Draft Documentation

Release 1.6.0

Thinkbox Software

May 19, 2016

CONTENTS

CHAPTER

ONE

OVERVIEW

Draft is a lightweight compositing and video processing tool designed to automate typical post-render tasks. It is implemented as a Python library, which exposes functionality for use in python scripts. Draft is designed to be tightly integrated with Deadline, but it can also be used as a standalone tool.

CHAPTER

TWO

GETTING STARTED

2.1 Installation and Licensing

2.1.1 Installation

The installers for Deadline include Draft, but we do release Draft builds independently of Deadline. You can always download and (re)install a newest build of Draft. All you should need to do is to extract the .zip file containing the newest build in your Deadline Repository under the draft folder. The slaves will automatically sync up new versions of Draft to their local working folders when needed.

2.1.2 Licensing

It is important to note that Draft requires a license separate from Deadline. Draft and Quick Draft are both licensed by the same and FlexLM `FEATURE draft` in your Thinkbox license file. Draft and Quick Draft are free for users with an active Deadline annual support and maintenance contract.

The majority of Draft features require *only* that a license be present. Actual checkout of licenses happens *only* while videos are being encoded or decoded. A Draft Pro license (`FEATURE draft-pro-codec`) is *only* required if you are encoding to the 3rd party Avid DNxHD codec.

2.2 Usage

In order to get you familiar with Draft, many code snippets can be found in the *Cookbook*. Additionally, Draft sample scripts can be found in the Draft install directory under the Samples folder.

2.2.1 Submitting Draft Jobs To Deadline

There are many ways to submit Draft jobs to Deadline. As always, you can simply submit a Draft job from within the Monitor from the Submit menu. In addition, we've also added a right-click job script to the Monitor, which will allow you to submit a Draft job based on an existing job. This will pull over output information from the original job, and fill in Draft parameters automatically where possible.

When submitting jobs to Deadline through many of our submitters, you have the option to have Deadline create a dependent Draft Job once the submitted job has finished rendering. You can either use the Quick Draft settings, which use a general Draft Template script that ships with Deadline, or you can specify a custom Template script.

For more information, consult the Deadline Documentation. Specifically, the section Draft in Application Plugins and Draft / Quick Draft in Event Plugins can be very useful.

2.2.2 Draft Python Library

Draft can be used outside of Deadline (it still requires a **draft** license). In order to `import Draft` successfully, you will need to ensure Python knows where to look for the Draft module, either by including its install directory in the PYTHONPATH environment variable, or by appending it to `os.sys.path` before you do the import. You will also need to set the MAGICK_CONFIGURE_PATH and LD_LIBRARY_PATH (or DYLD_LIBRARY_PATH) environment variables to point to your install directory; this is so that ImageMagick knows where to find its 'type.xml' file. If you are simply planning on using Draft through Deadline, however, you do not need to worry about this at all.

If using Deadline's Python, ensure you use these paths to instantiate dpython as they ensure the Deadline environment is setup correctly:

- **Windows**: `C:\Program Files\Thinkbox\DeadlineX\bin\dpython.exe`
- **Mac OS X**: `/Applications/Thinkbox/DeadlineX/Resources/dpython`
- **Linux**: `/opt/Thinkbox/DeadlineX/bin/dpython`

where X is the MAJOR version of Deadline.

Setting Environment Variables

All platforms need specific environment variables to be set for Draft Standalone (Python) to successfully `import Draft`:

- **Windows**:

```
PYTHONPATH=/path/to/Draft install directory/Windows/XXbit
MAGICK_CONFIGURE_PATH=/path/to/Draft install directory/Windows/XXbit

where XX is either 32 or 64.
```

- **Mac OS X**:

```
PYTHONPATH=/path/to/Draft install directory/Mac
MAGICK_CONFIGURE_PATH=/path/to/Draft install directory/Mac
DYLD_LIBRARY_PATH=/path/to/Draft install directory/Mac
```

- **Linux**:

```
PYTHONPATH=/path/to/Draft install directory/Linux/64bit
MAGICK_CONFIGURE_PATH=/path/to/Draft install directory/Linux/64bit
LD_LIBRARY_PATH=/path/to/Draft install directory/Linux/64bit
```

CHAPTER

THREE

CONCEPTS

3.1 Image

3.1.1 Overview

A `Draft.Image` holds an actual image, along with the image's width and height, its channel names and a *file channel map*.

3.1.2 Implementation Details

In Draft, the method `CreateImage()` creates a new image and the method `ReadFromFile()` reads an image from file. An image can be written to file using the method `WriteToFile()`.

The image's `width` and `height` are stored as properties. You can modify those properties using the method `Resize()`.

To get the image's channel names, you can use the method `GetChannelNames()`. You can also manipulate the image's channels using the methods `SetChannel()`, `HasChannel()`, `RemoveChannel()`, `RenameChannel()` and `SetToColor()`.

A file channel map is used to specify on write and retrieve on read the bit depth and type of the image channels. It is possible to access the file channel map using the methods `SetFileChannelMap()` and `GetFileChannelMap()`. For more details, see the section *Image File Channel Map* in Concepts.

Beside the bit depth and type of the image channels, other image saving settings can be specified using `Draft.ImageInfo`. Consult the section *ImageInfo* in Concepts for more information.

3.1.3 Relevant Cookbook Recipes

- *Creating an Image*
- *Resize an Image*
- *Write an Image to File with a Specific File Channel Map*

3.2 Image File Channel Map

3.2.1 Overview

A file channel map is associated with a `Draft.Image` and is used to specify on write and retrieve on read the bit depth and type of the image channels.

3.2.2 Implementation Details

A file channel map is represented using a dictionary. Each dictionary entry is made of two strings separated by a colon. The first string corresponds to the channel name, typically 'R', 'G', 'B' and 'A', and the second string corresponds to the channel bit depth. An 'ui' is appended to the bit depth string to specify that the channel represents an index and an 'f' to specify that the channel is of type float. Such a string represents a channel data type.

Here are some common valid channel data types: '8', '16', '16f', '32f' and '32ui'. When Draft creates an image or add a new channel to an existing image, a default file channel data type of '8' is assigned to the new channel(s).

A dictionary representing a file channel map can be passed to the function `SetFileChannelMap()` to set the file channel map of a Draft.Image. Note that the dictionary must contain one entry for each channel present in the Draft.Image and for those channels only. An error will be thrown otherwise.

A dictionary is returned by the function `GetFileChannelMap()` to retrieve the current file channel map of a Draft.Image.

Internally, Draft is still storing image channel data using 32 bit floats.

Valid Channel Data Types

It is important to understand that each file format only supports specific bit depth and type. Here's a list of the valid channel data types supported by Draft for common file formats:

File Format	Channel Data Type
BMP	8
CIN	10
DPX	8, 10, 12, 16
EXR	16f, 32f, 32ui
GIF	8
HDR	16
JPEG/JPG	8
PNG	8, 16
TGA	5*, 8
TIF/TIFF	8, 16

* Writing a TGA image file with a channel data type of '5' corresponds to ImageMagick's 16-bit TGA – 5*3 + 1.

While Draft supports specifying the file channel map for other file formats, correct results are not guaranteed.

3.2.3 Relevant Cookbook Recipes

- *Write an Image to File with a Specific File Channel Map*
- *Write an Image to File with a Maximum Bit Depth*

3.3 ImageInfo

3.3.1 Overview

A *Draft.ImageInfo* object is used to specify on write and retrieve on read the image's compression, quality, tile size and *Timecode*, when applicable.

3.3.2 Implementation Details

Besides the image's channel data type described in the section *Image File Channel Map* in Concepts, other image saving settings can be controlled. Unlike the file channel map, those saving settings are not associated with a *Draft.Image* but with a *Draft.ImageInfo*. An *ImageInfo* stores those saving settings using the four properties: *compression*, *quality*, *tileSize* and *timecode*.

To specify the saving settings, you can simply set those properties in an *ImageInfo* and pass that *ImageInfo* as an additional parameter in the function *WriteToFile()*.

To retrieve the saving settings, you can pass an *ImageInfo* as an additional parameter in the function *ReadFromFile()* and the fields in *ImageInfo* will be automatically populated. Those fields can then be easily queried.

Valid Compression Values

Valid values for compression are summarized in the following table.

File Format	Valid compression
EXR	'none', 'rle', 'zip', 'zips', 'piz', 'pxr24', 'b44', 'b44a', 'dwaa' and 'dwab'
GIF	'lzw'
HDR	'rle'
JPEG/JPG	'jpeg'
PNG	'zip'
TGA	'none' and 'rle'
TIF/TIFF	'none', 'jpeg', 'lzw', 'rle' and 'zip'

The property compression defaults to 'default' which corresponds to the file format default.

Valid quality values are integer value in the range [0..100]. The property quality defaults to None and is only valid for format EXR and JPEG/JPG.

Note that the mapping between Draft and OpenEXR DWA quality is done using an exponential regression to fit the three points proposed by Karl Rasche (creator of DWA). See: https://lists.nongnu.org/archive/html/openexr-devel/2014-08/msg00049.html

Valid tileSize values are tuples of size 2 of strictly positive integers. The property tileSize default to None which corresponds to non-tiled image and is only valid for the file format EXR.

Please, consult the section *Timecode* in Concepts for valid timecodes values.

3.3.3 Relevant Cookbook Recipes

- *Set Saving Settings in an Image File*
- *Embed a Timecode in an Image File*
- *Embed a Timecode in a Video File*

3.4 Timecode

3.4.1 Overview

A *Draft.Timecode* is used to embed on write and extract on read a timecode in a DPX or EXR image file or in a video file.

3.4.2 Implementation Details

A *Timecode* object stores the hours, the minutes, the seconds and the frame associated to a timecode as described in SMPTE standard. In addition, a flag indicating whether the *Timecode* object represents a non-drop frame or a drop frame timecode is stored.

A *Timecode* is created using a string with format hh:mm:ss:ff for non-drop frame timecode and hh:mm:ss;ff for drop frame timecode, where hh indicates the hours in the range [0..23], mm the minutes in the range [0..59], ss the seconds in the range [0..59] and ff the frame in the range [0..59].

In the case of a DPX or EXR image file, a *Draft.ImageInfo* can be used to embed or to extract a *Timecode*. In the case of a video file, the *Timecode* to be embedded can be specified when creating a *Draft.VideoEncoder* and a *Draft.VideoDecoder* has a *timecode* property that can be used to extract the embedded timecode.

3.4.3 Relevant Cookbook Recipes

- *Embed a Timecode in an Image File*
- *Embed a Timecode in a Video File*

3.5 Video

3.5.1 Overview

A *Draft.VideoEncoder* is used to create videos and a *Draft.VideoDecoder* is used to extract frames from videos.

3.5.2 Implementation Details

Video Encoder

When creating a *Draft.VideoEncoder* you can specify many parameters including the video's framerate, kbit rate and codec. For a complete list, please consult the constructor *VideoEncoder()*. Once your *Draft.VideoEncoder* is created, you can encode each frame using the method *EncodeNextFrame()*.

Supported Codecs

Draft supports multiple codec options. They include:

- MPEG4 (default)
- MJPEG

- DNxHD® (**requires a "Draft Codec Pack" license**)
- H264
- RAWVIDEO

Valid Avid DNxHD® Settings

It's important to remember that when you're encoding with Avid DNxHD® you must use a specific set of parameters. Here is a table of all the Avid DNxHD® settings that work with Draft.

width	height	fps	kbitRate
1920	1080	59.94	440000
1920	1080	59.94	290000
1920	1080	59.94	90000
1920	1080	50	365000 *
1920	1080	50	240000 *
1920	1080	50	75000 *
1920	1080	29.97	220000
1920	1080	29.97	145000
1920	1080	29.97	45000
1920	1080	25	185000 *
1920	1080	25	120000 *
1920	1080	25	36000 *
1920	1080	24	175000 *
1920	1080	24	115000 *
1920	1080	24	36000 *
1920	1080	23.976	175000
1920	1080	23.976	115000
1920	1080	23.976	36000
1280	720	59.94	220000
1280	720	59.94	145000
1280	720	50	180000
1280	720	50	120000
1280	720	29.97	110000
1280	720	29.97	75000
1280	720	25	90000
1280	720	25	60000
1280	720	23.976	90000
1280	720	23.976	60000

* Draft supports writing MXF files for these combinations only

Video Files Concatenation

It's possible to concatenate video files using the function *ConcatenateVideoFiles()* by providing a list of input files and an output file. Note that the input files must all share the same codec and all have the same file extension than the output file.

