)

Sets the current draw colour for future drawing operations.

- context - the draw context
- colour - the RGBA colour

Return

- *kOfxStatOK* - the colour was changed

OfxStatus (***setLineWidth**)(OfxDrawContextHandle context, float width)

Sets the current draw colour for future drawing operations.

- context - the draw context
- width - the line width - use 0 for a single pixel line or non-zero for a smooth line of the desired width

The host should adjust for screen density.

Return

- *kOfxStatOK* - the width was changed

OfxStatus (***setLineStipple**)(OfxDrawContextHandle context, *OfxDrawLineStipplePattern* pattern)

Sets the current line stipple pattern.

- context - the draw context
- pattern - the desired stipple pattern

  Return

  - *kOfxStatOK* - the pattern was changed

OfxStatus (***draw**)(OfxDrawContextHandle context, *OfxDrawPrimitive* primitive, const OfxPointD *points, int point_count)

Draws a primitive of the desired type.

- context - the draw context
- primitive - the desired primitive
- points - the array of points in the primitive
- point_count - the number of points in the array

The number of points required matches the equivalent OpenGL primitives. For an ellipse, point_count should be 2, and points should form the top-left and bottom-right of a rectangle to draw the ellipse inside

Return

- *kOfxStatOK* - the pattern was changed
- *kOfxStatErrValue* - point_count was not valid

OfxStatus (***drawText**)(OfxDrawContextHandle context, const char *text, const OfxPointD *pos, int alignment)

Draws text at the specified position in the current font size.

- context - the draw context
- text - the text to draw (UTF-8 encoded)
- pos - the position of the lower-left corner of the text baseline
- alignment - the text alignment flags (see kOfxDrawTextAlignment*)

The text font face and size are determined by the host.

Return

- *kOfxStatOK* - the text was drawn
- *kOfxStatErrValue* - text or pos were not defined

#defines

kOfxInteractPropDrawContext

The Draw Context handle.

- Type - pointer X 1
- Property Set - read only property on the inArgs of the following actions…
- *kOfxInteractActionDraw*

Enums

enum **OfxStandardColour**

Defines valid values for *OfxDrawSuiteV1::getColour*.

Values:

enumerator **kOfxStandardColourOverlayBackground**

enumerator **kOfxStandardColourOverlayActive**

enumerator **kOfxStandardColourOverlaySelected**

enumerator **kOfxStandardColourOverlayDeselected**

enumerator **kOfxStandardColourOverlayMarqueeFG**

enumerator **kOfxStandardColourOverlayMarqueeBG**

enumerator **kOfxStandardColourOverlayText**

enum **OfxDrawLineStipplePattern**

Defines valid values for *OfxDrawSuiteV1::setLineStipple*.

Values:

enumerator **kOfxDrawLineStipplePatternSolid**

enumerator **kOfxDrawLineStipplePatternDot** enumerator **`kOfxDrawLineStipplePatternDash`**

enumerator **`kOfxDrawLineStipplePatternAltDash`**

enumerator **`kOfxDrawLineStipplePatternDotDash`**

enum **`OfxDrawPrimitive`**

Defines valid values for *OfxDrawSuiteV1::draw*.

Values:

enumerator **`kOfxDrawPrimitiveLines`**

enumerator **`kOfxDrawPrimitiveLineStrip`**

enumerator **`kOfxDrawPrimitiveLineLoop`**

enumerator **`kOfxDrawPrimitiveRectangle`**

enumerator **`kOfxDrawPrimitivePolygon`**

enumerator **`kOfxDrawPrimitiveEllipse`**

enum **`OfxDrawTextAligment`**

Defines text alignment values for *OfxDrawSuiteV1::drawText*.

Values:

enumerator **`kOfxDrawTextAlignmentLeft`**

enumerator **`kOfxDrawTextAlignmentRight`**

enumerator **`kOfxDrawTextAlignmentTop`**

enumerator **`kOfxDrawTextAlignmentBottom`**

enumerator **`kOfxDrawTextAlignmentBaseline`** enumerator **`kOfxDrawTextAlignmentCenterH`**

enumerator **`kOfxDrawTextAlignmentCenterV`**

Warning: This section is outdated and should be properly generated automatically from source code instead of maintaining it aside

1.19 Properties by object reference

1.19.1 Properties on the Image Effect Host

- kOfxPropName - (read only) the globally unique name of the application, eg: "com.acmesofware.funkyCompositor"
- kOfxPropLabel - (read only) the user visible name of the appliaction,
- kOfxPropVersion - (read only) the version number of the host
- kOfxPropVersionLabel - (read only) a user readable version label
- kOfxImageEffectHostPropIsBackground - (read only) is the application a background renderer
- kOfxImageEffectPropSupportsOverlays - (read only) does the application support overlay interactive GUIs
- kOfxImageEffectPropSupportsMultiResolution - (read only) does the application support images of different sizes
- kOfxImageEffectPropSupportsTiles - (read only) does the application support image tiling
- kOfxImageEffectPropTemporalClipAccess - (read only) does the application allow random temporal access to source images
- kOfxImageEffectPropSupportedComponents - (read only) a list of supported colour components
- kOfxImageEffectPropSupportedContexts - (read only) a list of supported effect contexts
- kOfxImageEffectPropSupportsMultipleClipDepths - (read only) does the application allow inputs and output clips to have differing bit depths
- kOfxImageEffectPropSupportsMultipleClipPARs - (read only) does the application allow inputs and output clips to have differing pixel aspect ratios
- kOfxImageEffectPropSetableFrameRate - (read only) does the application allow an effect to change the frame rate of the output clip
- kOfxImageEffectPropSetableFielding - (read only) does the application allow an effect to change the fielding of the output clip
- kOfxParamHostPropSupportsCustomInteract - (read only) does the application
- kOfxParamHostPropSupportsStringAnimation - (read only) does the application allow the animation of string parameters
- kOfxParamHostPropSupportsChoiceAnimation - (read only) does the application allow the animation of choice parameters

- kOfxParamHostPropSupportsBooleanAnimation - (read only does the application allow the animation of boolean parameters)
- kOfxParamHostPropSupportsCustomAnimation - (read only) does the application allow the animation of custom parameters
- kOfxParamHostPropMaxParameters - (read only) the maximum number of parameters the application allows a plug-in to have
- kOfxParamHostPropMaxPages - (read only) the maximum number of parameter pages the application allows a plug-in to have
- kOfxParamHostPropPageRowColumnCount - (read only) the number of rows and columns on a page parameter
- kOfxPropHostOSHandle - (read only) a pointer to an OS specific application handle (eg: the root hWnd on Windows)
- kOfxParamHostPropSupportsParametricAnimation - (read only) does the host support animation of parametric parameters
- kOfxImageEffectInstancePropSequentialRender - (read only) does the host support sequential rendering
- kOfxImageEffectPropOpenGLRenderSupported - (read only) does the host support OpenGL accelerated rendering
- kOfxImageEffectPropRenderQualityDraft - (read only) does the host support draft quality rendering
- kOfxImageEffectHostPropNativeOrigin - (read only) native origin of the host

1.19.2 Properties on an Effect Descriptor

An image effect plugin (ie: that thing passed to the initial 'describe' action) has the following properties, these can only be set inside the 'describe' actions …

- kOfxPropType - (read only)
- kOfxPropLabel - (read/write)
- kOfxPropShortLabel - (read/write)
- kOfxPropLongLabel - (read/write)
- kOfxPropVersion - (read only) the version number of the plugin
- kOfxPropVersionLabel - (read only) a user readable version label
- kOfxPropPluginDescription - (read/write), a short description of the plugin
- kOfxImageEffectPropSupportedContexts - (read/write)
- kOfxImageEffectPluginPropGrouping - (read/write)
- kOfxImageEffectPluginPropSingleInstance - (read/write)
- kOfxImageEffectPluginRenderThreadSafety - (read/write)
- kOfxImageEffectPluginPropHostFrameThreading - (read/write)
- kOfxImageEffectPluginPropOverlayInteractV1 - (read/write)
- kOfxImageEffectPropSupportsMultiResolution - (read/write)
- kOfxImageEffectPropSupportsTiles - (read/write)
- kOfxImageEffectPropTemporalClipAccess - (read/write)
- kOfxImageEffectPropSupportedPixelDepths - (read/write)

- kOfxImageEffectPluginPropFieldRenderTwiceAlways - (read/write)
- kOfxImageEffectPropSupportsMultipleClipDepths - (read/write)
- kOfxImageEffectPropSupportsMultipleClipPARs - (read/write)
- kOfxImageEffectPluginRenderThreadSafety - (read/write)
- kOfxImageEffectPropClipPreferencesSlaveParam - (read/write)
- kOfxImageEffectPropOpenGLRenderSupported - (read and write)
- kOfxPluginPropFilePath (read only)

1.19.3 Properties on an Effect Instance

An image effect instance has the following properties, all but kOfxPropInstanceData and kOfxImageEffectInstance-PropSequentialRender are read only…

- kOfxPropType - (read only)
- kOfxImageEffectPropContext - (read only)
- kOfxPropInstanceData - (read and write)
- kOfxImageEffectPropProjectSize - (read only)
- kOfxImageEffectPropProjectOffset - (read only)
- kOfxImageEffectPropProjectExtent - (read only)
- kOfxImageEffectPropProjectPixelAspectRatio - (read only)
- kOfxImageEffectInstancePropEffectDuration - (read only)
- kOfxImageEffectInstancePropSequentialRender - (read and write)
- kOfxImageEffectPropSupportsTiles - (read/write)
- kOfxImageEffectPropOpenGLRenderSupported - (read and write)
- kOfxImageEffectPropFrameRate - (read only)
- kOfxPropIsInteractive - (read only)

1.19.4 Properties on a Clip Descriptor

All OfxImageClipHandle accessed inside the `kOfxActionDescribe` or `kOfxActionDescribeInContext` are clip descriptors, used to describe the behaviour of clips in a specific context.

- kOfxPropType - (read only) set to
- kOfxPropName - (read only) the name the clip was created with
- kOfxPropLabel - (read/write) the user visible label for the clip
- kOfxPropShortLabel - (read/write)
- kOfxPropLongLabel - (read/write)
- kOfxImageEffectPropSupportedComponents - (read/write)
- kOfxImageEffectPropTemporalClipAccess - (read/write)
- kOfxImageClipPropOptional - (read/write)

- kOfxImageClipPropFieldExtraction - (read/write)
- kOfxImageClipPropIsMask - (read/write)
- kOfxImageEffectPropSupportsTiles - (read/write)

1.19.5 Properties on a Clip Instance

- kOfxPropType - (read only)
- kOfxPropName - (read only)
- kOfxPropLabel - (read only)
- kOfxPropShortLabel - (read only)
- kOfxPropLongLabel - (read only)
- kOfxImageEffectPropSupportedComponents - (read only)
- kOfxImageEffectPropTemporalClipAccess - (read only)
- kOfxImageClipPropOptional - (read only)
- kOfxImageClipPropFieldExtraction - (read only)
- kOfxImageClipPropIsMask - (read only)
- kOfxImageEffectPropSupportsTiles - (read only)
- kOfxImageEffectPropPixelDepth - (read only)
- kOfxImageEffectPropComponents - (read only)
- kOfxImageClipPropUnmappedPixelDepth - (read only)
- kOfxImageClipPropUnmappedComponents - (read only)
- kOfxImageEffectPropPreMultiplication - (read only)
- kOfxImagePropPixelAspectRatio - (read only)
- kOfxImageEffectPropFrameRate - (read only)
- kOfxImageEffectPropFrameRange - (read only)
- kOfxImageClipPropFieldOrder - (read only)
- kOfxImageClipPropConnected - (read only)
- kOfxImageEffectPropUnmappedFrameRange - (read only)*
- kOfxImageEffectPropUnmappedFrameRate - (read only)*
- kOfxImageClipPropContinuousSamples - (read only)

1.19.6 Properties on an Image

All images are instances, there is no such thing as an image descriptor.

- kOfxPropType - (read only)
- kOfxImageEffectPropPixelDepth - (read only)
- kOfxImageEffectPropComponents - (read only)
- kOfxImageEffectPropPreMultiplication - (read only)
- kOfxImageEffectPropRenderScale - (read only)
- kOfxImagePropPixelAspectRatio - (read only)
- kOfxImagePropData - (read only)
- kOfxImagePropBounds - (read only)
- kOfxImagePropRegionOfDefinition - (read only) *
- kOfxImagePropRowBytes - (read only)
- kOfxImagePropField - (read only)
- kOfxImagePropUniqueIdentifier - (read only)

1.19.7 Properties on Parameter Set Instances

kOfxPropParamSetNeedsSyncing , which indicates if private data is dirty and may need re-syncing to a parameter set .. ParameterProperties:

1.19.8 Properties on Parameter Descriptors and Instances

1.19.9 Properties Common to All Parameters

The following properties are common to all parameters. . . .

