



2020 3D Media Spatial Sound and Vision

Developing new tools for stereo imaging postproduction

Project ref. no.	ICT-FP7-215475
Project acronim	2020 3D Media
Document due Date :	26 June, 2008
Prepared by	Miriam Balaguer, Barcelona Media
Document status	Revision

Version	Date	Description
1.0	26/06/2008	Common development architecture description (OpenFx)

Developing new tools for stereo imaging postproduction

Because the huge demanding of fast postproduction solutions in the stereographics production pipeline, several new tools need to be developed. To straightforward this process, a good approach could be to include support for stereo in current software packages. A common standard in this environment is the OpenFX architecture.

This document covers the next topics:

- Introduction to the OpenFX architecture (OFX) 2
- Developing OpenFX plug-ins 3
- The OFX Image Effect API 4
- OpenFX contexts 5
- Plug-in input/output parameters 5
- Stereoscopic postproduction plug-ins 6

Introduction to the OpenFX architecture (OFX)

OpenFX (<http://openfx.sourceforge.net/>) is an open architecture that could be used to implement a variety of plug-in APIs such as sound effects API, 3D API, etc.

OpenFX grew during 2002 between Bruno Nicoletti of *The Foundry* and Tom Benoist of *Interactive Effects*. The OpenFX standard is currently owned by The Foundry, however they have released it under a BSD (*Berkeley Software Distribution*) open source license.

Nowadays, the platform is supported by Mac OS X, both 32 and 64 bit versions of Windows XP and Windows Vista and on either 32-bit or 64-bit versions of Linux.

At the lowest level, OFX is a generic C++ based plug-in architecture that can be used to implement higher level APIs.

While most of composing and processing image applications often have very different workflows and present effects in incompatible ways, OFX is an attempt to deal with all of these in a consistent manner. For this reason, its most important feature is that it could avoid compatibility difficulties by setting a single open standard for graphic and audio plug-ins.

In the context of the 2020 3D Media project, OpenFX could be a valuable platform to implement stereoscopic plug-ins. The aim of this document is to research and study whether OpenFX is a good architecture to exchange data and to develop closer integrations between project partners.

Stereoscopic plug-ins will have different purposes. On the one hand, they might be powerful import and export tools for 3D video formats, allowing as much

encodings as possible in order to be suitable for use with existing postproduction tools in the market.

On the other hand, to develop other postproduction tools to supply actual stereography necessities. That is to say, to create stereographic plug-ins which simulate the features of existing monoscopic postproduction techniques.

Developing OpenFX plug-ins

OFX image effect plug-ins should use the *OFX Image Effect API* to implement visual effects. This API is defined in the OpenFX core.

Depending on the 3D video image format, stereoscopic plug-ins need different input images and processing methods to generate the stereographic output. So it is important to establish the way to build these tools using the structures provided by the OpenFX APIs.

The host relies on two symbols within a plug-in, all other communication is bootstrapped from those two symbols, so the plug-in has minimal symbolic dependencies from the host. These symbolic dependencies allow for run-time determination of what features to provide over the API, making implementation much more flexible.

The plug-ins, via these two symbols, should indicate the API they implement, the version of the API, their name, their version and their main entry point.

To create a new plug-in it is necessary to implement some specific methods used by the host to identify the plug-in and to load/unload it. These methods are contained in the main plug-in function (denominated "*pluginMain*"). This function is where the host tells the plug-in what it needs to do. It is passed to the host in the "*OfxPlugin*" struct. This struct is returned by the host method "*OfxGetPlugin*".

More precisely, the required functions in the plug-in are:

- The "*getPluginID*" method which provides the unique name and version of the plug-in.
- The "*loadAction*" and "*unloadAction*" methods that should be called, respectively, after the plug-in is loaded the first time, and before the plug-in is unloaded and all instances have been destroyed.
- The "*describe*" function which tells the host the overall behaviour of the plug-in. And the "*describeInContext*" method that is called once for each context that a plug-in says it can work in.
- The "*createInstance*" function which should be called when a new instance of a plug-in needs to be created.

The "*render*" function is where a plug-in turns its input images into output images. So this method could be overridden to define all the necessary actions needed to handle the input images and obtain the output images in the stereographic format desired.

The host loads every plug-in from a binary file. This file could contain more than one plug-in. Therefore, after loading the binary, the host uses an OFX function to identify the exact number of plug-ins included in it. Then, for each plug-in inside the binary, the host recovers the *OfxPlugin* struct representing the plug-in.

The objects passed to a plug-in are generally specified by a blind data pointer to allow the object implementation to be hidden from the plug-in. We can take advantage of this pointer to enter the necessary input data from the host. This data will depend on the stereoscopic format.

For example, as mentioned before, to generate the 3D depth impression it is necessary to process at least two different input images: the 2D RGB image and its depth map. In this case, the plug-in should need more than one input parameter to load data.

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

A plug-in communicates with a host by using sets of functions pointers given by the host. They are returned on request from the host as pointers within a C struct.

The OFX Image Effect API

The advantage of using OpenFX to implement stereographic tools is that it contains an Image Effect API designed to develop image effect plug-ins for visual effects. Generally, this API allows the host to send a set of images to a plug-in, stating the value of a set of parameters and getting the result back.

To label the developed plug-ins as image effect plug-ins, it is enough to set the *pluginApi* member of the *OfxPlugin* struct to be of the type *kOfxImageEffectPluginApi*, and to assign the API version to 1.0.

This API uses a variety of different objects and actions to trap all the standard actions used to drive the plug-in.