Video Decoder

When creating a *Draft.VideoDecoder* you must specify the filename of the video you want to decode. Once your *Draft.VideoDecoder* is created, you can decode each frame using the method *DecodeNextFrame()*.

3.5.3 Relevant Cookbook Recipes

- *Create a QuickTime Movie*
- *Concatenate Video Files*
- *Change the Encoding of a Movie Clip*
- *Split a Movie into Single Frames*

CHAPTER

FOUR

LIBRARY REFERENCE

4.1 Anchor

class Draft.**Anchor**

This enum contains the different values that can be provided to functions that require positional anchor arguments.

Center = Draft.Anchor.Center

East = Draft.Anchor.East

North = Draft.Anchor.North

NorthEast = Draft.Anchor.NorthEast

NorthWest = Draft.Anchor.NorthWest

South = Draft.Anchor.South

SouthEast = Draft.Anchor.SouthEast

SouthWest = Draft.Anchor.SouthWest

West = Draft.Anchor.West

Sample Code:

```
anchor = Draft.Anchor.NorthWest
```

Anchor replaces the now-deprecated PositionalGravity enumeration:

class Draft.**PositionalGravity**

Deprecated since version beta14: Please use *Draft.Anchor* instead.

This enum contains the different values that can be provided to functions that require positional gravity arguments.

CenterGravity = Draft.PositionalGravity.CenterGravity

EastGravity = Draft.PositionalGravity.EastGravity

NorthEastGravity = Draft.PositionalGravity.NorthEastGravity

NorthGravity = Draft.PositionalGravity.NorthGravity

NorthWestGravity = Draft.PositionalGravity.NorthWestGravity

SouthEastGravity = Draft.PositionalGravity.SouthEastGravity

SouthGravity = Draft.PositionalGravity.SouthGravity

SouthWestGravity = Draft.PositionalGravity.SouthWestGravity

WestGravity = Draft.PositionalGravity.WestGravity

4.2 AnnotationInfo

class Draft.**AnnotationInfo**((*object*)*arg1*) → None
A class that contains properties used for textual annotations.

__init__((object)arg1) -> None

BackgroundColor
A *Draft.ColorRGBA* value denoting the background color. (Defaults to transparent.)

Color
A *Draft.ColorRGBA* value denoting the color of the text. (Defaults to white.)

DrawShadow
A boolean value denoting whether or not to draw a shadow around the text. (Defaults to false.)

FontMetric
A *Draft.FontTypeMetric* object used to retrieve font and text properties for the annotation. Its values are set when the annotation is created.

FontType
A string value denoting the type of font to use. (Defaults to Adobe's Source Sans Pro.)

Padding
A decimal value denoting the padding around the text. (Defaults to 0.0.)

PointSize
An integer value denoting the size of the font. (Defaults to 32.)

ShadowColor
A *Draft.ColorRGBA* value denoting the color of the shadow. (Defaults to black.)

Sample Code:

```
textInfo = Draft.AnnotationInfo()
textInfo.PointSize = 24
textInfo.FontType = "Helvetica"
textInfo.Color = Draft.ColorRGBA( 0.75, 0.54, 0.975, 1.0 )
textInfo.BackgroundColor = Draft.ColorRGBA( 0.0, 0.0, 0.0, 0.0 )
```

4.3 ColorRGBA

class Draft.**ColorRGBA**((*object*)*arg1*) → None
This class represents a color, in RGBA format.

__init__((object)arg1) -> None

__init__((object)arg1, (float)R, (float)G, (float)B, (float)A) -> None

A
Decimal value representing the alpha component. Normally in the range from 0.0 to 1.0.

B
Decimal value representing the blue component. Normally in the range from 0.0 to 1.0.

G
Decimal value representing the green component. Normally in the range from 0.0 to 1.0.

R
Decimal value representing the red component. Normally in the range from 0.0 to 1.0.

Sample Code:

```
newColor = Draft.ColorRGBA( 0.75, 0.54, 0.975, 1.0 )
```

or:

```
newColor = Draft.ColorRGBA()
newColor.R = 0.75
newColor.G = 0.54
newColor.B = 0.975
newColor.A = 1.0
```

4.4 CompositeOperator

class Draft.**CompositeOperator**
This enum contains all the operators available when compositing images.

AddCompositeOp = Draft.CompositeOperator.AddCompositeOp

AtopCompositeOp = Draft.CompositeOperator.AtopCompositeOp

BumpmapCompositeOp = Draft.CompositeOperator.BumpmapCompositeOp

CopyBlueCompositeOp = Draft.CompositeOperator.CopyBlueCompositeOp

CopyCompositeOp = Draft.CompositeOperator.CopyCompositeOp

CopyGreenCompositeOp = Draft.CompositeOperator.CopyGreenCompositeOp

CopyOpacityCompositeOp = Draft.CompositeOperator.CopyOpacityCompositeOp

CopyRedCompositeOp = Draft.CompositeOperator.CopyRedCompositeOp

DifferenceCompositeOp = Draft.CompositeOperator.DifferenceCompositeOp

InCompositeOp = Draft.CompositeOperator.InCompositeOp

MinusCompositeOp = Draft.CompositeOperator.MinusCompositeOp

MultiplyCompositeOp = Draft.CompositeOperator.MultiplyCompositeOp

OutCompositeOp = Draft.CompositeOperator.OutCompositeOp

OverCompositeOp = Draft.CompositeOperator.OverCompositeOp

PlusCompositeOp = Draft.CompositeOperator.PlusCompositeOp

SubtractCompositeOp = Draft.CompositeOperator.SubtractCompositeOp

UndefinedCompositeOp = Draft.CompositeOperator.UndefinedCompositeOp

XorCompositeOp = Draft.CompositeOperator.XorCompositeOp

Sample Code:

```
compOperator = Draft.CompositeOperator.OverCompositeOp
```

4.5 FontTypeMetric

class Draft.**FontTypeMetric**(*(object)arg1*) → None
A class to retrieve font and text properties for annotations.

__init__((object)arg1) -> None

Ascent
The distance in pixels from the text baseline to the highest/upper grid coordinate used to place an outline point.

BaselineOffset
Offset in pixels from the bottom of the image to the baseline used to write the text. Use this to align text from different point sizes and fonts.

Descent
The distance in pixels from the baseline to the lowest grid coordinate used to place an outline point. Always a negative value.

MaxHorizontalAdvance
Maximum horizontal advance in pixels.

TextHeight
The height in pixels of the text written, this does not include any padding.

TextWidth
The width in pixels of the text written, this does not include any padding.

4.6 Image

class Draft.**Image**(*(object)arg1*) → None
The Draft.Image class contains all of Draft's image-related functionality. It contains two types of functions: Static functions, and Member functions. Static functions can be invoked without an instance (by calling Draft.Image.<function name>), whereas Member functions require to be invoked from an instance of a Draft.Image object (by calling <someImage>.<function name>).

The Static functions are used to create new instances of the Draft.Image class, whereas the Member functions are used to modify pre-existing instances of the Draft.Image. The sample code snippets should clarify this distinction function in case you are unsure.

static Anaglyph(*(Image)leftImage*, *(Image)rightImage*, *(str)anaglyphType*) → Image :
Returns an anaglyph of the specified type created from the two left/right input images.

Arguments:

leftImage A *Draft.Image* containing the left-eye image.

rightImage A *Draft.Image* containing the right-eye image.

anaglyphType A string value containing the anaglyph type; can be either "LSA" or "PS".

Usage:

```
anaglyphImage = Draft.Image.Anaglyph( leftEye, rightEye, "LSA" )
```

ApplyGamma(*(Image)self*, *(float)gamma*) → None :
Apply the specified gamma correction to the image.

New in version 1.1.

Arguments:

gamma A decimal value indicating the gamma that should be applied.

Usage:

```
someImage = Draft.Image.ReadFromFile( "//path/to/some/image/file.png" )
someImage.ApplyGamma( 2.2 )
```

Composite(*(Image)self*, *(Image)image*, *(float)left*, *(float)bottom*, *(CompositeOperator)operation*) → None :

Composites an image with the current image at the given location using the specified compositing operation.

Arguments:

image A *Draft.Image* that will be copied from.

left A float value that denotes how far from the left the composite operation should take place.

bottom A float value that denotes how far from the bottom the composite operation should take place.

operation A *Draft.CompositeOperator* enum value indicating the type of operation to perform.

Usage:

```
img1 = Draft.Image.CreateImage( 800, 600 )
img2 = Draft.Image.ReadFromFile( "//path/to/image.png" )
img1.Composite( img2, 0, 0.33, Draft.CompositeOperator.OverCompositeOp )
```

CompositeWithAnchor(*(Image)self*, *(Image)image*, *(Anchor)anchor*, *(CompositeOperator)operation*) → None :

Composites an image with the current image using the specified positional anchor to determine the location of the image being composited.

Arguments:

image A *Draft.Image* that will be copied from.

anchor A *Draft.Anchor* enum value used to determine the location on the current image where Image will be composited.

operation A *Draft.CompositeOperator* enum value indicating the type of operation to perform.

Usage:

```
img1 = Draft.Image.CreateImage( 800, 600 )
img2 = Draft.Image.ReadFromFile( "//path/to/image.png" )
compOp = Draft.CompositeOperator.OverCompositeOp
img1.CompositeWithAnchor( img2, Draft.Anchor.NorthWest, compOp )
```

CompositeWithGravity(*(Image)self*, *(Image)image*, *(PositionalGravity)gravity*, *(CompositeOperator)operation*) → None :

Deprecated since version beta14: Use *Draft.Image.CompositeWithAnchor()* instead.

Composites an image with the current image using the specified positional gravity to determine the location of the image being composited.

Arguments:

image A *Draft.Image* that will be copied from.

gravity A *Draft.PositionalGravity* enum value used to determine the location of the image being composited.

operation A *Draft.CompositeOperator* enum value indicating the type of operation to perform.

Usage:

```
img1 = Draft.Image.CreateImage( 800, 600 )
img2 = Draft.Image.ReadFromFile( "//path/to/image.png" )
compOp = Draft.CompositeOperator.OverCompositeOp
gravity = Draft.PositionalGravity.NorthWestGravity
img1.CompositeWithGravity( img2, gravity, compOp )
```

CompositeWithPositionAndAnchor (*(Image)self*, *(Image)image*, *(float)x*, *(float)y*, *(Anchor)anchor*, *(CompositeOperator)operation*) → None :

Composites an image with the current image at the given location using the specified compositing operation and positional anchor.

Arguments:

image A *Draft.Image* that will be copied from.

x A float value that denotes how far from the left (as a percentage of the width) to position the anchor for the composite operation.

y A float value that denotes how far from the bottom (as a percentage of the height) to position the anchor for the composite operation.

anchor A *Draft.Anchor* enum value used to determine the location of the image being composited. The anchor specifies which location of the image being composited will be anchored at the location specified by (x, y).

operation A *Draft.CompositeOperator* enum value indicating the type of operation to perform.

Usage:

```
img1 = Draft.Image.CreateImage( 800, 600 )
img2 = Draft.Image.ReadFromFile( "//path/to/image.png" )
compOp = Draft.CompositeOperator.OverCompositeOp
anchor = Draft.Anchor.NorthWest
img1.CompositeWithPositionAndAnchor( img2, 0, 0.66, anchor, compOp )
```

CompositeWithPositionAndGravity (*(Image)self*, *(Image)image*, *(float)x*, *(float)y*, *(PositionalGravity)gravity*, *(CompositeOperator)operation*) → None :

Deprecated since version beta14: Use *Draft.Image.CompositeWithPositionAndAnchor()* instead.

Composites an image with the current image at the given location using the specified compositing operation and positional gravity.

Arguments:

image A *Draft.Image* that will be copied from.

x A float value that denotes how far from the left (as a percentage of the width) to position the anchor for the composite operation.

y A float value that denotes how far from the bottom (as a percentage of the height) to position the anchor for the composite operation.

gravity A *Draft.PositionalGravity* enum value used to determine the location on the current image where Image will be composited.

operation A *Draft.CompositeOperator* enum value indicating the type of operation to perform.