- kOfxPropType , which will always be kOfxTypeParameter (read only)
- kOfxPropName read/write in the descriptor, read only on an instance
- kOfxPropLabel read/write in the descriptor and instance
- kOfxPropShortLabel read/write in the descriptor and instance
- kOfxPropLongLabel read/write in the descriptor and instance
- kOfxParamPropType read only in the descriptor and instance, the value is set on construction
- kOfxParamPropSecret read/write in the descriptor and instance
- kOfxParamPropHint read/write in the descriptor and instance
- kOfxParamPropScriptName read/write in the descriptor, read only on an instance
- kOfxParamPropParent read/write in the descriptor, read only on an instance
- kOfxParamPropEnabled read/write in the descriptor and instance
- kOfxParamPropDataPtr read/write in the descriptor and instance
- kOfxPropIcon , read/write on a descriptor, read only on an instance

1.19.10 Properties On Group Parameters

- kOfxParamPropGroupOpen read/write in the descriptor, read only on an instance

1.19.11 Properties Common to All But Group and Page Parameters

- kOfxParamPropInteractV1 read/write in the descriptor, read only on an instance
- kOfxParamPropInteractSize read/write in the descriptor, read only on an instance
- kOfxParamPropInteractSizeAspect read/write in the descriptor, read only on an instance
- kOfxParamPropInteractMinimumSize read/write in the descriptor, read only on an instance
- kOfxParamPropInteractPreferedSize read/write in the descriptor, read only on an instance
- kOfxParamPropHasHostOverlayHandle read only in the descriptor and instance
- kOfxParamPropUseHostOverlayHandle read/write in the descriptor and read only in the instance

1.19.12 Properties Common to All Parameters That Hold Values

- kOfxParamPropDefault read/write in the descriptor, read only on an instance
- kOfxParamPropAnimates read/write in the descriptor, read only on an instance
- kOfxParamPropIsAnimating read/write in the descriptor, read only on an instance
- kOfxParamPropIsAutoKeying read/write in the descriptor, read only on an instance
- kOfxParamPropPersistant read/write in the descriptor, read only on an instance
- kOfxParamPropEvaluateOnChange read/write in the descriptor and instance
- kOfxParamPropPluginMayWrite read/write in the descriptor, read only on an instance
- kOfxParamPropCacheInvalidation read/write in the descriptor, read only on an instance
- kOfxParamPropCanUndo read/write in the descriptor, read only on an instance

1.19.13 Properties Common to All Numeric Parameters

- kOfxParamPropMin read/write in the descriptor and instance
- kOfxParamPropMax read/write in the descriptor and instance
- kOfxParamPropDisplayMin read/write in the descriptor and instance
- kOfxParamPropDisplayMax read/write in the descriptor and instance

1.19.14 Properties Common to All Double Parameters

- kOfxParamPropIncrement read/write in the descriptor and instance
- kOfxParamPropDigits read/write in the descriptor and instance

1.19.15 Properties On 1D Double Parameters

- kOfxParamPropShowTimeMarker read/write in the descriptor and instance
- kOfxParamPropDoubleType read/write in the descriptor, read only on an instance

1.19.16 Properties On 2D and 3D Double Parameters

- kOfxParamPropDoubleType read/write in the descriptor, read only on an instance

1.19.17 Properties On Non Normalised Spatial Double Parameters

- kOfxParamPropDefaultCoordinateSystem read/write in the descriptor, read only on an instance

1.19.18 Properties On 2D and 3D Integer Parameters

- kOfxParamPropDimensionLabel read/write in the descriptor, read only on an instance

1.19.19 Properties On String Parameters

- kOfxParamPropStringMode read/write in the descriptor, read only on an instance
- kOfxParamPropStringFilePathExists read/write in the descriptor, read only on an instance

1.19.20 Properties On Choice Parameters

- kOfxParamPropChoiceOption read/write in the descriptor and instance

1.19.21 Properties On Custom Parameters

- kOfxParamPropCustomInterpCallbackV1 read/write in the descriptor, read only on an instance

1.19.22 Properties On Page Parameters

- kOfxParamPropPageChild read/write in the descriptor, read only on an instance

1.19.23 On Parametric Parameters

- kOfxParamPropAnimates read/write in the descriptor, read only on an instance
- kOfxParamPropIsAnimating read/write in the descriptor, read only on an instance
- kOfxParamPropIsAutoKeying read/write in the descriptor, read only on an instance
- kOfxParamPropPersistant read/write in the descriptor, read only on an instance
- kOfxParamPropEvaluateOnChange read/write in the descriptor and instance
- kOfxParamPropPluginMayWrite read/write in the descriptor, read only on an instance
- kOfxParamPropCacheInvalidation read/write in the descriptor, read only on an instance
- kOfxParamPropCanUndo read/write in the descriptor, read only on an instance
- kOfxParamPropParametricDimension read/write in the descriptor, read only on an instance
- kOfxParamPropParametricUIColour read/write in the descriptor, read only on an instance
- kOfxParamPropParametricInteractBackground read/write in the descriptor, read only on an instance
- kOfxParamPropParametricRange read/write in the descriptor, read only on an instance

1.19.24 Properties on Interact Descriptors

- kOfxInteractPropHasAlpha read only
- kOfxInteractPropBitDepth read only

1.19.25 Properties on Interact Instances

- kOfxPropEffectInstance read only
- kOfxPropInstanceData read/write only
- kOfxInteractPropPixelScale read only
- kOfxInteractPropBackgroundColour read only
- kOfxInteractPropHasAlpha read only
- kOfxInteractPropBitDepth read only
- kOfxInteractPropSlaveToParam read/write
- kOfxInteractPropSuggestedColour read only

1.20 Properties Reference

`kOfxImageEffectFrameVarying`

Indicates whether an effect will generate different images from frame to frame.

- Type - int X 1
- Property Set - out argument to *kOfxImageEffectActionGetClipPreferences* action (read/write).

- Default - 0
- Valid Values - This must be one of 0 or 1

This property indicates whether a plugin will generate a different image from frame to frame, even if no parameters or input image changes. For example a generater that creates random noise pixel at each frame.

`kOfxImageEffectPluginRenderThreadSafety`

Indicates how many simultaneous renders the plugin can deal with.

- Type - string X 1
- Property Set - plugin descriptor (read/write)
- Default - *kOfxImageEffectRenderInstanceSafe*
- Valid Values - This must be one of
  - *kOfxImageEffectRenderUnsafe* - indicating that only a single 'render' call can be made at any time amoung all instances,
  - *kOfxImageEffectRenderInstanceSafe* - indicating that any instance can have a single 'render' call at any one time,
  - *kOfxImageEffectRenderFullySafe* - indicating that any instance of a plugin can have multiple renders running simultaneously

`kOfxPropAPIVersion`

Property on the host descriptor, saying what API version of the API is being implemented.

- Type - int X N
- Property Set - host descriptor.

This is a version string that will specify which version of the API is being implemented by a host. It can have multiple values. For example "1.0", "1.2.4" etc…..

If this is not present, it is safe to assume that the version of the API is "1.0".

`kOfxPropTime`

General property used to get/set the time of something.

- Type - double X 1
- Default - 0, if a setable property
- Property Set - commonly used as an argument to actions, input and output.

`kOfxPropIsInteractive`

Indicates if a host is actively editing the effect with some GUI.

- Type - int X 1
- Property Set - effect instance (read only)
- Valid Values - 0 or 1

If false the effect currently has no interface, however this may be because the effect is loaded in a background render host, or it may be loaded on an interactive host that has not yet opened an editor for the effect.

The output of an effect should only ever depend on the state of its parameters, not on the interactive flag. The interactive flag is more a courtesy flag to let a plugin know that it has an interace. If a plugin want's to have its behaviour dependant on the interactive flag, it can always make a secret parameter which shadows the state if the flag.

`kOfxPluginPropFilePath`

The file path to the plugin.

- Type - C string X 1
- Property Set - effect descriptor (read only)

This is a string that indicates the file path where the plug-in was found by the host. The path is in the native path format for the host OS (eg: UNIX directory separators are forward slashes, Windows ones are backslashes).

The path is to the bundle location, see InstallationLocation. eg: '/usr/OFX/Plugins/AcmePlugins/AcmeFantasticPlugin.ofx.bundle'

`kOfxPropInstanceData`

A private data pointer that the plug-in can store its own data behind.

- Type - pointer X 1
- Property Set - plugin instance (read/write),
- Default - NULL

This data pointer is unique to each plug-in instance, so two instances of the same plug-in do not share the same data pointer. Use it to hang any needed private data structures.

`kOfxPropType`

General property, used to identify the kind of an object behind a handle.

- Type - ASCII C string X 1
- Property Set - any object handle (read only)
- Valid Values - currently this can be…
  - kOfxTypeImageEffectHost
  - kOfxTypeImageEffect
  - kOfxTypeImageEffectInstance
  - kOfxTypeParameter
  - kOfxTypeParameterInstance
  - kOfxTypeClip
  - kOfxTypeImage

`kOfxPropName`

Unique name of an object.

- Type - ASCII C string X 1
- Property Set - on many objects (descriptors and instances), see PropertiesByObject (read only)

This property is used to label objects uniquely amoung objects of that type. It is typically set when a plugin creates a new object with a function that takes a name.

`kOfxPropVersion`

Identifies a specific version of a host or plugin.

- Type - int X N
- Property Set - host descriptor (read only), plugin descriptor (read/write)
- Default - “0”
- Valid Values - positive integers

This is a multi dimensional integer property that represents the version of a host (host descriptor), or plugin (plugin descriptor). These represent a version number of the form ‘1.2.3.4’, with each dimension adding another ‘dot’ on the right.

A version is considered to be more recent than another if its ordered set of values is lexicographically greater than another, reading left to right. (ie: 1.2.4 is smaller than 1.2.6). Also, if the number of dimensions is different, then the values of the missing dimensions are considered to be zero (so 1.2.4 is greater than 1.2).

`kOfxPropVersionLabel`

Unique user readable version string of a plugin or host.

- Type - string X 1
- Property Set - host descriptor (read only), plugin descriptor (read/write)
- Default - none, the host needs to set this
- Valid Values - ASCII string

This is purely for user feedback, a plugin or host should use *kOfxPropVersion* if they need to check for specific versions.

`kOfxPropPluginDescription`

Description of the plug-in to a user.

- Type - string X 1
- Property Set - plugin descriptor (read/write) and instance (read only)
- Default - “”
- Valid Values - UTF8 string

This is a string giving a potentially verbose description of the effect.

`kOfxPropLabel`

User visible name of an object.

- Type - UTF8 C string X 1

- Property Set - on many objects (descriptors and instances), see PropertiesByObject. Typically readable and writable in most cases.
- Default - the *kOfxPropName* the object was created with.

The label is what a user sees on any interface in place of the object's name.

Note that resetting this will also reset *kOfxPropShortLabel* and *kOfxPropLongLabel*.

`kOfxPropIcon`

If set this tells the host to use an icon instead of a label for some object in the interface.

- Type - string X 2
- Property Set - various descriptors in the API
- Default - ""
- Valid Values - ASCII string

The value is a path is defined relative to the Resource folder that points to an SVG or PNG file containing the icon.

The first dimension, if set, will the name of and SVG file, the second a PNG file.

`kOfxPropShortLabel`

Short user visible name of an object.

- Type - UTF8 C string X 1
- Property Set - on many objects (descriptors and instances), see PropertiesByObject. Typically readable and writable in most cases.
- Default - initially *kOfxPropName*, but will be reset if *kOfxPropLabel* is changed.

This is a shorter version of the label, typically 13 character glyphs or less. Hosts should use this if they have limitted display space for their object labels.

`kOfxPropLongLabel`

Long user visible name of an object.

- Type - UTF8 C string X 1
- Property Set - on many objects (descriptors and instances), see PropertiesByObject. Typically readable and writable in most cases.
- Default - initially *kOfxPropName*, but will be reset if *kOfxPropLabel* is changed.

This is a longer version of the label, typically 32 character glyphs or so. Hosts should use this if they have mucg display space for their object labels.