An *effect* is an object that represents an image processing plug-in that has associated a set of image clips and properties. The *pluginMain* function receives a handle to an image effect.

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

The *parameters* are user visible objects that an effect uses to specify its state. They are represented as blind data handles of type *OfxParamHandle*.

An *image instance* is an instance object that maintains the state about a 2D image being passed to an effect instance.

OpenFX contexts

How an image effect will be used in the plug-in affects on how it should interact with the host application. For that reason, OFX has standardized several different uses and have called them *contexts*. A context mandates certain behaviours from an effect when it is described or instantiated in one specific context. The major issues are the number of input clips it takes and how it can interact with those input clips. Besides, all the contexts have a single output clip.

In the case of stereography, stereographic tools should have an arbitrary number of inputs (images and parameters), depending on the 3D video input format. As a consequence, having analysed the different types of image effect contexts, we have considered that the most appropriate might be the *general context*.

In this context the effects can take multiple inputs. Moreover, it has no constraints on the pixel preferences actions. The only restriction is that it only returns one output clip.

As exposed before, in the *describeInContext* plug-in action, an effect should define the clips required for this *general context* and set the clip properties to control its use. This fact is important in stereographic plug-ins, because they have multiple inputs that may have different properties.

Once the clips are defined, it is possible to recover images from clips. The image effect API provides functions to extract an image from a given clip in a specific time.

Furthermore, if a plug-in needs to define its own temporary image buffers during processing or to cache images between actions, then this API offers memory allocation routines to it.

It is also possible to define some stereography plug-ins with another context: the *paint context*.

Paint effects are effects used inside a digital painting system, where the effect is limited to a small area of the source image via a masking image. The plug-in has some mandated objects. These objects are the input source clip to perform, another input clip which represents the painter mask and the output clip.

The drawback is that this context only accepts two input clips being one of them the painter mask. Therefore, to create the postproduction tools it would be the desired context.

Plug-in input/output parameters

To control the behaviour of the plug-ins, it is necessary to define some sort of parameters. These parameters must be defined in the plug-in *describe* action using the *paramDefine* function. All parameters hold properties, though the exact set of properties on a parameter is dependant on its type. Parameters can have various properties overridden via a separate XML based resource file.

The image effect API provides a *render* routine. This routine is passed from the host to the plug-in when it requires rendering an output frame. As indicated previously, the plug-in could override this function to define the input source clip, the destination clip and, if necessary, the render mask. Image effect API supplies some actions to allow the plug-in to specify how it wishes to deal with its input clips and to set the properties to the output clip. Some of these properties are the depth and the components of a clip's pixel, the frame rate of the output clip, the variation between the frames of the output clip or how this output clip can be sampled at sub-frame times to produce different images.

Host may need to cast images from their native data format depending on the suitable format for the plug-in.

Stereographic postproduction plug-ins need to allow the effects to be manipulated in response to a user changing parameter value (for example, the distance between the two viewpoints or the conversion angle between them). OpenFX supports interactive rendering since the host has an interactive thread and a rendering thread. This allows an effect to be manipulated while having renders batches off to a background thread. Consequently, it is necessary to hold some degree of locking to prevent simultaneous reads or writes and some method to abort a backgrounded render.

Furthermore, stereographic plug-ins generate differing output images at different times as they have animating parameters. For this reason, the host must use a property provided by the image effect API (*kOfxImageEffectFrameVarying*) to indicate that the effect will need to be rendered at each frame, even if no input images or parameters are varying.

To develop stereoscopic plug-ins, it could be useful to explore the existing interfaces implemented over the image effects API in order to use or expand them to take advantage on the new 3D functionalities they add to the simple image API.

Stereoscopic postproduction plug-ins

Based on the previous technical specifications, it is possible to specify import and export stereoscopic plug-ins for stereo video formats. These plug-ins should be included in the *generic context* as they must receive some different input parameters corresponding to the input view clips, as well as other parameters used to indicate the 3D format of the output clip. Besides this, all these plug-ins could be outlined in the same binary file being distinguished by its name.

For example, if we desire an interlaced output format, the plug-in should require two input clips. Then the plug-in would read the different images contained in the clips and operate with them separately in order to create an output clip in the Page Flipping scanning format.

It is also advisable to specify some postproduction plug-ins that helps to cover the actual necessities in this field. These desirable tools should operate

specifically with two different channels which would represent separately the human eyes.

A proposed plug-in could be a plug-in that synchronizes the views between the two eyes. Other postproduction plug-ins used to simulate the depth of field would adjust the convergence angle, the perspective, the geometry or the viewpoint distance between the two eyes' images. Or a plug-in to parameterize the screen size where the images are projected since this size influences directly on the distance between the projected images and, consequently, on the depth perception. It could be necessary to implement other tools to do a colour correction of each eye or to filter the input images of every eye to apply different effects like blur effect.

Just like before, depending on the stereographic format, these tools would have different number of input clips and other parameters used to indicate the modifiable attributes and the values that these attributes will take.

For example, a plug-in to adjust the convergence angle between the two eye images, should require an input value to specify the convergence grade and other inputs which determine the exact coordinates where the eyes are placed (the viewpoints). Then the plug-in should compute the conversion and generate the output clip according to it. Based on this conversion, the audience will experience a different 3D depth perception.

Another example could be a plug-in to compute a free multi-viewpoint. It would also need an arbitrary number of inputs. In this exact case, it is necessary to get a high number of input images of the same scene from different points of view so the plug-in must compute them to obtain the desired output clip.