Usage:

```
img1 = Draft.Image.CreateImage( 800, 600 )
img2 = Draft.Image.ReadFromFile( "//path/to/image.png" )
compOp = Draft.CompositeOperator.OverCompositeOp
gravity = Draft.PositionalGravity.NorthWestGravity
img1.CompositeWithPositionAndGravity( img2, 0, 0.66, gravity, compOp )
```

Copy (*(Image)self*, *(Image)image*[, *(int)left=0*[, *(int)bottom=0*[, *(object)channels=None*]]]) → None :

Copy the other image onto self, with the specified offset applied to the bottom left corner of image.

New in version 1.1.

Arguments:

image A Draft.Image that will be copied from.

left Optional. An integer number of pixels that denotes how far from the left the image should be offset. (Default is 0.)

bottom Optional. An integer number of pixels that denotes how far from the bottom the image should be offset. (Default is 0.)

channels Optional. A list of channels to copy from image to this image, or None. The channels must exist in both images. If None, then both images must have the same channels, and all channels are copied. (Default is all channels.)

Usage:

```
img1 = Draft.Image.ReadFromFile( "//path/to/some/image.png" )
img2 = Draft.Image.ReadFromFile( "//path/to/other/image.png" )
img1.Copy( img2, channels=['A'] )
```

static **CreateAnnotation** (*(unicode)text*, *(AnnotationInfo)textInfo*) → Image :

Returns a new image consisting of the specified text. The provided *Draft.AnnotationInfo* object describes the various text parameters. CreateAnnotation also sets the values in the AnnotationInfo object's *Draft.FontTypeMetric* property.

Arguments:

text A string value providing the contents of the annotation.

textInfo A *Draft.AnnotationInfo* value providing parameters describing how to draw the text.

Usage:

```
textInfo = Draft.AnnotationInfo()
textImage = Draft.Image.CreateAnnotation( "Annotation text.", textInfo )
```

Note: In order to prevent clipping of certain characters in certain fonts, we added a small amount of padding to the left and right edges of the image. The amount added is proportional to the font size, and can be computed using:

```
math.ceil( 0.16 * textInfo.PointSize )
```

---

static **CreateImage** (*(int)width*, *(int)height* [, *(list)channels=['R', 'G', 'B', 'A']*]) → Image :

Returns a new image of the specified size with the specified channels (RGBA channels by default).

Arguments:

width An integer value denoting the width of the image to create.

height An integer value denoting the height of the image to create.

channels Optional. A list of channels to create in the image. (Defaults to ['R', 'G', 'B', 'A'].)

Changed in version 1.1: Added the optional channel parameter.

Usage:

```
newImage = Draft.Image.CreateImage( 800, 600 )
```

Crop (*(Image)self*, *(int)left*, *(int)bottom*, *(int)right*, *(int)top*) → None :
Crops the image to the given bounds.

Arguments:

left An integer value denoting the left bound of the crop.

bottom An integer value denoting the bottom bound of the crop.

right An integer value denoting the right bound of the crop.

top An integer value denoting the top bound of the crop.

Usage:

```
someImage = Draft.Image.CreateImage( 800, 600 )
someImage.Crop( 100, 150, 200, 250 )
```

GetChannelNames (*(Image)self*) → list :
Get a list of all the channels in the image.

New in version 1.1.

Arguments: (none)

Usage:

```
image = Draft.Image.ReadFromFile( 'image.png' )
channelNames = image.GetChannelNames()
```

GetFileChannelMap (*(Image)self*) → dict :
Return a dictionary representing the image's file channel map. Each dictionary entry is made of two strings separated by a colon. The first string corresponds to the channel name, typically 'R', 'G', 'B' and 'A', and the second string corresponds to the channel bit depth. An 'ui' is appended to the bit depth string to specify that the channel represents an index, and an 'f' to specify that the channel is of type float.

New in version 1.5.

Arguments: (none)

Usage:

```
someImage = Draft.Image.ReadFromFile( "//path/to/some/image/file.exr" )
fileChannelMap = someImage.GetFileChannelMap()
```

HasChannel (*(Image)self*, *(str)channel*) → bool :
Determine whether a channel exists in the image. Returns True if the image has channel, and False otherwise.

New in version 1.1.

Arguments:

channel The name of the channel to check for.

Usage:

```
image = Draft.Image.ReadFromFile( 'image.png' )
if image.HasChannel( 'A' ):
    print 'image has an alpha channel'
else:
    print 'image does not have an alpha channel'
```

Premultiply (*(Image)self*) → None :
Premultiply the image's R, G, and B channels by the A (alpha) channel. The image is modified in-place.

Arguments: (none)

Usage:

```
someImage = Draft.Image.ReadFromFile( "//path/to/some/image/file.png" )
someImage.Premultiply()
```

static ReadFromFile (*(unicode)filename*[, *(ImageInfo)imageInfo=<Draft.ImageInfo object at 0x0000000004156A98>*]) → Image :
Returns a new image from a file on disk. Supports various image types determined by file extension.

Arguments:

filename A string value indicating where on the machine to find the image file.

imageInfo Optional. A *Draft.ImageInfo* that will be populated with properties from the image file.

Changed in version 1.1: Added the optional imageInfo parameter.

Usage:

```
imageFromFile = Draft.Image.ReadFromFile( "//path/to/image/file.png" )
```

RemoveChannel (*(Image)self*, *(str)channel*) → None :
Remove an existing channel from the image.

New in version 1.1.

Arguments:

channel The channel to set. Typical channels are 'R', 'G', 'B', and 'A'.

Usage:

```
someImage = Draft.Image.ReadFromFile( "//path/to/some/image/file.png" )
if someImage.HasChannel( 'A' ):
    someImage.RemoveChannel( 'A' )
```

RenameChannel (*(Image)self*, *(str)oldChannel*, *(str)newChannel*) → None :
Rename the existing specified channel in the image.

New in version 1.1.

Arguments:

oldChannel The name of the existing channel to copy from.

newChannel The name of the new channel to copy to. This channel must not exist in the image.

Usage:

```
image = Draft.Image.ReadFromFile( 'stereo.exr' )
image.RemoveChannel( 'R' )
image.RenameChannel( 'right.R', 'R' )
```

Resize (*(Image)self*, *(int)width*, *(int)height*[, *(str)type='fit'*[, *(str)border='transparent'*]]) → None :
Resizes the image to the given size.

Arguments:

width An integer value denoting the width to which the image will be re-sized.

height An integer value denoting the height to which the image will be re-sized.

type Optional. A string that specifies how the image data should be scaled to fit the new size. (Default is 'fit'.) Valid type values are:

'none' Don't scale the image data.

'width' Scale the image to fit the new width, without changing the aspect ratio.

'height' Scale the image to fit the new height, without changing the aspect ratio.

'fit' Scale the image as large as possible while staying inside the new image, without changing the aspect ratio.

'fill' Scale the image as small as possible while covering the new image, without changing the aspect ratio.

'distort' Scale the image to match the new width and height, changing the aspect ratio if necessary.

border Optional. A string that specifies how the border around a resized image should be set. (Default is 'transparent'.) Border argument doesn't do anything if the type is 'fill'. Valid border values are:

'transparent' Set the border to be transparent.

'stretch' Stretch the left, right, top and bottom edges of the image to the edge of the resized frame.

Usage:

```
someImage = Draft.Image.CreateImage( 800, 600 )
someImage.Resize( 1920, 1080 )
```

or:

```
someImage = Draft.Image.CreateImage( 800, 600 )
someImage.Resize( 1920, 1080, 'distort' )
```

or:

```
someImage = Draft.Image.CreateImage( 800, 600 )
someImage.Resize( 1920, 1080, 'fit', 'stretch' )
```

SetChannel (*(Image)self*, *(str)channel*, *(float)value*) → None :
Set the channel to the given value. If the channel does not already exist, then it will be created.

Arguments:

channel The channel to set. Typical channel names are 'R', 'G', 'B', and 'A'.

value The value to assign to the channel. Typically in the range from 0.0 to 1.0.

Usage:

```
someImage = Draft.Image.ReadFromFile( "//path/to/some/image/file.png" )
someImage.SetChannel( 'A', 1.0 )
```

SetFileChannelMap (*(Image)self*, *(dict)fileChannelMap*) → None :
Set the image's file channel map using a dictionary. A valid dictionary entry is made of two strings separated by a colon. The first string corresponds to the channel name, typically 'R', 'G', 'B' and 'A', and

the second string corresponds to the channel bit depth. Append an 'ui' to the bit depth string to specify that the channel represents an index, and an 'f' to specify that the channel is of type float.

New in version 1.5.

Arguments:

fileChannelMap A dictionary indicating the image's file channel map.

Usage:

```
someImage = Draft.Image.ReadFromFile( "//path/to/some/image/file.exr" )
fileChannelMap = { 'R':'16f', 'G':'16f', 'B':'16f', 'ID':'32ui' }
someImage.SetFileChannelMap( fileChannelMap )
```

SetToColor (*(Image)self*, *(ColorRGBA)color*) → None :
Set the image to the given color.

Arguments:

color A *Draft.ColorRGBA* indicating the color that the image should be set to.

Usage:

```
someImage = Draft.Image.CreateImage( 800, 600 )
someImage.SetToColor( Draft.ColorRGBA( 1.0, 0.0, 0.0, 1.0 ) )
```

ToBytes (*(Image)self*) → str :
Get an ARGB32 string representing the image.

New in version 1.5.

Arguments: (none)

Usage:

```
someImage = Draft.Image.ReadFromFile( "//path/to/some/image/file.png" )
imageString = someImage.ToBytes()
imageQT = QImage( imageString, width, height, QImage.Format_ARGB32 )
```

Unpremultiply (*(Image)self*) → None :
Reverse the effects of Premultiply: unpremultiply the image's R, G, and B channels by the A (alpha) channel. The image is modified in-place.

Arguments: (none)

Usage:

```
someImage = Draft.Image.ReadFromFile( "//path/to/some/image/file.png" )
someImage.Unpremultiply()
```

WriteToFile (*(Image)self*, *(unicode)filename*[, *(ImageInfo)imageInfo=<Draft.ImageInfo object at 0x00000000041569E8>*]) → None :
Writes the image to a file on disk. Supports various image types determined by file extension.

Arguments:

filename A string value indicating where to save the file to.

imageInfo Optional. A *Draft.ImageInfo* that will be populated with properties from the image file.

Changed in version 1.1: Added the optional imageInfo parameter.

Usage:

```
        someImage = Draft.Image.CreateImage( 800, 600 )
        someImage.WriteToFile( "//path/to/save/location.png" )
```

height
An integer value indicating the height of the image.

width
An integer value indicating the width of the image.

4.7 ImageInfo

class Draft.**ImageInfo**(*(object)arg1*) → None
A class that contains image file properties.

New in version 1.1.

__init__((object)arg1) -> None

compression
A string used to retrieve (on read) and set (on write) the compression of an image. Defaults to 'default' which corresponds to the file format default. Valid compression values are: 'default', 'none', 'jpeg', 'lzw', 'rle', 'zip', 'zips', 'piz', 'pxr24', 'b44', 'b44a', 'dwaa' and 'dwab'.

New in version 1.4.

quality
An integer value in the range [0..100] used to retrieve (on read) and set (on write) the quality of an image. Defaults to None which corresponds to the file format default.

New in version 1.4.

tileSize
A tuple used to retrieve (on read) and set (on write) the width and height of the tiles in an image. Defaults to None for non-tiled images.

timecode
A *Draft.Timecode* object used to retrieve (on read) and set (on write) the timecode associated to an image. Defaults to None.

New in version 1.5.