`kOfxPropChangeReason`

Indicates why a plug-in changed.

- Type - ASCII C string X 1
- Property Set - inArgs parameter on the *kOfxActionInstanceChanged* action.

- Valid Values - this can be…
  - *kOfxChangeUserEdited* - the user directly edited the instance somehow and caused a change to something, this includes undo/redos and resets
  - *kOfxChangePluginEdited* - the plug-in itself has changed the value of the object in some action
  - *kOfxChangeTime* - the time has changed and this has affected the value of the object because it varies over time

Argument property for the *kOfxActionInstanceChanged* action.

`kOfxPropEffectInstance`

A pointer to an effect instance.

- Type - pointer X 1
- Property Set - on an interact instance (read only)

This property is used to link an object to the effect. For example if the plug-in supplies an openGL overlay for an image effect, the interact instance will have one of these so that the plug-in can connect back to the effect the GUI links to.

`kOfxPropHostOSHandle`

A pointer to an operating system specific application handle.

- Type - pointer X 1
- Property Set - host descriptor.

Some plug-in vendor want raw OS specific handles back from the host so they can do interesting things with host OS APIs. Typically this is to control windowing properly on Microsoft Windows. This property returns the appropriate 'root' window handle on the current operating system. So on Windows this would be the hWnd of the application main window.

`kOfxInteractPropDrawContext`

The Draw Context handle.

- Type - pointer X 1
- Property Set - read only property on the inArgs of the following actions…
- *kOfxInteractActionDraw*

`kOfxImageEffectPropSupportedContexts`

Indicates to the host the contexts a plugin can be used in.

- Type - string X N
- Property Set - image effect descriptor passed to kOfxActionDescribe (read/write)
- Default - this has no defaults, it must be set
- Valid Values - This must be one of
  - kOfxImageEffectContextGenerator

  - kOfxImageEffectContextFilter
  - kOfxImageEffectContextTransition
  - kOfxImageEffectContextPaint
  - kOfxImageEffectContextGeneral
  - kOfxImageEffectContextRetimer

`kOfxImageEffectPropPluginHandle`

The plugin handle passed to the initial 'describe' action.

- Type - pointer X 1
- Property Set - plugin instance, (read only)

This value will be the same for all instances of a plugin.

`kOfxImageEffectHostPropIsBackground`

Indicates if a host is a background render.

- Type - int X 1
- Property Set - host descriptor (read only)
- Valid Values - This must be one of
  - 0 if the host is a foreground host, it may open the effect in an interactive session (or not)
  - 1 if the host is a background 'processing only' host, and the effect will never be opened in an interactive session.

`kOfxImageEffectPluginPropSingleInstance`

Indicates whether only one instance of a plugin can exist at the same time.

- Type - int X 1
- Property Set - plugin descriptor (read/write)
- Default - 0
- Valid Values - This must be one of
  - 0 - which means multiple instances can exist simultaneously,
  - 1 - which means only one instance can exist at any one time.

Some plugins, for whatever reason, may only be able to have a single instance in existance at any one time. This plugin property is used to indicate that.

`kOfxImageEffectPluginPropHostFrameThreading`

Indicates whether a plugin lets the host perform per frame SMP threading.

- Type - int X 1
- Property Set - plugin descriptor (read/write)

- Default - 1
- Valid Values - This must be one of
  - 0 - which means that the plugin will perform any per frame SMP threading
  - 1 - which means the host can call an instance's render function simultaneously at the same frame, but with different windows to render.

`kOfxImageEffectPropSupportsMultipleClipDepths`

Indicates whether a host or plugin can support clips of differing component depths going into/out of an effect.

- Type - int X 1
- Property Set - plugin descriptor (read/write), host descriptor (read only)
- Default - 0 for a plugin
- Valid Values - This must be one of
  - 0 - in which case the host or plugin does not support clips of multiple pixel depths,
  - 1 - which means a host or plugin is able to to deal with clips of multiple pixel depths,

If a host indicates that it can support multiple pixels depths, then it will allow the plugin to explicitly set the output clip's pixel depth in the *kOfxImageEffectActionGetClipPreferences* action. See ImageEffectClipPreferences.

`kOfxImageEffectPropSupportsMultipleClipPARs`

Indicates whether a host or plugin can support clips of differing pixel aspect ratios going into/out of an effect.

- Type - int X 1
- Property Set - plugin descriptor (read/write), host descriptor (read only)
- Default - 0 for a plugin
- Valid Values - This must be one of
  - 0 - in which case the host or plugin does not support clips of multiple pixel aspect ratios
  - 1 - which means a host or plugin is able to to deal with clips of multiple pixel aspect ratios

If a host indicates that it can support multiple pixel aspect ratios, then it will allow the plugin to explicitly set the output clip's aspect ratio in the *kOfxImageEffectActionGetClipPreferences* action. See ImageEffectClipPreferences.

`kOfxImageEffectPropClipPreferencesSlaveParam`

Indicates the set of parameters on which a value change will trigger a change to clip preferences.

- Type - string X N
- Property Set - plugin descriptor (read/write)
- Default - none set
- Valid Values - the name of any described parameter

The plugin uses this to inform the host of the subset of parameters that affect the effect's clip preferences. A value change in any one of these will trigger a call to the clip preferences action.

The plugin can be slaved to multiple parameters (setting index 0, then index 1 etc…)

`kOfxImageEffectPropSetableFrameRate`

Indicates whether the host will let a plugin set the frame rate of the output clip.

- Type - int X 1
- Property Set - host descriptor (read only)
- Valid Values - This must be one of
  - 0 - in which case the plugin may not change the frame rate of the output clip,
  - 1 - which means a plugin is able to change the output clip's frame rate in the *kOfxImageEffectActionGetClipPreferences* action.

See ImageEffectClipPreferences.

If a clip can be continosly sampled, the frame rate will be set to 0.

`kOfxImageEffectPropSetableFielding`

Indicates whether the host will let a plugin set the fielding of the output clip.

- Type - int X 1
- Property Set - host descriptor (read only)
- Valid Values - This must be one of
  - 0 - in which case the plugin may not change the fielding of the output clip,
  - 1 - which means a plugin is able to change the output clip's fielding in the *kOfxImageEffectActionGetClipPreferences* action.

See ImageEffectClipPreferences.

`kOfxImageEffectInstancePropSequentialRender`

Indicates whether a plugin needs sequential rendering, and a host support it.

- Type - int X 1
- Property Set - plugin descriptor (read/write) or plugin instance (read/write), and host descriptor (read only)
- Default - 0
- Valid Values -
  - 0 - for a plugin, indicates that a plugin does not need to be sequentially rendered to be correct, for a host, indicates that it cannot ever guarantee sequential rendering,
  - 1 - for a plugin, indicates that it needs to be sequentially rendered to be correct, for a host, indicates that it can always support sequential rendering of plugins that are sequentially rendered,
  - 2 - for a plugin, indicates that it is best to render sequentially, but will still produce correct results if not, for a host, indicates that it can sometimes render sequentially, and will have set *kOfxImageEffectPropSequentialRenderStatus* on the relevant actions

Some effects have temporal dependancies, some information from from the rendering of frame N-1 is needed to render frame N correctly. This property is set by an effect to indicate such a situation. Also, some effects are more efficient if they run sequentially, but can still render correct images even if they do not, eg: a complex particle system.

During an interactive session a host may attempt to render a frame out of sequence (for example when the user scrubs the current time), and the effect needs to deal with such a situation as best it can to provide feedback to the user.

However if a host caches output, any frame frame generated in random temporal order needs to be considered invalid and needs to be re-rendered when the host finally performs a first to last render of the output sequence.

In all cases, a host will set the kOfxImageEffectPropSequentialRenderStatus flag to indicate its sequential render status.

`kOfxImageEffectPropSequentialRenderStatus`

Property on all the render action that indicate the current sequential render status of a host.

- Type - int X 1
- Property Set - read only property on the inArgs of the following actions…
  - *kOfxImageEffectActionBeginSequenceRender*
  - *kOfxImageEffectActionRender*
  - *kOfxImageEffectActionEndSequenceRender*
- Valid Values -
  - 0 - the host is not currently sequentially rendering,
  - 1 - the host is currentely rendering in a way so that it guarantees sequential rendering.

This property is set to indicate whether the effect is currently being rendered in frame order on a single effect instance. See *kOfxImageEffectInstancePropSequentialRender* for more details on sequential rendering.

`kOfxImageEffectPropInteractiveRenderStatus`

Property that indicates if a plugin is being rendered in response to user interaction.

- Type - int X 1
- Property Set - read only property on the inArgs of the following actions…
  - *kOfxImageEffectActionBeginSequenceRender*
  - *kOfxImageEffectActionRender*
  - *kOfxImageEffectActionEndSequenceRender*
- Valid Values -
  - 0 - the host is rendering the instance due to some reason other than an interactive tweak on a UI,
  - 1 - the instance is being rendered because a user is modifying parameters in an interactive session.

This property is set to 1 on all render calls that have been triggered because a user is actively modifying an effect (or up stream effect) in an interactive session. This typically means that the effect is not being rendered as a part of a sequence, but as a single frame.

`kOfxImageEffectPluginPropGrouping`

Indicates the effect group for this plugin.

- Type - UTF8 string X 1
- Property Set - plugin descriptor (read/write)
- Default - ""

This is purely a user interface hint for the host so it can group related effects on any menus it may have.

`kOfxImageEffectPropSupportsOverlays`

Indicates whether a host support image effect ImageEffectOverlays.

- Type - int X 1
- Property Set - host descriptor (read only)
- Valid Values - This must be one of
  - 0 - the host won't allow a plugin to draw a GUI over the output image,
  - 1 - the host will allow a plugin to draw a GUI over the output image.

`kOfxImageEffectPluginPropOverlayInteractV1`

Sets the entry for an effect's overlay interaction.

- Type - pointer X 1
- Property Set - plugin descriptor (read/write)
- Default - NULL
- Valid Values - must point to an *OfxPluginEntryPoint*

The entry point pointed to must be one that handles custom interaction actions.

`kOfxImageEffectPropSupportsMultiResolution`

Indicates whether a plugin or host support multiple resolution images.

- Type - int X 1
- Property Set - host descriptor (read only), plugin descriptor (read/write)
- Default - 1 for plugins
- Valid Values - This must be one of
  - 0 - the plugin or host does not support multiple resolutions
  - 1 - the plugin or host does support multiple resolutions

Multiple resolution images mean…

- input and output images can be of any size
- input and output images can be offset from the origin

`kOfxImageEffectPropSupportsTiles`

Indicates whether a clip, plugin or host supports tiled images.

- Type - int X 1
- Property Set - host descriptor (read only), plugin descriptor (read/write), clip descriptor (read/write), instance (read/write)
- Default - to 1 for a plugin and clip
- Valid Values - This must be one of 0 or 1

Tiled images mean that input or output images can contain pixel data that is only a subset of their full RoD.

If a clip or plugin does not support tiled images, then the host should supply full RoD images to the effect whenever it fetches one.

V1.4: It is now possible (defined) to change OfxImageEffectPropSupportsTiles in Instance Changed

`kOfxImageEffectPropTemporalClipAccess`

Indicates support for random temporal access to images in a clip.

- Type - int X 1
- Property Set - host descriptor (read only), plugin descriptor (read/write), clip descriptor (read/write)
- Default - to 0 for a plugin and clip
- Valid Values - This must be one of 0 or 1

On a host, it indicates whether the host supports temporal access to images.

On a plugin, indicates if the plugin needs temporal access to images.

On a clip, it indicates that the clip needs temporal access to images.

`kOfxImageEffectPropContext`

Indicates the context a plugin instance has been created for.

- Type - string X 1
- Property Set - image effect instance (read only)
- Valid Values - This must be one of
  - kOfxImageEffectContextGenerator
  - kOfxImageEffectContextFilter
  - kOfxImageEffectContextTransition
  - kOfxImageEffectContextPaint
  - kOfxImageEffectContextGeneral
  - kOfxImageEffectContextRetimer

`kOfxImageEffectPropPixelDepth`

Indicates the type of each component in a clip or image (after any mapping)

- Type - string X 1
- Property Set - clip instance (read only), image instance (read only)
- Valid Values - This must be one of
  - kOfxBitDepthNone (implying a clip is unconnected, not valid for an image)
  - kOfxBitDepthByte
  - kOfxBitDepthShort
  - kOfxBitDepthHalf
  - kOfxBitDepthFloat

Note that for a clip, this is the value set by the clip preferences action, not the raw 'actual' value of the clip.

`kOfxImageEffectPropComponents`

Indicates the current component type in a clip or image (after any mapping)

- Type - string X 1
- Property Set - clip instance (read only), image instance (read only)
- Valid Values - This must be one of
  - kOfxImageComponentNone (implying a clip is unconnected, not valid for an image)
  - kOfxImageComponentRGBA
  - kOfxImageComponentRGB
  - kOfxImageComponentAlpha

Note that for a clip, this is the value set by the clip preferences action, not the raw 'actual' value of the clip.

`kOfxImagePropUniqueIdentifier`

Uniquely labels an image.

- Type - ASCII string X 1
- Property Set - image instance (read only)

This is host set and allows a plug-in to differentiate between images. This is especially useful if a plugin caches analysed information about the image (for example motion vectors). The plugin can label the cached information with this identifier. If a user connects a different clip to the analysed input, or the image has changed in some way then the plugin can detect this via an identifier change and re-evaluate the cached information.

`kOfxImageClipPropContinuousSamples`

Clip and action argument property which indicates that the clip can be sampled continously.

- Type - int X 1
- Property Set - clip instance (read only), as an out argument to *kOfxImageEffectActionGetClipPreferences* action (read/write)
- Default - 0 as an out argument to the *kOfxImageEffectActionGetClipPreferences* action

- Valid Values - This must be one of…
  - 0 if the images can only be sampled at discreet times (eg: the clip is a sequence of frames),
  - 1 if the images can only be sampled continuously (eg: the clip is infact an animating roto spline and can be rendered anywhen).

If this is set to true, then the frame rate of a clip is effectively infinite, so to stop arithmetic errors the frame rate should then be set to 0.

`kOfxImageClipPropUnmappedPixelDepth`

Indicates the type of each component in a clip before any mapping by clip preferences.

- Type - string X 1
- Property Set - clip instance (read only)
- Valid Values - This must be one of
  - kOfxBitDepthNone (implying a clip is unconnected image)
  - kOfxBitDepthByte
  - kOfxBitDepthShort
  - kOfxBitDepthHalf
  - kOfxBitDepthFloat

This is the actual value of the component depth, before any mapping by clip preferences.

`kOfxImageClipPropUnmappedComponents`

Indicates the current 'raw' component type on a clip before any mapping by clip preferences.

- Type - string X 1
- Property Set - clip instance (read only),
- Valid Values - This must be one of
  - kOfxImageComponentNone (implying a clip is unconnected)
  - kOfxImageComponentRGBA
  - kOfxImageComponentRGB
  - kOfxImageComponentAlpha

`kOfxImageEffectPropPreMultiplication`

Indicates the premultiplication state of a clip or image.

- Type - string X 1
- Property Set - clip instance (read only), image instance (read only), out args property in the *kOfxImageEffectActionGetClipPreferences* action (read/write)
- Valid Values - This must be one of
  - kOfxImageOpaque - the image is opaque and so has no premultiplication state
  - kOfxImagePreMultiplied - the image is premultiplied by its alpha

- kOfxImageUnPreMultiplied - the image is unpremultiplied

See the documentation on clip preferences for more details on how this is used with the *kOfxImageEffectActionGetClipPreferences* action.

`kOfxImageEffectPropSupportedPixelDepths`

Indicates the bit depths support by a plug-in or host.

- Type - string X N
- Property Set - host descriptor (read only), plugin descriptor (read/write)
- Default - plugin descriptor none set
- Valid Values - This must be one of
  - kOfxBitDepthNone (implying a clip is unconnected, not valid for an image)
  - kOfxBitDepthByte
  - kOfxBitDepthShort
  - kOfxBitDepthHalf
  - kOfxBitDepthFloat

The default for a plugin is to have none set, the plugin *must* define at least one in its describe action.

`kOfxImageEffectPropSupportedComponents`

Indicates the components supported by a clip or host,.

- Type - string X N
- Property Set - host descriptor (read only), clip descriptor (read/write)
- Valid Values - This must be one of
  - kOfxImageComponentNone (implying a clip is unconnected)
  - kOfxImageComponentRGBA
  - kOfxImageComponentRGB
  - kOfxImageComponentAlpha

This list of strings indicate what component types are supported by a host or are expected as input to a clip.

The default for a clip descriptor is to have none set, the plugin *must* define at least one in its define function

`kOfxImageClipPropOptional`

Indicates if a clip is optional.

- Type - int X 1
- Property Set - clip descriptor (read/write)
- Default - 0
- Valid Values - This must be one of 0 or 1

`kOfxImageClipPropIsMask`

Indicates that a clip is intended to be used as a mask input.

- Type - int X 1
- Property Set - clip descriptor (read/write)
- Default - 0
- Valid Values - This must be one of 0 or 1

Set this property on any clip which will only ever have single channel alpha images fetched from it. Typically on an optional clip such as a junk matte in a keyer.

This property acts as a hint to hosts indicating that they could feed the effect from a rotoshape (or similar) rather than an 'ordinary' clip.

`kOfxImagePropPixelAspectRatio`

The pixel aspect ratio of a clip or image.

- Type - double X 1
- Property Set - clip instance (read only), image instance (read only) and *kOfxImageEffectActionGetClipPreferences* action out args property (read/write)

`kOfxImageEffectPropFrameRate`

The frame rate of a clip or instance's project.

- Type - double X 1
- Property Set - clip instance (read only), effect instance (read only) and *kOfxImageEffectActionGetClipPreferences* action out args property (read/write)

For an input clip this is the frame rate of the clip.

For an output clip, the frame rate mapped via pixel preferences.

For an instance, this is the frame rate of the project the effect is in.

For the outargs property in the *kOfxImageEffectActionGetClipPreferences* action, it is used to change the frame rate of the ouput clip.

`kOfxImageEffectPropUnmappedFrameRate`

Indicates the original unmapped frame rate (frames/second) of a clip.

- Type - double X 1
- Property Set - clip instance (read only),

If a plugin changes the output frame rate in the pixel preferences action, this property allows a plugin to get to the original value.

`kOfxImageEffectPropFrameStep`

The frame step used for a sequence of renders.

- Type - double X 1
- Property Set - an in argument for the *kOfxImageEffectActionBeginSequenceRender* action (read only)
- Valid Values - can be any positive value, but typically
  - 1 for frame based material
  - 0.5 for field based material

`kOfxImageEffectPropFrameRange`

The frame range over which a clip has images.

- Type - double X 2
- Property Set - clip instance (read only)

Dimension 0 is the first frame for which the clip can produce valid data.

Dimension 1 is the last frame for which the clip can produce valid data.

`kOfxImageEffectPropUnmappedFrameRange`

The unmaped frame range over which an output clip has images.

- Type - double X 2
- Property Set - clip instance (read only)

Dimension 0 is the first frame for which the clip can produce valid data.

Dimension 1 is the last frame for which the clip can produce valid data.

If a plugin changes the output frame rate in the pixel preferences action, it will affect the frame range of the output clip, this property allows a plugin to get to the original value.

`kOfxImageClipPropConnected`

Says whether the clip is actually connected at the moment.

- Type - int X 1
- Property Set - clip instance (read only)
- Valid Values - This must be one of 0 or 1

An instance may have a clip may not be connected to an object that can produce image data. Use this to find out.

Any clip that is not optional will *always* be connected during a render action. However, during interface actions, even non optional clips may be unconnected.

`kOfxImageEffectPropRenderScale`

The proxy render scale currently being applied.

- Type - double X 2
- Property Set - an image instance (read only) and as read only an in argument on the following actions,

   - *kOfxImageEffectActionRender*
   - *kOfxImageEffectActionBeginSequenceRender*
   - *kOfxImageEffectActionEndSequenceRender*
   - *kOfxImageEffectActionIsIdentity*
   - *kOfxImageEffectActionGetRegionOfDefinition*
   - *kOfxImageEffectActionGetRegionsOfInterest*
   - *kOfxActionInstanceChanged*
   - *kOfxInteractActionDraw*
   - *kOfxInteractActionPenMotion*
   - *kOfxInteractActionPenDown*
   - *kOfxInteractActionPenUp*
   - *kOfxInteractActionKeyDown*
   - *kOfxInteractActionKeyUp*
   - *kOfxInteractActionKeyRepeat*
   - *kOfxInteractActionGainFocus*
   - *kOfxInteractActionLoseFocus*

This should be applied to any spatial parameters to position them correctly. Not that the 'x' value does not include any pixel aspect ratios.

`kOfxImageEffectPropRenderQualityDraft`

Indicates whether an effect can take quality shortcuts to improve speed.

- Type - int X 1
- Property Set - render calls, host (read-only)
- Default - 0 - 0: Best Quality (1: Draft)
- Valid Values - This must be one of 0 or 1

This property indicates that the host provides the plug-in the option to render in Draft/Preview mode. This is useful for applications that must support fast scrubbing. These allow a plug-in to take short-cuts for improved performance when the situation allows and it makes sense, for example to generate thumbnails with effects applied. For example switch to a cheaper interpolation type or rendering mode. A plugin should expect frames rendered in this manner that will not be stucked in host cache unless the cache is only used in the same draft situations. If an host does not support that property a value of 0 is assumed. Also note that some hosts do implement kOfxImageEffectPropRenderScale - these two properties can be used independently.

`kOfxImageEffectPropProjectExtent`

The extent of the current project in canonical coordinates.

- Type - double X 2
- Property Set - a plugin instance (read only)

The extent is the size of the 'output' for the current project. See NormalisedCoordinateSystem for more infomation on the project extent.

The extent is in canonical coordinates and only returns the top right position, as the extent is always rooted at 0,0.

For example a PAL SD project would have an extent of 768, 576.

`kOfxImageEffectPropProjectSize`

The size of the current project in canonical coordinates.

- Type - double X 2
- Property Set - a plugin instance (read only)

The size of a project is a sub set of the *kOfxImageEffectPropProjectExtent*. For example a project may be a PAL SD project, but only be a letter-box within that. The project size is the size of this sub window.

The project size is in canonical coordinates.

See NormalisedCoordinateSystem for more infomation on the project extent.

`kOfxImageEffectPropProjectOffset`

The offset of the current project in canonical coordinates.

- Type - double X 2
- Property Set - a plugin instance (read only)

The offset is related to the *kOfxImageEffectPropProjectSize* and is the offset from the origin of the project 'sub-window'.

For example for a PAL SD project that is in letterbox form, the project offset is the offset to the bottom left hand corner of the letter box.

The project offset is in canonical coordinates.

See NormalisedCoordinateSystem for more infomation on the project extent.

`kOfxImageEffectPropProjectPixelAspectRatio`

The pixel aspect ratio of the current project.

- Type - double X 1
- Property Set - a plugin instance (read only)

`kOfxImageEffectInstancePropEffectDuration`

The duration of the effect.

- Type - double X 1
- Property Set - a plugin instance (read only)

This contains the duration of the plug-in effect, in frames.

`kOfxImageClipPropFieldOrder`

Which spatial field occurs temporally first in a frame.

- Type - string X 1
- Property Set - a clip instance (read only)
- Valid Values - This must be one of
  - kOfxImageFieldNone - the material is unfielded
  - kOfxImageFieldLower - the material is fielded, with image rows 0,2,4.... occuring first in a frame
  - kOfxImageFieldUpper - the material is fielded, with image rows line 1,3,5.... occuring first in a frame

`kOfxImagePropData`

The pixel data pointer of an image.

- Type - pointer X 1
- Property Set - an image instance (read only)

This property contains a pointer to memory that is the lower left hand corner of an image.

`kOfxImagePropBounds`

The bounds of an image's pixels.

- Type - integer X 4
- Property Set - an image instance (read only)

The bounds, in PixelCoordinates, are of the addressable pixels in an image's data pointer.

The order of the values is x1, y1, x2, y2.

X values are x1 <= X < x2 Y values are y1 <= Y < y2

For less than full frame images, the pixel bounds will be contained by the *kOfxImagePropRegionOfDefinition* bounds.

`kOfxImagePropRegionOfDefinition`

The full region of definition of an image.

- Type - integer X 4
- Property Set - an image instance (read only)

An image's region of definition, in PixelCoordinates, is the full frame area of the image plane that the image covers.

The order of the values is x1, y1, x2, y2.

X values are x1 <= X < x2 Y values are y1 <= Y < y2

The *kOfxImagePropBounds* property contains the actuall addressable pixels in an image, which may be less than its full region of definition.