Usage:

To determine whether an EXR file is tiled:

```
imageInfo = Draft.ImageInfo()
image = Draft.Image.ReadFromFile( '//path/to/test.exr', imageInfo=imageInfo )
if imageInfo.tileSize is None:
        print "Image is not tiled"
else:
        print "Image is tiled"
```

To write a tiled EXR file:

```
imageInfo = Draft.ImageInfo()
imageInfo.tileSize = ( 32, 32 )

image = Draft.Image.CreateImage( 1920, 1080 )
image.WriteToFile( '//path/to/out.exr', imageInfo=imageInfo )
```

To preserve an EXR file's tile or scanline settings:

```
imageInfo = Draft.ImageInfo()
image = Draft.Image.ReadFromFile( '//path/to/in.exr', imageInfo=imageInfo )
image.ApplyGamma( 1.8 )
image.WriteToFile( '//path/to/out.exr', imageInfo=imageInfo )
```

To write a DWAA compressed EXR file with quality set to 75:

```
imageInfo = Draft.ImageInfo()
imageInfo.compression = 'dwaa'
imageInfo.quality = 75

image = Draft.Image.ReadFromFile( '//path/to/in.exr' )
image.WriteToFile( '//path/to/out.exr', imageInfo=imageInfo )
```

To embed a timecode in an image:

```
timecode = Draft.Timecode( "09:10:11:12" )
imageInfo = Draft.ImageInfo()
imageInfo.timecode = timecode

image = Draft.Image.ReadFromFile( '//path/to/in.exr' )
image.WriteToFile( '//path/to/out.exr', imageInfo=imageInfo )
```

4.8 LibraryInfo

class Draft.**LibraryInfo**(*(object)arg1*) → None

These functions provide information about the Draft library itself.

static About() → str :

Returns a string containing information about Draft, and its contributors.

Usage:

```
print Draft.LibraryInfo.About()
```

static Description() → str :

Returns a string containing a brief description of Draft.

Usage:

```
print Draft.LibraryInfo.Description()
```

static Revision() → str :

Returns a string containing the revision of the Draft library being used.

Usage:

```
print Draft.LibraryInfo.Revision()
```

static Version() → str :

Returns a string containing the version of the Draft library being used.

Usage:

```
print Draft.LibraryInfo.Version()
```

4.9 LUT

class Draft.**LUT**

The Draft.LUT class contains Draft's methods for working with Color Look-Up Tables (LUT). It contains two types of functions: Static functions, and Member functions. Static functions can be invoked without an instance (by calling Draft.LUT.<function name>), whereas Member functions must be invoked from an instance of a Draft.LUT object (by calling <someLUT>.<function name>).

The Static functions are used to create new instances of the Draft.LUT class, whereas the Member functions are used to work with existing instances of the Draft.LUT. The sample code snippets should clarify this distinction function in case you are unsure.

Raises an exception This class cannot be instantiated from Python

Apply (*(LUT)arg1*, *(Image)image*) → None :

Transform a *Draft.Image*'s color using this LUT.

Arguments:

image The image to transform. The image will be transformed in-place.

Usage:

```
image = Draft.Image.ReadFromFile( '//path/to/some/image/file.png' )
lut = Draft.LUT.CreateSRGB()
lut.Apply( image )
```

static ClearOCIOCaches () → None :

Flush OCIO's cached contents of LUTs on disk, intermediate results, etc.

New in version 1.2.

Arguments: (none)

Usage:

```
Draft.LUT.ClearOCIOCaches()
```

Note: Calling this is normally not necessary.

static CreateASCCDL (*(object)slope*, *(object)offset*, *(object)power*[, *(float)saturation=1*]) → LUT :

Return a new linear-to-ASC-CDL LUT.

New in version 1.2.

Arguments:

slope Three float values greater than or equal to 0 indicating the slope correction for the R, G, B channels. For example: [1.2, 0, 3.4].

offset Three float values indicating the offset correction for the R, G, B channels. For example: [-0.6, 0.2, 0].

power Three float values strictly greater than 0 indicating the power correction for the R, G, B channels. For example: [1.1, 2, 3].

saturation Optional. A float value greater than or equal to 0 indicating the saturation to apply. Defaults to 1.

Usage:

```
lut = Draft.LUT.CreateASCCDL( [1, 0, 3.4], [-0.6, 0.2, 0], [1.1, 2, 3], 1 )
```

Note: Draft does not compute the inverse of an ASC CDL LUT, and so calling Inverse() on one will throw a runtime error.

---

static `CreateAlexaV3LogC` () → LUT :
Return a new linear-to-ALEXA V3 Log C LUT.

Arguments: (none)

Usage:

```
alexaLut = Draft.LUT.CreateAlexaV3LogC()
```

static `CreateCineon` ([*(float)blackLevel=95*[, *(float)whiteLevel=685*]]) → LUT :
Return a new linear-to-Cineon LUT.

Arguments:

blackLevel Optional. An integer in the range [0..1023] indicating the black level. (Default is 95.)

whiteLevel Optional. An integer in the range [0..1023] indicating the white level. (Default is 685.)

Usage:

```
cineonLut = Draft.LUT.CreateCineon()
```

static `CreateGamma` (*(float)gamma*) → LUT :
Return a new linear-to-Gamma LUT.

Arguments:

gamma A float value indicating the gamma to apply.

Usage:

```
gammaLut = Draft.LUT.CreateGamma( 2.2 )
```

static `CreateOCIOProcessor` (*(str)colorSpaceIn*, *(str)colorSpaceOut*) → LUT :
Return a new OCIO LUT for converting images from colorSpaceIn to colorSpaceOut. (Note: Can also use role alias names from config.ocio.)

New in version 1.2.

Arguments:

colorSpaceIn the colorspace the input images will be in.

colorSpaceOut the desired colorspace for the processed images.

Usage:

```
ocioLUT = Draft.LUT.CreateOCIOProcessor( 'linear', 'Cineon' )
```

static `CreateOCIOProcessorFromFile` (*(str)filename*) → LUT :
Return a new OCIO LUT based on the specified file.

New in version 1.2.

Arguments:

filename name of the LUT file (including the path, either absolute, or relative to the search paths in config.ocio).

Usage:

```
lut = Draft.LUT.CreateOCIOProcessorFromFile( '//path/to/LUTfile.ext' )
```

Note: For a list of supported LUT file types, see: http://opencolorio.org/FAQ.html#what-lut-formats-are-supported.

static `CreateRec709` () → LUT :
Return a new linear-to-Rec. 709 LUT.

Arguments: (none)

Usage:

```
rec709Lut = Draft.LUT.CreateRec709()
```

static `CreateSRGB` () → LUT :
Return a new linear-to-sRGB LUT.

Arguments: (none)

Usage:

```
srgbLut = Draft.LUT.CreateSRGB()
```

`Inverse` (*(LUT)arg1*) → LUT :
Return a new LUT that performs the inverse operation of this LUT.

Arguments: (none)

Usage:

```
cineonLut = Draft.LUT.CreateCineon()
inverseCineonLut = cineonLut.Inverse()
```

static `SetOCIOConfig` ([*(unicode)path=''*]) → None :
Initialize OCIO to use the config.ocio file specified by path. If no path is supplied, Draft checks the OCIO environment variable, then configurations distributed with Draft, and finally searches PATH for a directory containing a config.ocio file.

New in version 1.2.

Arguments:

path path/to/config.ocio (optional)

Usage:

```
Draft.LUT.SetOCIOConfig( '//path/to/some/config.ocio' )
```

4.10 `Timecode`

class `Draft.`**`Timecode`** (*(object)arg1*, *(str)timecodeString*) → None :
Create a Timecode object to store a timecode as described in SMPTE standard.

Constructor Arguments:

timecode A string value with format hh:mm:ss:ff for non-drop frame timecode and hh:mm:ss;ff for drop frame timecode, where hh indicates the hours in the range [0..23], mm the minutes in the range [0..59], ss the seconds in the range [0..59] and ff the frame in the range [0..59].

Usage:

```
timecode = Draft.Timecode( "12:40:10:15" )
```

New in version 1.5.

4.11 TileAssembler

class Draft.**TileAssembler**((*object*)*arg1*) → None

The Draft.TileAssembler class provides the ability to assemble images or portions of images, called tiles, into a single final image that will be written to a file on disk. Each tile is represented by a *Draft.Image* created from a filename and corresponds to an image file on disk. The image files on disk associated with the tiles must not be modified during the assembly process; doing so might lead to unexpected behavior.

New in version 1.3.

__init__((object)arg1) -> None

AddTile((*TileAssembler*)*self*, (*Image*)*image*, (*int*)*left*, (*int*)*bottom*) → None :

Adds an image to be included in the assembled final image.

Arguments:

image A *Draft.Image* created from a filename.

left An integer number of pixels that denotes how far from the left the image should be offset.

bottom An integer number of pixels that denotes how far from the bottom the image should be offset.

Usage:

```
tileAssembler = Draft.TileAssembler()
imageFromFile = Draft.Image.ReadFromFile( "//path/to/some/image.exr" )
tileAssembler.AddTile( imageFromFile, 160, 120 )
```

AssembleToFile((*TileAssembler*)*self*, (*unicode*)*filename*[, (*ImageInfo*)*imageInfo=<Draft.ImageInfo object at 0x0000000004156B48>*]) → None :

Assembles the images added with the function *Draft.TileAssembler.AddTile()* to one single image which is written to a file on disk.

Arguments:

filename A string value indicating where to save the file to.

imageInfo Optional. A *Draft.ImageInfo* that determines saving properties. (Default is 'None'.)

Usage:

```
tileAssembler = Draft.TileAssembler()
imageFromFile = Draft.Image.ReadFromFile( "//path/to/some/image.exr" )
tileAssembler.AddTile( imageFromFile, 160, 120 )
tileAssembler.AssembleToFile( "//path/to/final/image.exr" )
```

SetChannels((*TileAssembler*)*self*, (*object*)*channels*) → None :

Sets the channels of the final image.

Arguments:

channels A list of (string) channel names to be included in the final image.

Usage:

```
tileAssembler = Draft.TileAssembler()
tileAssembler.SetChannels( ['R', 'G', 'B', 'A'] )
```

SetSize (*(TileAssembler)self*, *(int)width*, *(int)height*) → None :
Sets the size of the final image.

Arguments:

width An integer value indicating the width of the final image.

height An integer value indicating the height of the final image.

Usage:

```
tileAssembler = Draft.TileAssembler()
tileAssembler.SetSize( 640, 480 )
```

4.12 Video

Draft.**QTFastStart** (*(unicode)inputFile*, *(unicode)outputFile*) → int :
Rearranges the atoms inside the input QT file to enable playback without first loading the entire file.

Arguments:

inputFile The filename of the input .mov file.

outputFile The filename of the output .mov file.

Usage:

```
Draft.QTFastStart( "/path/to/input.mov", "/path/to/output.mov" )
```

Draft.**ConcatenateVideoFiles** (*(list)inputFiles*, *(unicode)outputFile*) → None :
Concatenate a list of video files.

New in version 1.5.

Arguments:

inputFiles A list of string values indicating paths to the video files to concatenate.

outputFile A string value indicating the path to the concatenated video to create.

Usage:

```
Draft.ConcatenateVideoFiles( [ "/path/to/input1.mov", "/path/to/input2.mov" ],
    "/path/to/output.mov" )
```

4.12.1 VideoEncoder

class Draft.**VideoEncoder**

object __init__(tuple args, dict kwds) : __init__((VideoEncoder) arg1, (str)filename [, (object)fps=24 [, (int)width=640 [, (int)height=480 [, (int)kbitRate=None [, (str)codec='MJPEG' [, (str)audioFilename='' [, (int)audioDelay=0]]]]]]) -> None[, (TimeCode)timecode=None

Create a VideoEncoder.

Constructor Arguments:

filename A string value indicating where the video file should be saved.

fps Optional. An integer, float, or Fraction value indicating the framerate to use. (Default is 24.)

width Optional. An integer value indicating the width to use. (Default is 640.)

height Optional. An integer value indicating the height to use. (Default is 480.)

kbitRate Optional. An integer value indicating the kbit rate to use. (Default is None, which corresponds to quality = 85.) Only one of kbitRate or quality can be specified.

codec A string value indicating the Codec to use. (Default is 'MJPEG'.) Valid codec values are: 'MPEG4', 'MJPEG', 'H264', 'DNXHD', 'RAWVIDEO', and 'VP8'.

audioFilename A string value indicating the name of an audio file, or a video file with an audio track, to include in the video. This can be an empty string for no audio file. (Default is ''.)

audioDelay Optional. An integer value indicating the delay to apply to the audio file, measured in frames. (Default is 0.)

timecode Optional. A *Draft.Timecode* object used to set the timecode that will be embedded in the video. (Default is None.)