`kOfxImagePropRowBytes`

The number of bytes in a row of an image.

- Type - integer X 1
- Property Set - an image instance (read only)

For various alignment reasons, a row of pixels may need to be padded at the end with several bytes before the next row starts in memory.

This property indicates the number of bytes in a row of pixels. This will be at least sizeof(PIXEL) * (bounds.x2-bounds.x1). Where bounds is fetched from the *kOfxImagePropBounds* property.

Note that row bytes can be negative, which allows hosts with a native top down row order to pass image into OFX without having to repack pixels.

`kOfxImagePropField`

Which fields are present in the image.

- Type - string X 1
- Property Set - an image instance (read only)
- Valid Values - This must be one of
  - kOfxImageFieldNone - the image is an unfielded frame
  - *kOfxImageFieldBoth* - the image is fielded and contains both interlaced fields
  - kOfxImageFieldLower - the image is fielded and contains a single field, being the lower field (rows 0,2,4…)
  - kOfxImageFieldUpper - the image is fielded and contains a single field, being the upper field (rows 1,3,5…)

`kOfxImageEffectPluginPropFieldRenderTwiceAlways`

Controls how a plugin renders fielded footage.

- Type - integer X 1
- Property Set - a plugin descriptor (read/write)
- Default - 1
- Valid Values - This must be one of
  - 0 - the plugin is to have its render function called twice, only if there is animation in any of its parameters
  - 1 - the plugin is to have its render function called twice always

`kOfxImageClipPropFieldExtraction`

Controls how a plugin fetched fielded imagery from a clip.

- Type - string X 1
- Property Set - a clip descriptor (read/write)

- Default - kOfxImageFieldDoubled
- Valid Values - This must be one of
  - kOfxImageFieldBoth - fetch a full frame interlaced image
  - kOfxImageFieldSingle - fetch a single field, making a half height image
  - kOfxImageFieldDoubled - fetch a single field, but doubling each line and so making a full height image

This controls how a plug-in wishes to fetch images from a fielded clip, so it can tune it behaviour when it renders fielded footage.

Note that if it fetches kOfxImageFieldSingle and the host stores images natively as both fields interlaced, it can return a single image by doubling rowbytes and tweaking the starting address of the image data. This saves on a buffer copy.

`kOfxImageEffectPropFieldToRender`

Indicates which field is being rendered.

- Type - string X 1
- Property Set - a read only in argument property to *kOfxImageEffectActionRender* and *kOfxImageEffectActionIsIdentity*
- Valid Values - this must be one of
  - kOfxImageFieldNone - there are no fields to deal with, all images are full frame
  - kOfxImageFieldBoth - the imagery is fielded and both scan lines should be renderred
  - kOfxImageFieldLower - the lower field is being rendered (lines 0,2,4…)
  - kOfxImageFieldUpper - the upper field is being rendered (lines 1,3,5…)

`kOfxImageEffectPropRegionOfDefinition`

Used to indicate the region of definition of a plug-in.

- Type - double X 4
- Property Set - a read/write out argument property to the *kOfxImageEffectActionGetRegionOfDefinition* action
- Default - see *kOfxImageEffectActionGetRegionOfDefinition*

The order of the values is x1, y1, x2, y2.

This will be in CanonicalCoordinates

`kOfxImageEffectPropRegionOfInterest`

The value of a region of interest.

- Type - double X 4
- Property Set - a read only in argument property to the *kOfxImageEffectActionGetRegionsOfInterest* action

A host passes this value into the region of interest action to specify the region it is interested in rendering.

The order of the values is x1, y1, x2, y2.

This will be in CanonicalCoordinates.

`kOfxImageEffectPropRenderWindow`

The region to be rendered.

- Type - integer X 4
- Property Set - a read only in argument property to the *kOfxImageEffectActionRender* and *kOfxImageEffectActionIsIdentity* actions

The order of the values is x1, y1, x2, y2.

This will be in PixelCoordinates

`kOfxInteractPropSlaveToParam`

The set of parameters on which a value change will trigger a redraw for an interact.

- Type - string X N
- Property Set - interact instance property (read/write)
- Default - no values set
- Valid Values - the name of any parameter associated with this interact.

If the interact is representing the state of some set of OFX parameters, then is will need to be redrawn if any of those parameters' values change. This multi-dimensional property links such parameters to the interact.

The interact can be slaved to multiple parameters (setting index 0, then index 1 etc…)

`kOfxInteractPropPixelScale`

The size of a real screen pixel under the interact's canonical projection.

- Type - double X 2
- Property Set - interact instance and actions (read only)

`kOfxInteractPropBackgroundColour`

The background colour of the application behind an interact instance.

- Type - double X 3
- Property Set - read only on the interact instance and in argument to the *kOfxInteractActionDraw* action
- Valid Values - from 0 to 1

The components are in the order red, green then blue.

`kOfxInteractPropSuggestedColour`

The suggested colour to draw a widget in an interact, typically for overlays.

- Type - double X 3
- Property Set - read only on the interact instance

- Default - 1.0
- Valid Values - greater than or equal to 0.0

Some applications allow the user to specify colours of any overlay via a colour picker, this property represents the value of that colour. Plugins are at liberty to use this or not when they draw an overlay.

If a host does not support such a colour, it should return kOfxStatReplyDefault

`kOfxInteractPropPenPosition`

The position of the pen in an interact.

- Type - double X 2
- Property Set - read only in argument to the *kOfxInteractActionPenMotion*, *kOfxInteractActionPenDown* and *kOfxInteractActionPenUp* actions

This value passes the postion of the pen into an interact. This is in the interact's canonical coordinates.

`kOfxInteractPropPenViewportPosition`

The position of the pen in an interact in viewport coordinates.

- Type - int X 2
- Property Set - read only in argument to the *kOfxInteractActionPenMotion*, *kOfxInteractActionPenDown* and *kOfxInteractActionPenUp* actions

This value passes the postion of the pen into an interact. This is in the interact's openGL viewport coordinates, with 0,0 being at the bottom left.

`kOfxInteractPropPenPressure`

The pressure of the pen in an interact.

- Type - double X 1
- Property Set - read only in argument to the *kOfxInteractActionPenMotion*, *kOfxInteractActionPenDown* and *kOfxInteractActionPenUp* actions
- Valid Values - from 0 (no pressure) to 1 (maximum pressure)

This is used to indicate the status of the 'pen' in an interact. If a pen has only two states (eg: a mouse button), these should map to 0.0 and 1.0.

`kOfxInteractPropBitDepth`

Indicates whether the dits per component in the interact's openGL frame buffer.

- Type - int X 1
- Property Set - interact instance and descriptor (read only)

`kOfxInteractPropHasAlpha`

Indicates whether the interact's frame buffer has an alpha component or not.

- Type - int X 1
- Property Set - interact instance and descriptor (read only)
- Valid Values - This must be one of
  - 0 indicates no alpha component
  - 1 indicates an alpha component

`kOfxPropKeySym`

Property used to indicate which a key on the keyboard or a button on a button device has been pressed.

- Type - int X 1
- Property Set - an read only in argument for the actions *kOfxInteractActionKeyDown*, *kOfxInteractActionKeyUp* and *kOfxInteractActionKeyRepeat*.
- Valid Values - one of any specified by #defines in the file ofxKeySyms.h.

This property represents a raw key press, it does not represent the 'character value' of the key.

This property is associated with a *kOfxPropKeyString* property, which encodes the UTF8 value for the keypress/button press. Some keys (for example arrow keys) have no UTF8 equivalant.

Some keys, especially on non-english language systems, may have a UTF8 value, but *not* a keysym values, in these cases, the keysym will have a value of kOfxKey_Unknown, but the *kOfxPropKeyString* property will still be set with the UTF8 value.

`kOfxPropKeyString`

This property encodes a single keypresses that generates a unicode code point. The value is stored as a UTF8 string.

- Type - C string X 1, UTF8
- Property Set - an read only in argument for the actions *kOfxInteractActionKeyDown*, *kOfxInteractActionKeyUp* and *kOfxInteractActionKeyRepeat*.
- Valid Values - a UTF8 string representing a single character, or the empty string.

This property represents the UTF8 encode value of a single key press by a user in an OFX interact.

This property is associated with a *kOfxPropKeySym* which represents an integer value for the key press. Some keys (for example arrow keys) have no UTF8 equivalant, in which case this is set to the empty string "", and the associate *kOfxPropKeySym* is set to the equivilant raw key press.

Some keys, especially on non-english language systems, may have a UTF8 value, but *not* a keysym values, in these cases, the keysym will have a value of kOfxKey_Unknown, but the *kOfxPropKeyString* property will still be set with the UTF8 value.

`kOfxImageEffectPropInAnalysis`

Indicates whether an effect is performing an analysis pass. ofxImageEffects.h.

- Type - int X 1
- Property Set - plugin instance (read/write)
- Default - to 0

- Valid Values - This must be one of 0 or 1

Deprecated:

- This feature has been deprecated - officially commented out v1.4.

`kOfxInteractPropViewportSize`

The size of an interact's openGL viewport ofxInteract.h.

- Type - int X 2
- Property Set - read only property on the interact instance and in argument to all the interact actions.

Deprecated:

- V1.3: This property is the redundant and its use will be deprecated in future releases. V1.4: Removed

`kOfxImageEffectPropOpenGLRenderSupported`

Indicates whether a host or plugin can support OpenGL accelerated rendering.

- Type - C string X 1
- Property Set - plugin descriptor (read/write), host descriptor (read only) - plugin instance change (read/write)
- Default - "false" for a plugin
- Valid Values - This must be one of
  - "false" - in which case the host or plugin does not support OpenGL accelerated rendering
  - "true" - which means a host or plugin can support OpenGL accelerated rendering, in the case of plug-ins this also means that it is capable of CPU based rendering in the absence of a GPU
  - "needed" - only for plug-ins, this means that an effect has to have OpenGL support, without which it cannot work.

V1.4: It is now expected from host reporting v1.4 that the plugin can during instance change switch from true to false and false to true.

`kOfxOpenGLPropPixelDepth`

Indicates the bit depths supported by a plug-in during OpenGL renders.

This is analogous to *kOfxImageEffectPropSupportedPixelDepths*. When a plug-in sets this property, the host will try to provide buffers/textures in one of the supported formats. Additionally, the target buffers where the plug-in renders to will be set to one of the supported formats.

Unlike *kOfxImageEffectPropSupportedPixelDepths*, this property is optional. Shader-based effects might not really care about any format specifics when using OpenGL textures, so they can leave this unset and allow the host the decide the format.

- Type - string X N

- Property Set - plugin descriptor (read only)
- Default - none set
- Valid Values - This must be one of
  - *kOfxBitDepthNone* (implying a clip is unconnected, not valid for an image)
  - *kOfxBitDepthByte*
  - *kOfxBitDepthShort*
  - kOfxBitDepthHalf
  - *kOfxBitDepthFloat*

`kOfxImageEffectPropOpenGLEnabled`

Indicates that an image effect SHOULD use OpenGL acceleration in the current action.

When a plugin and host have established they can both use OpenGL renders then when this property has been set the host expects the plugin to render its result into the buffer it has setup before calling the render. The plugin can then also safely use the 'OfxImageEffectOpenGLRenderSuite'

- Type - int X 1
- Property Set - inArgs property set of the following actions…
  - *kOfxImageEffectActionRender*
  - *kOfxImageEffectActionBeginSequenceRender*
  - *kOfxImageEffectActionEndSequenceRender*
- Valid Values
  - 0 indicates that the effect cannot use the OpenGL suite
  - 1 indicates that the effect should render into the texture, and may use the OpenGL suite functions.

v1.4: kOfxImageEffectPropOpenGLEnabled should probably be checked in Instance Changed prior to try to read image via clipLoadTexture

Note: Once this property is set, the host and plug-in have agreed to use OpenGL, so the effect SHOULD access all its images through the OpenGL suite.

`kOfxImageEffectPropOpenGLTextureIndex`

Indicates the texture index of an image turned into an OpenGL texture by the host.

- Type - int X 1
- Property Set - texture handle returned by ` *OfxImageEffectOpenGLRenderSuiteV1::clipLoadTexture* (read only)

```
    This value should be cast to a GLuint and used as the texture index when
    performing OpenGL texture operations.
```

The property set of the following actions should contain this property:

- *kOfxImageEffectActionRender*
- *kOfxImageEffectActionBeginSequenceRender*
- *kOfxImageEffectActionEndSequenceRender*

`kOfxImageEffectPropOpenGLTextureTarget`

Indicates the texture target enumerator of an image turned into an OpenGL texture by the host.