Named arguments:

quality An integer in the range [0..100] indicating the video encoding quality to use. Greater values correspond to higher quality. Only one of quality or kbitRate can be specified.

Usage:

```
enc = Draft.VideoEncoder( "//path/to/video/save.mov" )
```

or, to specify the frame rate and frame size:

```
enc = Draft.VideoEncoder( "//path/to/video/save.mov", 24, 800, 600 )
```

or, to include an audio file:

```
enc = Draft.VideoEncoder( "//path/to/video/save.mov",
                          audioFilename='//path/to/audio/load.wav' )
```

or, to use a specific quality and codec:

```
enc = Draft.VideoEncoder( "//path/to/video/save.mov", quality=80, codec='H264' )
```

or, to use a Fraction FPS:

```
from fractions import Fraction
fps = Fraction(30000, 1001)
enc = Draft.VideoEncoder( "//path/to/video.mov", fps, 800, 600, 5120, "MJPEG" )
```

EncodeNextFrame (*(VideoEncoder)arg1*, *(Image)image*) → None :
EncodeNextFrame(image)

Encodes a given *Draft.Image* as the next frame in the video.

Arguments:

image The image to encode into the next frame of the video.

Usage:

```
defaultEncoder = Draft.VideoEncoder( "//path/to/video/save.mov" )
defaultEncoder.EncodeNextFrame( Draft.Image.CreateImage( 800, 600 ) )
```

FinalizeEncoding (*(VideoEncoder)arg1*) → None :
Finalizes the encoding process, completing the video.Arguments: (none)

Usage:

```
enc = Draft.VideoEncoder( "//path/to/video.mov" )
# encode some frames here...
enc.FinalizeEncoding()
```

4.12.2 VideoDecoder

class Draft.**VideoDecoder** (*(object)arg1*, *(unicode)filename*) → None :
Create a VideoDecoder to decode the specified file.

Constructor Arguments:

filename A string value indicating the path of the video to decode.

Usage:

```
decoder = Draft.VideoEncoder( "//path/to/video/load.mov" )
```

DecodeFrame (*(VideoDecoder)arg1*, *(long)frameNumber*, *(Image)image*) → bool :
Decodes a specified frame from the video and returns it by reference through the image argument. This function returns a boolean to indicate whether or not the decode succeeded.

Arguments:

frameNumber An integer indicating which frame should be decoded.

image A *Draft.Image* object used to return the decoded frame.

Usage:

```
decoder = Draft.VideoDecoder( "//path/to/video/file.mov" )
frameImage = Draft.Image.CreateImage( 800, 600 )
if decoder.DecodeFrame( 100, frameImage ):
    # process frameImage here...
```

DecodeNextFrame (*(VideoDecoder)arg1*, *(Image)image*) → bool :
Decodes a frame from the video and returns it by reference through the image argument. This function returns a boolean to indicate whether or not the decode succeeded.

Arguments:

image A *Draft.Image* object used to return the decoded frame.

Usage:

```
decoder = Draft.VideoDecoder( "//path/to/video/file.mov" )
frameImage = Draft.Image.CreateImage( 800, 600 )
while( decoder.DecodeNextFrame( frameImage ) ):
    # process the decoded frames here...
```

fps
The average frame rate of the video.

height
The height of a video frame.

timecode
The timecode embedded in the video.

width
The width of a video frame.

CHAPTER

FIVE

COOKBOOK

5.1 Introduction

We have compiled a series of commonly used code recipes in Draft into this cookbook. First off, if you aren't familiar, a code recipe is a snippet of code that captures a common case. The goal is so that you don't have to remember exactly how to do something, like how to setup a certain type of video encoder, and instead have a place to quickly refer to when you need to cook something up!

You can use this cookbook both as way to learn new techniques and functions in Draft, as well as a handy reference whenever you need it. While reading it from start to finish would be helpful, the intention is that you use it by jumping around the table of contents based on what you are interested in.

We will be adding new recipes in the future and if you would like to suggest new additions. please post on our forums.

These recipies make the assumption that you are familiar with the Python programming language, so we won't be taking too many pains to explain the syntatic details of the recipes. The discussions will focus primarily on what Draft is doing and what other options might be available to you.

5.1.1 Conventions Used

Here are the conventions we use throughout this cookbook.

Each recipe is in response to a given **Problem**. This can be something fairly simple like *Creating an Image* to something much more complex like Compositing images using Anchors.

The **Solution** is the code snippet that solves the stated problem. The recipe will be formatted with Python highlights to make it easier to look at. Copy the recipe code into your own scripts to use it.

Finally, each recipe is accompanied by a **Discussion** of the solution. The discussion is not required reading, but often includes further details and offers helpful hints when needing to solve more complex problems. Where appropriate, sometimes a **See Also** section is included to link to other relevant recipes or documentation.

When we refer to code within the discussion text, `quoted code is formatted like this` to help make it stand out. When we include code lines, they will appear on their own line and be in a different font.

For example, the Draft library is imported into a Python script in the standard manner:

```
import Draft
```

The module `Draft` contains several important objects such as *`Draft.Image`*.

5.2 Basic Operations

The basic operations cover the basics of Draft and what you will see in most Draft scripts. Things like creating an image and doing some simple text annotations are covered here, as well as how to setup a video encoder to create a video file.

5.2.1 Creating an Image

Problem

You want to create a new blank image with a width of 640 pixels and a height of 480 pixels.

Solution

```
import Draft

img = Draft.Image.CreateImage( 640, 480 )

# save the image for later use
img.WriteToFile( 'blank image.jpg' )
```

Discussion

The first line (`import Draft`) imports the functionality of Draft. The second line asks Draft to create an image with the given width (640 pixels) and height (480 pixels), and saves it in the variable which we have chosen to call `img`. Note that we can also save the width and height values in variables, and then pass the variables to *`CreateImage()`*:

```
newWidth = 1024
newHeight = 768

newImage = Draft.Image.CreateImage( newWidth, newHeight )
```

If, instead of starting with a blank image, you want to start with an image you have saved, you can use Draft's *`ReadFromFile()`* method instead of *`CreateImage()`*. Instead of specifying a width and height, we now specify the filename. If the image file is not in the current directory, we must include the path to the image as part of the filename. For example:

```
myImg = Draft.Image.ReadFromFile( r'X:\project\shot\Patches with ball.jpg' )
```

Note that we have used quotes around the filename. Quotation marks are used to indicate the start and end of non-numeric data literals, called strings (short for "string of characters"). We can use either single quotes (') or double quotes ("), as long as the quotes at the beginning and end match. We have also used the letter `r` before the first quote to indicate that the backslashes and characters following the backslashes are to be treated as separate characters, not as an escape sequence. (Escape sequences are used to include special characters in strings.)

The final line of the solution saves our image to a file, in this case, 'blank image.jpg' in the current working directory. (Again, if you wish the image to be saved elsewhere, you must specify the path as part of the directory.) Draft automatically detects the format for the image from the filename extension.

5.2.2 Crop an Image

Problem

You wish to crop an image, 'Patches with ball.jpg', coming in 100 pixels on the left, 80 on the bottom, 200 on the right, and 150 on the top.

Solution

```
import Draft

img = Draft.Image.ReadFromFile( 'Patches with ball.jpg' )
img.Crop( 100, 80, img.width - 200, img.height - 150 )
img.WriteToFile( 'Patches with ball (cropped).jpg' )
```

Discussion

Here we've decided that we want to remove parts of the image to improve the composition of the remaining image. The parameters to *Crop()* specify the left edge, bottom edge, right edge, and top edge, each measured in pixels from either the left or bottom edges of the uncropped image. In this example, we're removing 100 pixels from the left edge, 80 pixels from the bottom edge, 200 pixels from the right edge, and 150 pixels from the top edge.

Suppose you know how much you want to crop in inches (or cm), rather than how many pixels. You can convert this number from inches to pixels by using the image resolution. If we wanted to crop, say 0.75 inches off of each edge of an image whose resolution is 300 pixels per inch, then we can compute the number of pixels to crop by multiplying 0.75 inches by 300 ppi to get 225 pixels:

```
inches = 0.75
ppi = 300
cropAmt = inches * ppi

img.Crop( cropAmt, cropAmt, img.width - cropAmt, img.height - cropAmt )
```

See Also

For more information on the import statement, *ReadFromFile()*, and *WriteToFile()*, see the *Creating an Image* section of this Cookbook.

5.2.3 Composite Two Images

Problem

You have two images, Patches.exr and Ball.exr, and you wish to composite Ball.exr over Patches.exr, so that the ball appears in the bottom right corner of the image.

Solution

```
import Draft

img = Draft.Image.ReadFromFile( 'Patches.exr' )
ballImg = Draft.Image.ReadFromFile( 'Ball.exr' )
```

```
anchor = Draft.Anchor.SouthEast
compOp = Draft.CompositeOperator.OverCompositeOp
img.CompositeWithAnchor( ballImg, anchor, compOp )

img.WriteToFile( 'Patches with ball.exr' )
```

Discussion

We have three choices of composite methods, which differ in how we control the positioning of the top image: *Composite()*, *CompositeWithAnchor()*, and *CompositeWithPositionAndAnchor()*.

The first method, *Composite()*, takes four parameters: the second image that will be composited with the current image, two floating point values that specify the horizontal and vertical offset of the second image relative to the first image, and the composite operator, which specifies how the pixels of the two images will be combined. The coordinates of the bottom left corner of the second image are given as a fraction of the width and height of the current image, and specify the distance from the bottom left corner of the current image. For example, if we wanted the second image's bottom left corner to be positioned at 10% of the height of the first image, and 30% of the width, we would use:

```
xPos = 0.3
yPos = 0.1
img.Composite( ballImg, xPos, yPos, Draft.CompositeOperator.OverCompositeOp )
```

The second method, *CompositeWithAnchor()*, is the method we have chosen to use in our example. *CompositeWithAnchor()* takes three parameters; the first and last parameters (image and operation) are the same as *Composite()*, but now the second parameter, which specifies how to position the second image, is a named anchor constant. This named anchor constant specifies which coordinates of the two images are aligned. For example, *Draft.Anchor.North* specifies that the top centre of both images are aligned, whereas *Draft.Anchor.SouthEast* specifies that the bottom right corners of the images are aligned. The named Anchor coordinates correspond to the eight cardinal and intercardinal directions (*NorthWest*, *North*, *NorthEast*, *East*, *SouthEast*, *South*, *SouthWest*, *West*), plus *Center*.

The third method, *CompositeWithPositionAndAnchor()*, uses both position and anchor parameters, as its name implies. The position specifies a location on the current image, again as coordinates from the bottom left corner measured as a percentage of width and height. The anchor specifies which location on the second image will be aligned with the position on the first image. For example, if we want the centre of the second image to be located at the horizontal centre of the first image, 25% up from the bottom, we would use:

```
xPos = 0.5
yPos = 0.25
anchor = Draft.Anchor.Center
compOp = Draft.CompositeOperator.OverCompositeOp
img.CompositeWithPositionAndAnchor( ballImg, xPos, yPos, anchor, compOp )
```

Once we have chosen our method of positioning, there are many choices for how we combine the pixels of one image with the pixels of another. The most common composite operation is the "over" operation, which we select by using the over operator constant, *Draft.CompositeOperator.OverCompositeOp*. With the over operation, opaque pixels from `ballImg` will replace the corresponding pixels in `img`, and transparent pixels will leave the underlying pixels in `img` unchanged. For semi-transparent pixels in `ballImg`, we get a blend of the foreground and background that depends on the alpha value of the `ballImg` pixel.

Note that Draft assumes all images are not premultiplied; in other words, the color channels have not been multiplied by the alpha channel. Draft does any premultiplication necessary for the composite operation, and also returns an image that is unpremultiplied. If your images are already premultiplied, you can use `img.Unpremultiply()` *to unpremultiply the image.*

See Also

For more information on the `import` statement, *`ReadFromFile()`*, and *`WriteToFile()`*, see the *Creating an Image* section of this Cookbook.

For more information on the various composite operators, see the *Library Reference*.

5.2.4 Convert an Image From One Format To Another

Problem

You have an image, "Patches with ball.jpg", stored as a jpeg, but wish to have it in exr format.