- Type - int X 1
- Property Set - texture handle returned by *OfxImageEffectOpenGLRenderSuiteV1::clipLoadTexture* (read only) This value should be cast to a GLenum and used as the texture target when performing OpenGL texture operations.

The property set of the following actions should contain this property:

- *kOfxImageEffectActionRender*
- *kOfxImageEffectActionBeginSequenceRender*
- *kOfxImageEffectActionEndSequenceRender*

`kOfxParamHostPropSupportsCustomAnimation`

Indicates if the host supports animation of custom parameters.

- Type - int X 1
- Property Set - host descriptor (read only)
- Value Values - 0 or 1

`kOfxParamHostPropSupportsStringAnimation`

Indicates if the host supports animation of string params.

- Type - int X 1
- Property Set - host descriptor (read only)
- Valid Values - 0 or 1

`kOfxParamHostPropSupportsBooleanAnimation`

Indicates if the host supports animation of boolean params.

- Type - int X 1
- Property Set - host descriptor (read only)
- Valid Values - 0 or 1

`kOfxParamHostPropSupportsChoiceAnimation`

Indicates if the host supports animation of choice params.

- Type - int X 1

- Property Set - host descriptor (read only)
- Valid Values - 0 or 1

`kOfxParamHostPropSupportsCustomInteract`
Indicates if the host supports custom interacts for parameters.

- Type - int X 1
- Property Set - host descriptor (read only)
- Valid Values - 0 or 1

`kOfxParamHostPropMaxParameters`
Indicates the maximum numbers of parameters available on the host.

- Type - int X 1
- Property Set - host descriptor (read only)

If set to -1 it implies unlimited number of parameters.

`kOfxParamHostPropMaxPages`
Indicates the maximum number of parameter pages.

- Type - int X 1
- Property Set - host descriptor (read only)

If there is no limit to the number of pages on a host, set this to -1.

Hosts that do not support paged parameter layout should set this to zero.

`kOfxParamHostPropPageRowColumnCount`
This indicates the number of parameter rows and coloumns on a page.

- Type - int X 2
- Property Set - host descriptor (read only)

If the host has supports paged parameter layout, used dimension 0 as the number of columns per page and dimension 1 as the number of rows per page.

`kOfxParamPropInteractV1`
Overrides the parameter's standard user interface with the given interact.

- Type - pointer X 1
- Property Set - plugin parameter descriptor (read/write) and instance (read only)
- Default - NULL
- Valid Values - must point to a OfxPluginEntryPoint

If set, the parameter's normal interface is replaced completely by the interact gui.

`kOfxParamPropInteractSize`

The size of a parameter instance's custom interface in screen pixels.

- Type - double x 2
- Property Set - plugin parameter instance (read only)

This is set by a host to indicate the current size of a custom interface if the plug-in has one. If not this is set to (0,0).

`kOfxParamPropInteractSizeAspect`

The preferred aspect ratio of a parameter's custom interface.

- Type - double x 1
- Property Set - plugin parameter descriptor (read/write) and instance (read only)
- Default - 1.0
- Valid Values - greater than or equal to 0.0

If set to anything other than 0.0, the custom interface for this parameter will be of a size with this aspect ratio (x size/y size).

`kOfxParamPropInteractMinimumSize`

The minimum size of a parameter's custom interface, in screen pixels.

- Type - double x 2
- Property Set - plugin parameter descriptor (read/write) and instance (read only)
- Default - 10,10
- Valid Values - greater than (0, 0)

Any custom interface will not be less than this size.

`kOfxParamPropInteractPreferedSize`

The preferred size of a parameter's custom interface.

- Type - int x 2
- Property Set - plugin parameter descriptor (read/write) and instance (read only)
- Default - 10,10
- Valid Values - greater than (0, 0)

A host should attempt to set a parameter's custom interface on a parameter to be this size if possible, otherwise it will be of *kOfxParamPropInteractSizeAspect* aspect but larger than *kOfxParamPropInteractMinimumSize*.

`kOfxParamPropType`

The type of a parameter.

- Type - C string X 1
- Property Set - plugin parameter descriptor (read only) and instance (read only)

This string will be set to the type that the parameter was create with.

`kOfxParamPropAnimates`

Flags whether a parameter can animate.

- Type - int x 1
- Property Set - plugin parameter descriptor (read/write) and instance (read only)
- Default - 1
- Valid Values - 0 or 1

A plug-in uses this property to indicate if a parameter is able to animate.

`kOfxParamPropCanUndo`

Flags whether changes to a parameter should be put on the undo/redo stack.

- Type - int x 1
- Property Set - plugin parameter descriptor (read/write) and instance (read only)
- Default - 1
- Valid Values - 0 or 1

`kOfxPropParamSetNeedsSyncing`

States whether the plugin needs to resync its private data.

- Type - int X 1
- Property Set - param set instance (read/write)
- Default - 0
- Valid Values -
  - 0 - no need to sync
  - 1 - paramset is not synced

The plugin should set this flag to true whenever any internal state has not been flushed to the set of params.

The host will examine this property each time it does a copy or save operation on the instance. If it is set to 1, the host will call SyncPrivateData and then set it to zero before doing the copy/save. If it is set to 0, the host will assume that the param data correctly represents the private state, and will not call SyncPrivateData before copying/saving. If this property is not set, the host will always call SyncPrivateData before copying or saving the effect (as if the property were set to 1 but the host will not create or modify the property).

`kOfxParamPropIsAnimating`

Flags whether a parameter is currently animating.

- Type - int x 1

- Property Set - plugin parameter instance (read only)
- Valid Values - 0 or 1

Set by a host on a parameter instance to indicate if the parameter has a non-constant value set on it. This can be as a consequence of animation or of scripting modifying the value, or of a parameter being connected to an expression in the host.

`kOfxParamPropPluginMayWrite`

Flags whether the plugin will attempt to set the value of a parameter in some callback or analysis pass.

- Type - int x 1
- Property Set - plugin parameter descriptor (read/write) and instance (read only)
- Default - 0
- Valid Values - 0 or 1

This is used to tell the host whether the plug-in is going to attempt to set the value of the parameter.

Deprecated:

- v1.4: deprecated - to be removed in 1.5

`kOfxParamPropPersistant`

Flags whether the value of a parameter should persist.

- Type - int x 1
- Property Set - plugin parameter descriptor (read/write) and instance (read only)
- Default - 1
- Valid Values - 0 or 1

This is used to tell the host whether the value of the parameter is important and should be save in any description of the plug-in.

`kOfxParamPropEvaluateOnChange`

Flags whether changing a parameter's value forces an evalution (ie: render),.

- Type - int x 1
- Property Set - plugin parameter descriptor (read/write) and instance (read/write only)
- Default - 1
- Valid Values - 0 or 1

This is used to indicate if the value of a parameter has any affect on an effect's output, eg: the parameter may be purely for GUI purposes, and so changing its value should not trigger a re-render.

`kOfxParamPropSecret`

Flags whether a parameter should be exposed to a user,.

- Type - int x 1
- Property Set - plugin parameter descriptor (read/write) and instance (read/write)
- Default - 0
- Valid Values - 0 or 1

If secret, a parameter is not exposed to a user in any interface, but should otherwise behave as a normal parameter.

Secret params are typically used to hide important state detail that would otherwise be unintelligible to a user, for example the result of a statical analysis that might need many parameters to store.

`kOfxParamPropScriptName`

The value to be used as the id of the parameter in a host scripting language.

- Type - ASCII C string X 1,
- Property Set - plugin parameter descriptor (read/write) and instance (read only),
- Default - the unique name the parameter was created with.
- Valid Values - ASCII string unique to all parameters in the plug-in.

Many hosts have a scripting language that they use to set values of parameters and more. If so, this is the name of a parameter in such scripts.

`kOfxParamPropCacheInvalidation`

Specifies how modifying the value of a param will affect any output of an effect over time.

- Type - C string X 1
- Property Set - plugin parameter descriptor (read/write) and instance (read only),
- Default - kOfxParamInvalidateValueChange
- Valid Values - This must be one of
  - kOfxParamInvalidateValueChange
  - kOfxParamInvalidateValueChangeToEnd
  - kOfxParamInvalidateAll

Imagine an effect with an animating parameter in a host that caches rendered output. Think of the what happens when you add a new key frame. -If the parameter represents something like an absolute position, the cache will only need to be invalidated for the range of frames that keyframe affects.

- If the parameter represents something like a speed which is integrated, the cache will be invalidated from the keyframe until the end of the clip.
- There are potentially other situations where the entire cache will need to be invalidated (though I can't think of one off the top of my head).

`kOfxParamPropHint`

A hint to the user as to how the parameter is to be used.

- Type - UTF8 C string X 1

- Property Set - plugin parameter descriptor (read/write) and instance (read/write),
- Default - “”

`kOfxParamPropDefault`

The default value of a parameter.

- Type - The type is dependant on the parameter type as is the dimension.
- Property Set - plugin parameter descriptor (read/write) and instance (read/write only),
- Default - 0 cast to the relevant type (or “” for strings and custom parameters)

The exact type and dimension is dependant on the type of the parameter. These are. . . .

- *kOfxParamTypeInteger* - integer property of one dimension
- *kOfxParamTypeDouble* - double property of one dimension
- *kOfxParamTypeBoolean* - integer property of one dimension
- *kOfxParamTypeChoice* - integer property of one dimension
- *kOfxParamTypeRGBA* - double property of four dimensions
- *kOfxParamTypeRGB* - double property of three dimensions
- *kOfxParamTypeDouble2D* - double property of two dimensions
- *kOfxParamTypeInteger2D* - integer property of two dimensions
- *kOfxParamTypeDouble3D* - double property of three dimensions
- *kOfxParamTypeInteger3D* - integer property of three dimensions
- *kOfxParamTypeString* - string property of one dimension
- *kOfxParamTypeCustom* - string property of one dimension
- *kOfxParamTypeGroup* - does not have this property
- *kOfxParamTypePage* - does not have this property
- *kOfxParamTypePushButton* - does not have this property

`kOfxParamPropDoubleType`

Describes how the double parameter should be interpreted by a host.

- Type - C string X 1
- Default - *kOfxParamDoubleTypePlain*
- Property Set - 1D, 2D and 3D float plugin parameter descriptor (read/write) and instance (read only),
- Valid Values -This must be one of
  - *kOfxParamDoubleTypePlain* - parameter has no special interpretation,
  - *kOfxParamDoubleTypeAngle* - parameter is to be interpretted as an angle,
  - *kOfxParamDoubleTypeScale* - parameter is to be interpretted as a scale factor,
  - *kOfxParamDoubleTypeTime* - parameter represents a time value (1D only),
  - *kOfxParamDoubleTypeAbsoluteTime* - parameter represents an absolute time value (1D only),

   - *kOfxParamDoubleTypeX* - size wrt to the project's X dimension (1D only), in canonical coordinates,
   - *kOfxParamDoubleTypeXAbsolute* - absolute position on the X axis (1D only), in canonical coordinates,
   - *kOfxParamDoubleTypeY* - size wrt to the project's Y dimension(1D only), in canonical coordinates,
   - *kOfxParamDoubleTypeYAbsolute* - absolute position on the Y axis (1D only), in canonical coordinates,
   - *kOfxParamDoubleTypeXY* - size in 2D (2D only), in canonical coordinates,
   - *kOfxParamDoubleTypeXYAbsolute* - an absolute position on the image plane, in canonical coordinates.

Double parameters can be interpreted in several different ways, this property tells the host how to do so and thus gives hints as to the interface of the parameter.

`kOfxParamPropDefaultCoordinateSystem`

Describes in which coordinate system a spatial double parameter's default value is specified.

- Type - C string X 1
- Default - kOfxParamCoordinatesCanonical
- Property Set - Non normalised spatial double parameters, ie: any double param who's *kOfxParamPropDoubleType* is set to one of…
   - kOfxParamDoubleTypeX
   - kOfxParamDoubleTypeXAbsolute
   - kOfxParamDoubleTypeY
   - kOfxParamDoubleTypeYAbsolute
   - kOfxParamDoubleTypeXY
   - kOfxParamDoubleTypeXYAbsolute
- Valid Values - This must be one of
   - kOfxParamCoordinatesCanonical - the default is in canonical coords
   - kOfxParamCoordinatesNormalised - the default is in normalised coordinates

This allows a spatial param to specify what its default is, so by saying normalised and "0.5" it would be in the 'middle', by saying canonical and 100 it would be at value 100 independent of the size of the image being applied to.