Solution

```
import Draft

img = Draft.Image.ReadFromFile( 'Patches with ball.jpg' )
img.WriteToFile( 'Patches with ball.exr' )
```

Discussion

Converting an image from one format to another is as simple as reading in the image and saving it in a new file with the appropriate file extension. Draft automatically converts the image based on the file extension specified. Draft supports the following image extensions: .jpg (or .jpeg), .png, .tif (or .tiff), .exr, .dpx, .gif, .bmp, .hdr, .tga.

See Also

For more information on the `import` statement, *`ReadFromFile()`*, and *`WriteToFile()`*, see the *Creating an Image* section of this Cookbook.

5.2.5 Create Text Annotation

Problem

You wish to add some text, "My ball!!!", to your image, 'Patches with ball.jpg'.

Solution

```
import Draft

img = Draft.Image.ReadFromFile( 'Patches with ball.jpg' )

textInfo = Draft.AnnotationInfo()
textImage = Draft.Image.CreateAnnotation( "My ball!!!", textInfo )

anchor = Draft.Anchor.South
compOp = Draft.CompositeOperator.OverCompositeOp
img.CompositeWithPositionAndAnchor( textImage, 0.5, 0.1, anchor, compOp )
```

```
img.WriteToFile( 'Patches with ball (annotated).jpg' )
```

Discussion

There are three steps to adding text in this solution: getting the annotation information, creating an image that contains the text, and adding the text image to the photo.

The annotation information stores the properties, such as font, point size, colour, and more. If we use the *AnnotationInfo* object as is, the text image will use the default values:

`PointSize`: 32

`FontType`: SourceSansPro-Regular

`Padding`: 0.0

`Color`: white

`BackgroundColor`: transparent

`DrawShadow`: false

`ShadowColor`: black

However, we can also customize any of the above settings.

PointSize:

The point size is measured in pixels. If your image resolution is 72 ppi, the default point size will give you text that is nearly half an inch tall. However, if your image resolution is, say, 300 ppi, the text will only be a tenth of an inch tall. To make sure the text is the height you want it, be sure to set the point size after creating the annotation info, but before using it to create the text image:

```
text_height_inches = 0.5
ppi = 300
textInfo.PointSize = int( text_height_inches * ppi )
```

Since `PointSize` expects an integer, we need to use the `int()` function to convert `text_height_inches * ppi` from floating point to an integer.

FontType:

The font type specifies both which font, and which style (bold, italic, etc) to use for the text. The default font is set to Adobe's Source Sans Pro (regular) font. However, you can use any font contained in your type.xml file, or any font not in type.xml as long as you specify the full path to the font file:

```
textInfo.FontType = 'Times-New-Roman-Bold'  # Using the name field value from type.xml
```

or:

```
textInfo.FontType = r'C:\Windows\Fonts\timesbi.ttf'  # Specifying location of glyphs
```

Troubleshooting: If you get an error message when trying to use the default font, make sure the MAGICK_FONT_PATH environment variable is set to the Draft subdirectory containing SourceSansPro-Regular.otf.

Padding:

Padding specifies how much empty space to leave around the text in the image that is created using *CreateAnnotation()*. The padding is multiplied by the point size to get the measurement in pixels, so a padding of 1.0 with the default point size gives 32 pixels of space around the text.

Color:

The colors of the text (Color), background (BackgroundColor), and shadow (ShadowColor) can be changed by creating a new *Draft.ColorRGBA* object and performing the appropriate assignment. For example, to make the text dark red:

```
textInfo.Color = Draft.ColorRGBA( 0.5, 0.0, 0.0, 1.0)
```

The four parameters to *Draft.ColorRGBA* are the values to use for red, green, blue, and alpha, in that order. Values for these channels are normally in the range 0.0 to 1.0.

DrawShadow:

A shadow will be drawn for the text if DrawShadow is set to true:

```
textInfo.DrawShadow = True
```

Adding some of our customizations to the simple solution, we get:

```
import Draft

img = Draft.Image.ReadFromFile( 'Patches with ball.jpg' )

textInfo = Draft.AnnotationInfo()
text_height_inches = 0.5
ppi = 300
textInfo.PointSize = int( text_height_inches * ppi )
textInfo.FontType = 'Times-New-Roman-Bold-Italic'       # Using 'name' from type.xml
textInfo.Color = Draft.ColorRGBA( 0.5, 0.0, 0.0, 1.0)
textInfo.DrawShadow = True

textImage = Draft.Image.CreateAnnotation( "My ball!!!", textInfo )
anchor = Draft.Anchor.South
compOp = Draft.CompositeOperator.OverCompositeOp
img.CompositeWithPositionAndAnchor( textImage, 0.5, 0.1, anchor, compOp )

img.WriteToFile( 'Patches with ball (annotated).jpg' )
```

After the annotation information is set how we want it, we can use it in the call to *CreateAnnotation()* to create an image containing the text we want, in this case, “My ball!!!”. *CreateAnnotation()* saves the annotation as an image, which we can then composite over our photo using one of the three composite methods: *Composite()*, *CompositeWithAnchor()*, and *CompositeWithPositionAndAnchor()*. In this case, we have chosen to use *CompositeWithPositionAndAnchor()*.

After the call to *CreateAnnotation()*, we have our photo in the variable img, and the text in the variable textImg. *CompositeWithPositionAndAnchor()* is an instance method, meaning that we call it on an *Image* instance, in this case, img. The photo in img will be modified, which is why we do not need to assign a return value to another variable. The second image, textImg, is passed as an argument to the method. The remaining arguments specify how we wish the second image to be composited with the first: 0.5 and 0.1 specify that we want the anchor point to be located half way across img, and 10% up from the bottom; *Draft.Anchor.South* indicates that the anchor

point corresponds to the bottom center of `textImg`, and *`Draft.CompositeOperator.OverCompositeOp`* indicates to place `textImg` over `img`. Since the background of `textImg` is transparent by default, we will be able to see our photograph wherever there is neither text nor shadow.

See Also

For more information on the other two composite methods and the various composite operations, see the section *Composite Two Images*.

For more information on the `import` statement, *`ReadFromFile()`*, and *`WriteToFile()`*, see the *Creating an Image* section of this Cookbook.

5.2.6 Resize an Image

Problem

You want to resize 'Patches with ball.jpg' to 640x480.

Solution

```
import Draft

img = Draft.Image.ReadFromFile( 'Patches with ball.jpg' )
img.Resize( 640, 480 )
img.WriteToFile( 'Patches with ball 640x480.jpg' )
```

Discussion

In addition to the width and height, Resize has two optional parameters, `type` and `border`. Type specifies how the image should be scaled to fit the new size.The default value is `'fit'`, which scales the image as large as possible, keeping the aspect ratio locked, while staying inside the bounds of the new image size. If the aspect ratio of the new size does not match the aspect ratio of the old size, the unfilled portion of the resized image will be evenly distributed on both sides of the image, either top and bottom, or left and right. Most options for type preserve the aspect ratio of the original image. These aspect-ratio-preserving types include: `'none'` (don't scale the image at all), `'width'` (scale the image to fit the new width), `'height'` (scale the image to fit the new height), `'fit'` (already described), and `'fill'` (scale the image as small as possible while covering the new image size). One option does not preserve the aspect ratio: `'distort'` (scale the image to match the new width and height, changing the aspect ratio as necessary).

We can use the type parameter to letterbox or pillarbox the original image inside the new dimensions. To letterbox a shorter image within an image with a taller aspect ratio, choose `type='width'`:

```
img.Resize( 640, 480, type='width' )
```

To pillarbox a narrower image within an image with a wider aspect ratio, choose `type='height'`:

```
img.Resize( 1280, 720, type='height' )
```

Except for `'distort'` and `'fill'`, the type options could leave portions of the new image with no data. We specify how these areas are to be filled using the optional `border` parameter. There are two border options, `'transparent'` (default), and `'stretch'`. Choosing (or defaulting to) `'transparent'` sets the border pixels to black with an alpha of zero, while 'stretch' copies the pixels at the edges of the original image, and uses those values to fill the border. For example, to resize the image to 1024x768 with no scaling, and a stretched border, use:

```
img.Resize( 1024, 768, type='none', border='stretch' )
```

See Also

For more information on the import statement, *ReadFromFile()*, and *WriteToFile()*, see the *Creating an Image* section of this Cookbook

5.2.7 Create a QuickTime Movie

Problem

You have 200 frames of Patches playing with his ball, and you wish to combine them into a QuickTime movie. The individual frames are named using the format Patches_ball_###.jpg, where ### is a three digit frame number in the range 001 to 200.

Solution

```
import Draft
from DraftParamParser import ReplaceFilenameHashesWithNumber

encoder = Draft.VideoEncoder( 'Patches.mov' )    # Initialize the video encoder.

for currFrame in range( 1, 201 ):
    currFile = ReplaceFilenameHashesWithNumber( 'Patches_ball_###.jpg', currFrame )
    frame = Draft.Image.ReadFromFile( currFile )
    encoder.EncodeNextFrame( frame )     # Add each frame to the video.

encoder.FinalizeEncoding()     # Finalize and save the resulting video.
```

Discussion

In order to create a QuickTime movie, we need a video encoder. The line:

```
encoder = Draft.VideoEncoder( 'Patches.mov' )
```

creates a video encoder using default values for most parameters, and 'Patches.mov' specifies what filename the movie will be saved to. The optional parameters include, among others, fps, codec, and quality. The frames per second, fps, defaults to 24, the codec defaults to 'MJPEG', and the quality defaults to 85. If you wish to use values other than the defaults, simply use in the parameters. For example, we can increase the frame rate to 30, and decrease the quality to 70, using:

```
encoder = Draft.VideoEncoder( 'Patches.mov', fps=30, quality=70 )
```

The `fps` frame rate can be specified as an integer, floating point number, or `Fraction`. `quality` can be set to any integer in the range of 0 to 100, where higher numbers represent higher quality. (Note that you cannot simultaneously set both `quality` and `kbitRate`, another optional parameter.) In addition to 'MJPEG', valid codec values include 'MPEG4', 'H264', 'RAWVIDEO'.

Once the encoder is initialized, we can add our frames. Since there are many frames, we use a loop plus some logic to read in all of the frames one at a time. The function `ReplaceFilenameHashesWithNumber()` from the `DraftParamParser` library takes a filename pattern and replaces hash symbols with the frame number, padded with zeros to the appropriate number of digits indicated by the number of hash symbols. If no hash symbols are present,

`ReplaceFilenameHashesWithNumber()` will insert the frame number before the file extension (including period). We then use the constructed file name to read in the frame, and add it to the video using:

```
encoder.EncodeNextFrame( frame )
```

You may wonder why we used `range( 1, 201 )` in our for loop over the frames. This is because the second parameter to range specifies one past the end, not the actual last number. Since we want to include frame 200, we have to specify that 201 is where the range stops.

Once we have added all of our frames, we finalize the encoding, which finishes any remaining processing and saves the video to the file we specified previously. No parameters are required for the finalization:

```
encoder.FinalizeEncoding()
```

See Also

For more information on the import statement, *`ReadFromFile()`*, and *`WriteToFile()`*, see the *Creating an Image* section of this Cookbook

For more information on additional optional parameters for the video encoder, see *`Draft.VideoEncoder`*.

More information on `Fraction` is available in the Python Documentation.

5.2.8 Write an Image to File with a Specific File Channel Map

Problem

You want to write an image to file with a specific *file channel map*.

Solution

```
import Draft

# Read an image from file
image = Draft.Image.ReadFromFile( '//path/to/in.exr' )

# Create and set the file channel map
fileChannelMap = { 'R':'16f', 'G':'16f', 'B':'16f', 'A':'16f' }
image.SetFileChannelMap( fileChannelMap )

# Write the image back to file
image.WriteToFile( '//path/to/out.exr' )
```

Discussion

In order to specify the file channel map of an image you need to create a dictionary that represents a valid file channel map. A valid file channel map will have one entry for each image's channel and for those channels only. If you are unsure about the channels currently present in the image, you can get the current channel names by using:

```
print image.GetChannelNames()
```

To specify the file channel data types, you can consult the table in the section *Image File Channel Map* for valid file channel data types supported by Draft. Once you have loaded your image file and created a valid file channel map, you can set the file channel map using the method *`SetFileChannelMap()`*:

```
image.SetFileChannelMap( fileChannelMap )
```

When the file is written to file, the file channel map is adjusted according to the valid channel data types for the requested file format.

When you want to add another channel to your image in memory, you can use:

```
image.SetChannel( 'ID', 4 )
```

By default, Draft will assign a file channel data type of '8' to the new channel 'ID'. If you want to set it to another channel data type instead, you can modify the current file channel map using the method *GetFileChannelMap()* and *SetFileChannelMap()* in the following way:

```
fileChannelMap = image.GetFileChannelMap()
fileChannelMap[ 'ID' ] = '32ui'
image.SetFileChannelMap( fileChannelMap )
```

See Also

For more information on the import statement, *ReadFromFile()* and *WriteToFile()*, see the *Creating an Image* section of this Cookbook

You can also consult the sections *Image File Channel Map* in Concepts and *Write an Image to File with a Maximum Bit Depth* in the Intermediate Cookbook.

5.2.9 Set Saving Settings in an Image File

Problem

You want to set the compression, the quality and the tileSize in an image file.

Solution

```
import Draft

# Read an image from file
image = Draft.Image.ReadFromFile( '//path/to/in.exr' )

# Create a Draft.ImageInfo object and set compression, quality and tileSize
imageInfo = Draft.ImageInfo()
imageInfo.compression = 'dwaa'
imageInfo.quality = 80
imageInfo.tileSize = ( 64, 64 )

# Write the image back to file
image.WriteToFile( '//path/to/out.exr', imageInfo )
```

Discussion

In the above example, we decided to set the compression, the quality and the tileSize properties in an EXR image file. The lines:

```
imageInfo = Draft.ImageInfo()
imageInfo.compression = 'dwaa'
imageInfo.quality = 80
imageInfo.tileSize = ( 64, 64 )
```

creates a *ImageInfo* object and populates its fields with the desired valid values. For valid values, consult the section *ImageInfo* in Concepts.

Once you have created a valid *ImageInfo* object, you add the imageInfo as an additional parameter when the file is written to file:

```
image.WriteToFile( '//path/to/out.exr', imageInfo )
```

Alternatively, you can retrieve the saving settings in an image file using:

```
imageInfo = Draft.ImageInfo()
image = Draft.Image.ReadFromFile( '//path/to/in.exr', imageInfo )
compression = imageInfo.compression
quality = imageInfo.quality
tileSize = imageInfo.tileSize
```

See Also

For more information on the import statement, *ReadFromFile()* and *WriteToFile()*, see the *Creating an Image* section of this Cookbook.

You can consult the section *ImageInfo* in Concepts.

5.2.10 Embed a Timecode in an Image File

Problem

You want to embed a *timecode* in an image file.

Solution

```
import Draft

# Read an image from file
image = Draft.Image.ReadFromFile( '//path/to/in.dpx' )

# Create a Draft.Timecode object
timecode = Draft.Timecode( '12:40:10:15' )

# Create a Draft.ImageInfo object and set the timecode property
imageInfo = Draft.ImageInfo()
imageInfo.timecode = timecode

# Write the image back to file
image.WriteToFile( '//path/to/out.dpx', imageInfo )
```

Discussion

In the above example, we decided to embed a timecode in a DPX image file. Alternatively, it is possible to embed a timecode in an EXR image file. Note that those two file formats are the only one for which embedding a timecode is possible. The line:

```
timecode = Draft.Timecode( '12:40:10:15' )
```

creates a *Timecode* object representing a non-drop frame timecode. It is also possible to create a timecode representing a drop frame timecode using the following notation:

```
timecode = Draft.Timecode( '12:40:10;15' )
```

Once you have created a valid *Timecode* object, you need to create a *ImageInfo* object and set its timecode property in the following way:

```
imageInfo = Draft.ImageInfo()
imageInfo.timecode = timecode
```

Finally, you add the imageInfo as an additional parameter when the file is written to file:

```
image.WriteToFile( '//path/to/out.dpx', imageInfo )
```

Alternatively, you can retrieve a timecode previously embedded into an image file using:

```
imageInfo = Draft.ImageInfo()
image = Draft.Image.ReadFromFile( '//path/to/in.exr', imageInfo )
timecode = imageInfo.timecode
```

See Also

For more information on the import statement, *ReadFromFile()* and *WriteToFile()*, see the *Creating an Image* section of this Cookbook.

You can consult the sections *Timecode* in Concepts and *Embed a Timecode in a Video File* in the Intermediate Cookbook.

5.2.11 Concatenate Video Files

Problem

You want to concatenate some of your video files to create one single video file.

Solution

```
import Draft

# Define input and output files
inputVideoFile1 = '//path/to/movie_clip_1.mov'
inputVideoFile2 = '//path/to/movie_clip_2.mov'
inputVideoFile3 = '//path/to/movie_clip_3.mov'
outputVideoFile = '//path/to/complete_movie.mov'

# Concatenate video files
Draft.ConcatenateVideoFiles( [ inputVideo1, inputVideo2, inputVideo3 ], outputVideo )
```

Discussion

In order to successfully concatenate video files, the input files must all share the same codec and all have the same file extension than the output file. The line:

```
Draft.ConcatenateVideoFiles( [ inputVideoFile1, inputVideoFile2, inputVideoFile3 ], outputVideoFile )
```

concatenates the three input video clips and writes the result back in outputVideoFile, respecting the order in which they appear in the list.

5.3 Intermediate Operations

These operations are more involved but are typical things done with Draft such as working with Stereo footage and recipes for working with image sequences.

5.3.1 Create an Anaglyph Image

Problem

You want to create an anaglyph image from a left eye image and a right eye image.

Solution

Anaglyph images are easy to create in Draft.

```
import Draft
leftEyeImage = Draft.Image.ReadFromFile( "left.jpg" )
rightEyeImage = Draft.Image.ReadFromFile( "right.jpg" )
anaglyphImage = Draft.Image.Anaglyph( leftEyeImage, rightEyeImage, "LSA" )
```

Discussion

Anaglyph images are composed of two images, the left eye image and the right eye image. The last setting is the anaglyph type. In the above example we used `LSA` type anaglyph image but we also could have used `PS` type anaglyph image.

See Also

Draft.Image.Anaglyph()

5.3.2 Create a Frame Counter

Problem

You want to overlay a frame counter on a video.

Solution

To create a frame counter for a video we need to create a text annotation of the frame number and composite the frame number on the appropriate frame.

```
#Set up the decoder
decoder = Draft.VideoDecoder( "path/to/clip.mov" )
image = Draft.Image.CreateImage( 1, 1 )
frameNumber = 1
textInfo = Draft.AnnotationInfo()

#Set up encoder
encoder = Draft.VideoEncoder( "path/to/save/video.mov" )

while decoder.DecodeNextFrame( image ):
    frameText = Draft.Image.CreateAnnotation( str( frameNumber ), textInfo )

    #composite annotation onto frame
    anchor = Draft.Anchor.SouthEast
    compOp = Draft.CompositeOperator.OverCompositeOp
    image.CompositeWithAnchor( frameText, anchor, compOp )

    #encode the frame
    encoder.EncodeNextFrame( image )
    frameNumber += 1

encoder.FinalizeEncoding()
```

Discussion

Draft gives you a lot of freedom when creating a frame counter. Firstly, You can set `frameNumber` to start at any number you want. For example, you can make the frame count from the start of the reel or some other number like a timecode.

Also, you can set the frame number to be padded with zeros like this: 0012. Just change *`CreateAnnotation()`* to:

```
CreateAnnotation( "%04d" % frameNumber, textInfo )
```

The number after the zero is the total number of digits that will appear.

Secondly, the size, font and colour can be set to anything by changing `textInfo` (see the entry on *Create Text Annotation* for more information).

Finally, you can change the position of your frame counter by changing the *`Anchor`*. In this example, the frame counter will be put in the bottom right corner. By changing the *`Anchor`* you can make the frame counter appear anywhere on the frame such as the top right corner with a *`NorthEast`* anchor.

5.3.3 Split a Movie into Single Frames

Problem

You have a movie of Patches playing with his ball, 'Patches.mov', and you want to extract the individual frames into jpegs named Patches_###.jpg, where ### will be replaced by the corresponding three digit frame number.

Solution

```
import Draft
from DraftParamParser import ReplaceFilenameHashesWithNumber

frameNum = 1        # What number the first frame (jpeg) will have.
decoder = Draft.VideoDecoder( 'Patches.mov' )       # Initialize the video decoder.
frame = Draft.Image.CreateImage( 1, 1 )     # Create an image object.

while decoder.DecodeNextFrame( frame ) :
    currFile = ReplaceFilenameHashesWithNumber( 'decode/Patches_###.jpg', frameNum )
    frame.WriteToFile( currFile )
    frameNum += 1
```

Discussion

To separate a video into single frames, we need a *VideoDecoder* and an *Image* object. We initialize the *VideoDecoder* using the (path and) name of the movie we wish to decode, in this case, Patches.mov. The initial size of the *Image* is irrelevant, as the decoder will resize the image to fit the frame.

The call to *DecodeNextFrame()* attempts to save the next movie frame (if there is one) into the *Image* object, `frame`, and returns a boolean (true/false) to indicate if it was successful. Because the `frame` variable gets modified as a parameter, and a boolean is returned, we can use the call in our `while` loop to determine how long to loop: if another frame is successfully decoded, we want to process it and continue. If a frame wasn't successfully decoded, either we've reached the end of the movie, or some other problem is preventing us from continuing. Either way, there's no frame to process, and the loop stops.

Inside the loop we know we have a frame to save. We use the output path and filename pattern, 'decode/Patches_###.jpg' and frame number to produce a unique file name using `ReplaceFilenameHashesWithNumber()`, then write the frame to this file. Finally we increase the frame number, so that the next frame will be written to a differently numbered file.

Once the script has finished, the individual frames will be stored in the decode subdirectory (note: this script assumes the subdirectory named decode already exists), in separate files named Patches_001.jpg, Patches_002.jpg, Patches_003.jpg, etc., for as many frames as we decoded from the movie.

5.3.4 Create a Left-Eye, Right-Eye, Side-By-Side Movie

Problem

You have a set of frames pairs (left eye, right eye) for creating a 3D movie of Patches playing with his ball, and you want to use them to create a left-eye, right-eye, side-by-side movie, where each side is 640x480. The left eye images are saved using the filenames Patches_ball_left_###.jpg, with ### replaced by the three digit frame number, and the right eye images are similarly stored using the filenames Patches_ball_right_###.jpg.

Solution

```
import Draft
from DraftParamParser import ReplaceFilenameHashesWithNumber

width = 640
doubleWidth = width * 2  # twice the width to fit both left and right eye frames.
height = 480
encoder = Draft.VideoEncoder( 'Patches.mov', width=doubleWidth )  # Create encoder.
```

```
for currFrame in range( 1, 201 ): # Note: second parameter of range is one past end
    nameL = ReplaceFilenameHashesWithNumber( 'Patches_ball_left_###.jpg', currFrame )
    nameR = ReplaceFilenameHashesWithNumber( 'Patches_ball_right_###.jpg', currFrame )

    frameL = Draft.Image.ReadFromFile( nameL )
    frameR = Draft.Image.ReadFromFile( nameR )

    frameL.Resize( width, height )  # Make sure frames are correct size.
    frameR.Resize( width, height )

    frame = Draft.Image.CreateImage( doubleWidth, height )  # Frame to hold both eyes.
    compOp = Draft.CompositeOperator.OverCompositeOp
    frame.CompositeWithAnchor( frameL, Draft.Anchor.West, compOp )
    frame.CompositeWithAnchor( frameR, Draft.Anchor.East, compOp )

    encoder.EncodeNextFrame( frame )        # Add the frame to the video.

encoder.FinalizeEncoding()  # Finalize and save the resulting video.
```

Discussion

The main difference between this recipe and the basic *Create a QuickTime Movie* recipe is that here we have two images per frame of the movie, and we must load both and composite them into a single image before we can add them to the video encoder.

Using `Draft.Anchor.West` and `Draft.Anchor.East` we can easily place the images without having to worry about finding the exact location in the final image to use for the `Composite()` call. The left frame is placed at the leftmost edge with the `West` anchor and the right frame is place at the rightmost edge using the `East` anchor.