`kOfxParamPropHasHostOverlayHandle`

A flag to indicate if there is a host overlay UI handle for the given parameter.

- Type - int x 1
- Property Set - plugin parameter descriptor (read only)
- Valid Values - 0 or 1

If set to 1, then the host is flagging that there is some sort of native user overlay interface handle available for the given parameter.

`kOfxParamPropUseHostOverlayHandle`

A flag to indicate that the host should use a native UI overlay handle for the given parameter.

- Type - int x 1
- Property Set - plugin parameter descriptor (read/write only) and instance (read only)
- Default - 0
- Valid Values - 0 or 1

If set to 1, then a plugin is flaging to the host that the host should use a native UI overlay handle for the given parameter. A plugin can use this to keep a native look and feel for parameter handles. A plugin can use *kOfxParamPropHasHostOverlayHandle* to see if handles are available on the given parameter.

`kOfxParamPropShowTimeMarker`

Enables the display of a time marker on the host's time line to indicate the value of the absolute time param.

- Type - int x 1
- Property Set - plugin parameter descriptor (read/write) and instance (read/write)
- Default - 0
- Valid Values - 0 or 1

If a double parameter is has *kOfxParamPropDoubleType* set to *kOfxParamDoubleTypeAbsoluteTime*, then this indicates whether any marker should be made visible on the host's time line.

`kOfxPluginPropParamPageOrder`

Sets the parameter pages and order of pages.

- Type - C string X N
- Property Set - plugin parameter descriptor (read/write) and instance (read only)
- Default - ""
- Valid Values - the names of any page param in the plugin

This property sets the preferred order of parameter pages on a host. If this is never set, the preferred order is the order the parameters were declared in.

`kOfxParamPropPageChild`

The names of the parameters included in a page parameter.

- Type - C string X N
- Property Set - plugin parameter descriptor (read/write) and instance (read only)
- Default - ""
- Valid Values - the names of any parameter that is not a group or page, as well as *kOfxParamPageSkipRow* and *kOfxParamPageSkipColumn*

This is a property on parameters of type *kOfxParamTypePage*, and tells the page what parameters it contains. The parameters are added to the page from the top left, filling in columns as we go. The two pseudo param names *kOfxParamPageSkipRow* and *kOfxParamPageSkipColumn* are used to control layout.

Note parameters can appear in more than one page.

`kOfxParamPropParent`

The name of a parameter's parent group.

- Type - C string X 1
- Property Set - plugin parameter descriptor (read/write) and instance (read only),
- Default - "", which implies the "root" of the hierarchy,
- Valid Values - the name of a parameter with type of *kOfxParamTypeGroup*

Hosts that have hierarchical layouts of their params use this to recursively group parameter.

By default parameters are added in order of declaration to the 'root' hierarchy. This property is used to reparent params to a predefined param of type *kOfxParamTypeGroup*.

`kOfxParamPropGroupOpen`

Whether the initial state of a group is open or closed in a hierarchical layout.

- Type - int X 1
- Property Set - plugin parameter descriptor (read/write) and instance (read only)
- Default - 1
- Valid Values - 0 or 1

This is a property on parameters of type *kOfxParamTypeGroup*, and tells the group whether it should be open or closed by default.

`kOfxParamPropEnabled`

Used to enable a parameter in the user interface.

- Type - int X 1
- Property Set - plugin parameter descriptor (read/write) and instance (read/write),
- Default - 1
- Valid Values - 0 or 1

When set to 0 a user should not be able to modify the value of the parameter. Note that the plug-in itself can still change the value of a disabled parameter.

`kOfxParamPropDataPtr`

A private data pointer that the plug-in can store its own data behind.

- Type - pointer X 1
- Property Set - plugin parameter instance (read/write),
- Default - NULL

This data pointer is unique to each parameter instance, so two instances of the same parameter do not share the same data pointer. Use it to hang any needed private data structures.

`kOfxParamPropChoiceOption`

Set an option in a choice parameter.

- Type - UTF8 C string X N
- Property Set - plugin parameter descriptor (read/write) and instance (read/write),
- Default - the property is empty with no options set.

This property contains the set of options that will be presented to a user from a choice parameter. See ParametersChoice for more details.

`kOfxParamPropMin`

The minimum value for a numeric parameter.

- Type - int or double X N
- Property Set - plugin parameter descriptor (read/write) and instance (read/write),
- Default - the smallest possible value corresponding to the parameter type (eg: INT_MIN for an integer, -DBL_MAX for a double parameter)

Setting this will also reset *kOfxParamPropDisplayMin*.

`kOfxParamPropMax`

The maximum value for a numeric parameter.

- Type - int or double X N
- Property Set - plugin parameter descriptor (read/write) and instance (read/write),
- Default - the largest possible value corresponding to the parameter type (eg: INT_MAX for an integer, DBL_MAX for a double parameter)

Setting this will also reset :;kOfxParamPropDisplayMax.

`kOfxParamPropDisplayMin`

The minimum value for a numeric parameter on any user interface.

- Type - int or double X N
- Property Set - plugin parameter descriptor (read/write) and instance (read/write),
- Default - the smallest possible value corresponding to the parameter type (eg: INT_MIN for an integer, -DBL_MAX for a double parameter)

If a user interface represents a parameter with a slider or similar, this should be the minumum bound on that slider.

`kOfxParamPropDisplayMax`

The maximum value for a numeric parameter on any user interface.

- Type - int or double X N

- Property Set - plugin parameter descriptor (read/write) and instance (read/write),
- Default - the largest possible value corresponding to the parameter type (eg: INT_MAX for an integer, DBL_MAX for a double parameter)

If a user interface represents a parameter with a slider or similar, this should be the maximum bound on that slider.

`kOfxParamPropIncrement`

The granularity of a slider used to represent a numeric parameter.

- Type - double X 1
- Property Set - plugin parameter descriptor (read/write) and instance (read/write),
- Default - 1
- Valid Values - any greater than 0.

This value is always in canonical coordinates for double parameters that are normalised.

`kOfxParamPropDigits`

How many digits after a decimal point to display for a double param in a GUI.

- Type - int X 1
- Property Set - plugin parameter descriptor (read/write) and instance (read/write),
- Default - 2
- Valid Values - any greater than 0.

This applies to double params of any dimension.

`kOfxParamPropDimensionLabel`

Label for individual dimensions on a multidimensional numeric parameter.

- Type - UTF8 C string X 1
- Property Set - plugin parameter descriptor (read/write) and instance (read only),
- Default - “x”, “y” and “z”
- Valid Values - any

Use this on 2D and 3D double and integer parameters to change the label on an individual dimension in any GUI for that parameter.

`kOfxParamPropIsAutoKeying`

Will a value change on the parameter add automatic keyframes.

- Type - int X 1
- Property Set - plugin parameter instance (read only),
- Valid Values - 0 or 1

This is set by the host simply to indicate the state of the property.

`kOfxParamPropCustomInterpCallbackV1`

A pointer to a custom parameter's interpolation function.

- Type - pointer X 1
- Property Set - plugin parameter descriptor (read/write) and instance (read only),
- Default - NULL
- Valid Values - must point to a *OfxCustomParamInterpFuncV1*

It is an error not to set this property in a custom parameter during a plugin's define call if the custom parameter declares itself to be an animating parameter.

`kOfxParamPropStringMode`

Used to indicate the type of a string parameter.

- Type - C string X 1
- Property Set - plugin string parameter descriptor (read/write) and instance (read only),
- Default - *kOfxParamStringIsSingleLine*
- Valid Values - This must be one of the following
  - *kOfxParamStringIsSingleLine*
  - *kOfxParamStringIsMultiLine*
  - *kOfxParamStringIsFilePath*
  - *kOfxParamStringIsDirectoryPath*
  - *kOfxParamStringIsLabel*
  - kOfxParamStringIsRichTextFormat

`kOfxParamPropCustomValue`

Used by interpolating custom parameters to get and set interpolated values.

- Type - C string X 1 or 2

This property is on the *inArgs* property and *outArgs* property of a *OfxCustomParamInterpFuncV1* and in both cases contains the encoded value of a custom parameter. As an *inArgs* property it will have two values, being the two keyframes to interpolate. As an *outArgs* property it will have a single value and the plugin should fill this with the encoded interpolated value of the parameter.

`kOfxParamPropInterpolationTime`

Used by interpolating custom parameters to indicate the time a key occurs at.

- Type - double X 2
- Property Set - inArgs parameter of a *OfxCustomParamInterpFuncV1* (read only)

The two values indicate the absolute times the surrounding keyframes occur at. The keyframes are encoded in a *kOfxParamPropCustomValue* property.

`kOfxParamPropInterpolationAmount`

Property used by *OfxCustomParamInterpFuncV1* to indicate the amount of interpolation to perform.

- Type - double X 1
- Property Set - inArgs parameter of a *OfxCustomParamInterpFuncV1* (read only)
- Valid Values - from 0 to 1

This property indicates how far between the two *kOfxParamPropCustomValue* keys to interpolate.

`kOfxParamPropParametricDimension`

The dimension of a parametric param.

- Type - int X 1
- Property Set - parametric param descriptor (read/write) and instance (read only)
- default - 1
- Value Values - greater than 0

This indicates the dimension of the parametric param.

`kOfxParamPropParametricUIColour`

The colour of parametric param curve interface in any UI.

- Type - double X N
- Property Set - parametric param descriptor (read/write) and instance (read only)
- default - unset,
- Value Values - three values for each dimension (see *kOfxParamPropParametricDimension*) being interpretted as R, G and B of the colour for each curve drawn in the UI.

This sets the colour of a parametric param curve drawn a host user interface. A colour triple is needed for each dimension of the oparametric param.

If not set, the host should generally draw these in white.

`kOfxParamPropParametricInteractBackground`

Interact entry point to draw the background of a parametric parameter.

- Type - pointer X 1
- Property Set - plug-in parametric parameter descriptor (read/write) and instance (read only),
- Default - NULL, which implies the host should draw its default background.

Defines a pointer to an interact which will be used to draw the background of a parametric parameter's user interface. None of the pen or keyboard actions can ever be called on the interact.

The openGL transform will be set so that it is an orthographic transform that maps directly to the 'parametric' space, so that 'x' represents the parametric position and 'y' represents the evaluated value.

`kOfxParamHostPropSupportsParametricAnimation`

Property on the host to indicate support for parametric parameter animation.

- Type - int X 1
- Property Set - host descriptor (read only)
- Valid Values
  - 0 indicating the host does not support animation of parmetric params,
  - 1 indicating the host does support animation of parmetric params,

`kOfxParamPropParametricRange`

Property to indicate the min and max range of the parametric input value.

- Type - double X 2
- Property Set - parameter descriptor (read/write only), and instance (read only)
- Default Value - (0, 1)
- Valid Values - any pair of numbers so that the first is less than the second.

This controls the min and max values that the parameter will be evaluated at.

1.21 Status Codes

Status codes are returned by most functions in OFX suites and all plug-in actions to indicate the success or failure of the operation. All status codes are defined in ofxCore.h and *#defined* to be integers.

typedef int **`OfxStatus`**

OFX status return type.

Most OFX functions in host suites and all actions in a plug-in return a status code, where the status codes are all 32 bit integers. This typedef is used to label that status code.

`kOfxStatOK`

Status code indicating all was fine.

`kOfxStatFailed`

Status error code for a failed operation.

`kOfxStatErrFatal`

Status error code for a fatal error.

Only returned in the case where the plug-in or host cannot continue to function and needs to be restarted.

`kOfxStatErrUnknown`

Status error code for an operation on or request for an unknown object.

`kOfxStatErrMissingHostFeature`
Status error code returned by plug-ins when they are missing host functionality, either an API or some optional functionality (eg: custom params).

Plug-Ins returning this should post an appropriate error message stating what they are missing.

`kOfxStatErrUnsupported`
Status error code for an unsupported feature/operation.

`kOfxStatErrExists`
Status error code for an operation attempting to create something that exists.

`kOfxStatErrFormat`
Status error code for an incorrect format.

`kOfxStatErrMemory`
Status error code indicating that something failed due to memory shortage.

`kOfxStatErrBadHandle`
Status error code for an operation on a bad handle.

`kOfxStatErrBadIndex`
Status error code indicating that a given index was invalid or unavailable.

`kOfxStatErrValue`
Status error code indicating that something failed due an illegal value.

`kOfxStatReplyYes`
OfxStatus returned indicating a 'yes'.

`kOfxStatReplyNo`
OfxStatus returned indicating a 'no'.

`kOfxStatReplyDefault`
OfxStatus returned indicating that a default action should be performed.

1.22 Changes to the API for 1.2

1.22.1 Introduction

This chapter lists the changes and extensions between the 1.1 version of the API and the 1.2 version of the API. The extension are backwards compatible, so that a 1.2 plugin will run on an earlier version of a host, provided a bit of care is taken. A 1.2 host will easily support a plugin written to an earlier API.

1.22.2 Packaging

A new architecture directory was added to the bundle hierarchy to specifically contain Mac OSX 64 bit builds. The current 'MacOS' architecture is a fall back for 32 bit only and/or universal binary builds.

1.22.3 Versioning

Three new properties are provided to identify and version a plugin and/or host. These are...

- `kOfxPropAPIVersion` a multi-dimensional integer that specifies the version of the API being implemented by a host.
- `kOfxPropVersion` a multi-dimensional integer that provides a version number for host and plugin
- `kOfxPropVersionLabel` a user readable version label

Before 1.2 there was no way to identify the version of a host application, which a plugin could use to work around known bugs and problems in known versions. `kOfxPropVersion` provides a way to do that.

1.22.4 Plugin Description

The new property `kOfxPropPluginDescription` allows a plugin to set a string which provides a description to a user.

1.22.5 Parameter Groups and Icons

Group parameters are typically displayed in a hierarchical manner on many applications, with 'twirlies' to open and close the group. The new property `kOfxParamPropGroupOpen` is used to specify if a group parameter should be initially open or closed.

Some applications are able to display icons instead of text when labelling parameters. The new property, `kOfxPropIcon`, specifies an SVG file and/or a PNG file to use as an icon in such host applications.

1.22.6 New Message Suite

A new message suite has been specified, `OfxMessageSuiteV2`, this adds two new functions. One to set a persistent message on an effect, and a second to clear that message. This would typically be used to flag an error on an effects.

1.22.7 New Syncing Property

A new property has been added to parameter sets, `kOfxPropParamSetNeedsSyncing`. This is used by plugins with internal data structures that need syncing back to parameters for persistence and so on. This property should be set whenever the plugin changes it's internal state to inform the host that a sync will be required before the next serialisation of the plugin. Without this property, the host would have to continually force the plugin to sync it's private data, whether that was a redundant operation or not. For large data sets, this can be a significant overhead.

1.22.8 Sequential Rendering

Flagging sequential rendering has been slightly modified. The *kOfxImageEffectInstancePropSequentialRender* property has had a third allowed state added, which indicate that a plugin would prefer to be sequentially rendered if possible, but need not be.

The *kOfxImageEffectInstancePropSequentialRender* property has also been added to the host descriptor, to indicate whether the host can support sequential rendering.

The new property *kOfxImageEffectPropSequentialRenderStatus* is now passed to the render actions to indicate that a host is currently sequentially rendering or not.

1.22.9 Interactive Render Notification

A new property has been added to flag a render as being in response to an interactive change by a user, as opposed to a batch render. This is *kOfxImageEffectPropInteractiveRenderStatus*

1.22.10 Host Operating System Handle

A new property has been added to allow a plugin to request the host operating system specific application handle (ie: on Windows (tm) this would be the application's root hWnd). This is *kOfxPropHostOSHandle*

1.22.11 Non Normalised Spatial Parameters

Normalised double parameters have proved to be more of a problem than expected. The major idea was to provide resolution independence for spatial parameters. However, in practice, having to specify parameters as a fraction of a yet to be determined resolution is problematic. For example, if you want to set something to be explicitly '20', there is no way of doing that. The main problem stems from normalised params conflating two separate issues, flagging to the host that a parameter was spatial, and being able to specify defaults in a normalised co-ordinate system.

With 1.2 new spatial double parameter types are defined. These have their values manipulated in canonical coordinates, however, they have an option to specify their default values in a normalise coordinate system. These are. . . .

These new double parameter types are. . . .

- *kOfxParamDoubleTypeX* - a size in the X dimension dimension (1D only), new for 1.2
- *kOfxParamDoubleTypeXAbsolute* - a position in the X dimension (1D only), new for 1.2
- *kOfxParamDoubleTypeY* - a size in the Y dimension dimension (1D only), new for 1.2
- *kOfxParamDoubleTypeYAbsolute* - a position in the X dimension (1D only), new for 1.2
- *kOfxParamDoubleTypeXY* - a size in the X and Y dimension (2D only), new for 1.2
- *kOfxParamDoubleTypeXYAbsolute* - a position in the X and Y dimension (2D only), new for 1.2

These new parameter types can set their defaults in one of two coordinate systems, the property *kOfxParamPropDefaultCoordinateSystem* Specifies the coordinate system the default value is being specified in.

Plugins can check *kOfxPropAPIVersion* to see if these new parameter types are supported

1.22.12 Native Overlay Handles

Some applications have their own overlay handles for certain types of parameter (eg: spatial positions). It is often better to rely on those, than have a plugin implement their own overlay handles. Two new parameter, properties are available to do that, one used by the host to indicate if such handles are available. The other by a plugin telling the host to use such handle.

- *`kOfxParamPropHasHostOverlayHandle`* indicates a parameter has an host native overlay handle
- *`kOfxParamPropUseHostOverlayHandle`* indicates that a host should use a native overlay handle.

1.22.13 Interact Colour Hint

Some applications allow the user to specify colours of any overlay via a colour picker. Plug-ins can access this via the *`kOfxInteractPropSuggestedColour`* property.

1.22.14 Interact Viewport Pen Position

The new property *`kOfxInteractPropPenViewportPosition`* is used to pass a pen position in viewport coordinate, rather than a connaonical. This is sometimes much more convenient. It is passed to all actions that *`kOfxInteractPropPenPosition`* is passed to.

1.22.15 Parametric Parameters

A new optional parameter type, and supporting suite, is introduced, parametric parameters. This allows for the construction of user defined lookup tables and so on.

CHAPTER

TWO

OPENFX PROGRAMMING GUIDE

This is a mostly complete reference guide to the OFX image effect plugin architecture. It is a backwards compatible update to the 1.2 version of the API. The changes to the API are listed in an addendum.

2.1 Foreword

OFX is an open API for writing visual effects plug-ins for a wide variety of applications, such as video editing systems and compositing systems. This guide demonstrates by example the low level C APIs that defines OFX.

2.2 Intended Audience

Do you write visual effects or image processing software? Do you have an application which deals with moving images and hosts plug-ins or would like to host plug-ins? Then OFX is for you.

This guide assumes you can program in the `C` language and are familiar with the concepts involved in writing visual effects software. You need to understand concepts like pixels, clips, pixel aspect ratios and so on. If you don't, I suggest you read further or attempt to soldier on bravely and see how far you go before you get lost.

2.3 What is OFX?

OFX is actually several things. At the lowest level OFX is a generic `C` based plug-in architecture that can be used to define any kind of plug-in API. You could use this low level architecture to implement any API, however it was originally designed to host our visual effects image processing API. The basic architecture could be re-used to create other higher level APIs such as a sound effects API, a 3D API and more.

This guide describes the basic OFX plug-in architecture and the visual effects plug-in API built on top of it. The visual effects API is very broad and intended to allow visual effects plug-ins to work on a wide range of host applications, including compositing hosts, rotoscopers, encoding applications, colour grading hosts, editing hosts and more I haven't thought of yet. While all these type of applications process images, they often have very different work flows and present effects to a user in incompatible ways. OFX is an attempt to deal with all of these in a clear and consistent manner.

The API is by design feature rich, not all aspects of the API map to all hosts. This is to allow different host developers to implement OFX support in a manner that best fits their applications' capabilities.

Hosts are encouraged to extend OFX by providing extra proprietary suites, actions, properties and settings to extend the capabilities of the API. It would be nice that a broadly useful proprietary extension be put forward for incorporation into the open standard.

That said although there is no validation process in terms of what is an OFX host, a small minimal set of expectations is assumed, which we will cover in the following guides.

2.4 The Examples

I'll illustrate the API and how it works with a variety of example plugins, each of which will have it's own guide describing what is going one. You should work through them one by one as each will build on the one before.

For completeness and clarity of explanation, each plugin is entirely self contained and has no dependency on anything other than standard C and C++ libraries and the OFX headers.

- The basic machinery of an OFX plugin.
- How to access images.
- How to define parameters.
- How to write multi context effects.
- Coordinate systems and defining regions of definition.

2.5 Wrapping the API

This API can be somewhat awkward to use directly, and it is expected that most plugin or host developers will wrap the API in higher level C or C++ structures.

There are open source host and plugin side API C wrappers available from the openfx.org git repository. As you work through the examples you'll see that I actually start wrapping up various entities within the API into `C` classes as it can get unwieldy otherwise.

2.6 License

Please feel free to use any of the code you find here, provided you adhere to the BSD style license you'll find at the top of each header file.

```
This directory tree contains a set of plugins and corresponding
guides which take you through the basics of the OFX
Image Effects Plugin API.

There are two sub-directories...
   - Code - which contains the example plugins source files,
   - Doc  - has a guide to each plugin.

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
BUILDING THE PLUGINS

For Windows instructions, see below.


To build the example plugins you will need,
   - a C++ compiler
   - gmake (or nmake on Windows)
   - the ofx header files.


Within Code there is a subdirectory per plugin.
```

(continues on next page)

(continued from previous page)

```
The assumption is that you have checked out all the OFX
source code in one lump and so the
OFX header files will be in a standard relative path to
the plugin sources. If this is not the case you will
need to modify the file...

   Code/MakefileCommon

and change the line

   OFX_INC_DIR = -I../../../include

to point to the directory where you have put the headers.


To build all the examples simply go...

    cd Code
    make

this will compile all the plugins and place them in
a directory called 'built_plugins'.


You can build individual plugins by changing into the
relevant subdirectory and simply issuing a 'make' command.

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
BUILDING ON WINDOWS

NMakefiles are included for use with Windows' nmake utility.
This should build on any Visual Studio version (at least 2008 or newer).

Open a Visual Studio command-line window of the appropriate bitness
that you want (32 for a 32-bit OFX host, 64 for a 64-bit host).  From
the Start menu, go to Microsoft Visual Studio XXXX -> Visual Studio
Tools -> Visual Studio XXX Command Prompt (choose the appropriate
bitness here).

In that window, cd to the openfx/Guide/Code dir, and type:

  nmake /F nmakefile install

This will build and install the plugins into the standard OFX plugins
dir (c:\Program Files\Common Files\OFX\Plugins).

To clean up:

  nmake /F nmakefile clean
```

(continues on next page)

(continued from previous page)

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
BUILDING THE DOCUMENTATION

To build the documentation you will need...
   - gmake
   - asciidoctor


The documentation is written in asciidoctor markdown, which can be used to
generate HTML, docbook XML and more. You will need to download and install
asciidoctor to build the doc. Visit...

    http://asciidoctor.org/

for installation instructions.


There is a gnu Makefile currently configured to generate html files. To build
the documentation simply go...

   cd Doc
   make

this will generate a subdirectory called 'html' which will contain the
guides in html format.



Last Edit 11/11/14
```

```
This is a brief description of examples which could
do with being added to the Guides.

The following examples should be added to the guide.

OpenGL Overlay Example
======================
A trivial example along the lines of the circle drawing example.

This will illustrate...
    - openGL overlays


Temporal Difference Example
===========================
Compute the absolute difference between images at two different times.

This will illustrate...
    - fetching images at separate times
    - the get frames needed action

Transition Example
```

(continues on next page)

(continued from previous page)

```
===========================
A simple straight frame blend transition

This will illustrate...
   - the transition context


Custom Parameter
================
Something like the circle drawing plugin so that it has no double parameters, but only a␣
↪single custom parameter, controlled via the overlay UI.

This will illustrate...
    - using custom parameters.


3x3 2D Filter
=============
Does a variety of simple 2D filtering operations using a 3x3 window.

This will illustrate...
   - the get region of interest action


Analysis Plugin
===============
The plugin will find the minimum and maximum value of a given frame in response to a␣
↪push button, and store the values
into two parameters. During renders it will rescale pixel values so that 'min' is 0 and
↪'max' is the whitepoint.

This will illustrate...
    - writing to parameters in response to a button press outside of render.


Basic OpenGL Example
==============
Behaviour to be determined.

This will illustrate the basics of rendering in OpenGL

Last Edit 11/11/14
```

INDEX