See Also

Create a QuickTime Movie, *Resize an Image*, and *Composite Two Images* in the Basic Cookbook

5.3.5 Skipping Missing Frames

Problem

You have around 200 frames of Patches playing with his ball, and you wish to combine them into a QuickTime movie. The individual frames are named using the format Patches_ball_###.jpg, where ### is a three digit frame number in the range 001 to 200. Some of the frames are missing. You'd still like to create the movie, but you'd rather not have to type in the entire list of frames that are there.

Solution

```
import Draft
from DraftParamParser import ReplaceFilenameHashesWithNumber

encoder = Draft.VideoEncoder( 'Patches.mov' )       # Initialize the video encoder.

# Note that the second parameter of range is one past the last frame number
for currFrame in range( 1, 201 ):
    currFile = ReplaceFilenameHashesWithNumber( 'Patches_ball_###.jpg', currFrame )
```

```
    try:
        frame = Draft.Image.ReadFromFile( currFile )        # Try to read the frame.
    except:
        pass        # Reading was unsuccessful, skip frame.
    else:
        encoder.EncodeNextFrame( frame )    # Add the frame to the video.

encoder.FinalizeEncoding()  # Finalize and save the resulting video.
```

Discussion

If we try to read an image from a file that doesn't exist, the *ReadFromFile()* method will throw an exception. Since we want to continue even if some of the images are missing, we use the `try`/`except` construct: we try to load in the image; if loading fails, we catch the exception (`except`) and don't do anything (`pass`), otherwise (`else`), we add the frame to the video.

An alternate method is to replace the `try`/`except` construct with an `if` statement and the `exists()` function in the `os.path` module:

```
if os.path.exists( currFile ):
    frame = Draft.Image.ReadFromFile( currFile )    # Read the frame.
    encoder.EncodeNextFrame( frame )    # Add the frame to the video.
```

See Also

Create a QuickTime Movie in the Basic Cookbook.

5.3.6 Create Proxies for an Image Sequence

Problem

You have around 200 frames of Patches playing with his ball, and you wish to combine them into a QuickTime movie. The individual frames are named using the format Patches_ball_###.jpg, where ### is a three digit frame number in the range 001 to 200. Some of the frames are missing. You'd still like to create the movie, but you'd like proxies to be placed where the missing frames are.

Solution

```
import Draft
from DraftParamParser import ReplaceFilenameHashesWithNumber

proxy = Draft.Image.CreateImage( 640, 480 ) # Create a proxy frame.
proxy.SetToColor( Draft.ColorRGBA( 1, 0, 0, 1 ) )   # Make the proxy noticeable.

encoder = Draft.VideoEncoder( 'Patches.mov' )       # Initialize the video encoder.

# Note that the second parameter of range is one past the last frame number
for currFrame in range( 1, 201 ) :
    currFile = ReplaceFilenameHashesWithNumber( 'Patches_ball_###.jpg', currFrame )
    try:
        frame = Draft.Image.ReadFromFile( currFile )        # Try to read the frame.
    except:
```

```
        frame = proxy       # Reading was unsuccessful, use proxy frame.
    encoder.EncodeNextFrame( frame )        # Add the frame to the video.

encoder.FinalizeEncoding()  # Finalize and save the resulting video.
```

Discussion

If we try to read an image from a file that doesn't exist, the *ReadFromFile()* method will throw an exception. Since we want to continue even if some of the images are missing, we use the `try`/`except` construct: we try to load in the image; if loading fails, we catch the exception (`except`) and use the proxy frame instead. In both cases we have a frame to add to the video, and so this statement comes after the `try`/`except` block.

Since the proxy frame is the same for all missing frames, we can create it once, before the loop. In this example, we have set the color of the proxy frame to red, so that it is easily located in the movie.

An alternate method is to replace the `try`/`except` construct with an `if`/`else` statement and the `exists()` function in the `os.path` module:

```
if os.path.exists( currFile ):
    frame = Draft.Image.ReadFromFile( currFile )    # Read the frame.
else:
    frame = proxy     # Reading was unsuccessful, use proxy frame.
```

See Also

Create a QuickTime Movie in the Basic Cookbook.

5.3.7 Saving Multiple Videos from the Same Input

Problem

You would like to create movies of Patches playing with his ball in multiple formats, from the same set of Patches_ball_###.jpg input files, where ### represents a three digit frame number in the range 001 to 200. You have decided that you want an MJPEG codec movie at 640x480 pixels, and a H264 codec movie at 1280x720 pixels.

Solution

```
import Draft
from DraftParamParser import ReplaceFilenameHashesWithNumber
from copy import deepcopy

encoderMJPEG = Draft.VideoEncoder( 'PatchesMJPEG.mov', codec='MJPEG',
                        width=640, height=480 )  # Initialize the MJPEG video encoder.
encoderH264 = Draft.VideoEncoder( 'PatchesH264.mov', codec='H264',
                        width=1280, height=720 )  # Initialize the H264 video encoder.

# Note that the second parameter of range is one past the last frame number
for currFrame in range( 1, 201 ):
    currFile = ReplaceFilenameHashesWithNumber( 'Patches_ball_###.jpg', currFrame )
    frame = Draft.Image.ReadFromFile( currFile )    # Try to read the frame.

    frameCopy = deepcopy( frame )
    frameCopy.Resize( 640, 480 )
```

```
    encoderMJPEG.EncodeNextFrame( frameCopy )       # Add the frame to the video.

    frameCopy = deepcopy( frame )
    frameCopy.Resize( 1280, 720 )
    encoderH264.EncodeNextFrame( frameCopy )        # Add the frame to the video.

encoderMJPEG.FinalizeEncoding()     # Finalize and save the resulting video.
encoderH264.FinalizeEncoding()      # Finalize and save the resulting video.
```

Discussion

We need a seperate encoder object for each movie we are creating. We can then add the resized frames to the matching encoders. Be sure to remember to finalize each of the encoders.

To avoid a degredation in image quality from resizing multiple times, we work with a copy of the original image each time we want the frame in a new size. Since assigning an image to a new variable (`img2 = img1`) does not actually duplicate the internal data, we need to use the `deepcopy()` method from the `copy` module. Since we don't need the original frame any more after adding the frame to the last (in this case second) of the encoders, we could skip the final deep copy, and simply resize and encode the original frame.

See Also

Create a QuickTime Movie and *Resize an Image* in the Basic Cookbook.

5.3.8 Change the Encoding of a Movie Clip

Problem

You want to change the encoding of a movie clip.

Solution

Use the *VideoDecoder* to decode your existing clip and then use the *VideoEncoder* to encode that clip into a different codec. For example, the following code converts an input QuickTime to h.264.

```
image = Draft.Image.CreateImage( 1, 1 )

decoder = Draft.VideoDecoder( "//path/to/existing/clip.mov" )

hasFrames = decoder.DecodeNextFrame( image )
if hasFrames:
    encoder = Draft.VideoEncoder( "//path/to/video/save.mov", codec='H264' )

    while hasFrames:
        encoder.EncodeNextFrame( image )
        hasFrames = decoder.DecodeNextFrame( image )

    encoder.FinalizeEncoding()
```

Discussion

Taking a movie clip and encoding it with a new codec is just a matter of decoding the existing clip and encoding it into a new clip. An important thing to remember is that the paths for the decoder and the encoder must be different. You can't decode and encode the same file at the same time.

Draft will detect which decoder to use automatically so you do not need to specify it when you are decoding.

Draft supports multiple codec options. You can consult the sections *Video* in Concepts for a complete list.

5.3.9 Setting the Compression Quality when Encoding a Movie Clip

Problem

You want to set the compression quality when encoding a movie clip.

Solution

There are two ways of setting the compression quality when encoding a movie. The first way is to specify the bit rate of the encoder. The bit rate is typically measure in kilobits so the parameter to set is `kbitRate`:

```
encoder = Draft.VideoEncoder( '//path/to/video/save.mov', kbitRate = 85 )
```

The other way to set the compression quality is with the `quality` keyword:

```
encoder =  Draft.VideoEncoder( '//path/to/video/save.mov', quality = 80 )
```

Discussion

The compression `quality` parameter is an integer value between 0 and 100. Greater values correspond to higher quality. When using a specified quality Draft will vary the bitrate to maintain the desired quality. This works in most instances.

Specifying the bitrate provides a lot of control, but is only recommended if you the specific bitrate your movie needs to be. `kbitRate` is more dependent on the codec than `quality`. For example, Avid DNxHD® accepts only certain kbitRate values. On the other hand H.264 accepts almost any value for `kbitRate` but issues arise with very low and very high values. If you don't need to use a specific bitrate then use the `quality` keyword and let Draft do the work for you.

These two settings are exclusive and only one of `quality` or `kbitRate` can be specified.

See Also

Draft.VideoEncoder

5.3.10 Add a Slate Frame to a Movie Clip

Problem

You want to add a slate frame to a movie clip.

Solution

The first step is to either load a standard slate frame or create a new image to insert in front of the existing frames. You can even add text to this image to incorporate meta-data like shot information or project information.

```
slate = Draft.Image.CreateImage( outWidth, outHeight )
slate.SetToColor( Draft.ColorRGBA( 0.0, 0.0, 0.0, 1.0 ) )

#sets up the text on the slate frame
slateText = [("SHOW", jobParams.get('ExtraInfo1', '')), #skipped if no ExtraInfo1
        ("EPISODE", params.get('episode', '')), #skipped if 'episode' not in extra args
        ("SHOT", params['entity']),
        ("FRAMES", params['frameList']),
            ("HANDLES", params.get('handles', '')), #skipped if 'handles' not in extra args
        ("VERSION", params['version']),
        ("",''),
        ("",''),
        ("ARTIST", params['username']),
        ("DATE/TIME", datetime.datetime.now().strftime("%m/%d/%Y %I:%M %p") )]

#comp the annotations over top the slate frame
skipLines = 0
for i in range( 0, len( slateText ) ):
    if ( slateText[i][1] == "" ):
        skipLines += 1
        continue

    lineNum = i - skipLines
    if ( slateText[i][0] != "" ):
        txtImg = Draft.Image.CreateAnnotation( slateText[i][0] + ": ",
                annotationInfo )
        slate.CompositeWithPositionAndAnchor( txtImg, 0.45, 0.7 - (lineNum * 0.06),
                Draft.Anchor.SouthEast, Draft.CompositeOperator.OverCompositeOp )

    if ( slateText[i][1] != "" ):
        txtImg = Draft.Image.CreateAnnotation( slateText[i][1], annotationInfo )
        slate.CompositeWithPositionAndAnchor( txtImg, 0.46, 0.7 - (lineNum * 0.06),
                Draft.Anchor.SouthWest, Draft.CompositeOperator.OverCompositeOp )
```

Once the slate frame is created, you simply need to encode this image before feeding the rest of the frames into the *VideoEncoder*. By re-encoding the image multiple times you can hold the slate for as long as you desire. For example if we wanted to hold our slate for half a second in a 24 frames per second clip, we would encode the slate 12 times.

```
numberOfSlateFrames = 12 # hold for half a second @ 24fps
#encode the slate frames at the start of the video
for i in range( 0, numberOfSlateFrames ):
    encoder.EncodeNextFrame( slate )
#encode the rest of the video after this
```

Discussion

The first two lines of the solution create a black frame for our slate information to be written on. The next part is when we set up all the information that will appear on our slate. All of this info comes from Deadline (See Deadline Integration and Param Parsing for more details). The next parts will create the text images and composite them onto our black slate so that the titles (like SHOW or ARTIST) appear on the left and the values appear on the right.

The last part of the solution encodes the slate frame to the start of our video. Set `numberOfSlateFrames` to be the number of frames you want your slate to be.

5.3.11 Using the Parameter Parsing Utility for Custom Parameters

Problem

Sometimes you want to do several tasks that differ only slightly. Instead of creating a separate script for each task, we can create a more general script that uses parameters for the details that differ.

With all your great photos of Patches, you want to create a simple LOLcat script that will add two text strings to a photo, one at the top, and one at the bottom, both centered. You have already picked out the perfect font, etc, but each photo will have different text, and all the photos have different names. You also want the option of specifying the font size for the bottom text, in case it is too long to fit at the default size.

Solution

```
import sys       # Deadline sends script parameters as command line arguments, sys.argv.
from DraftParamParser import * # Draft helper functions for processing arguments.

# The argument name/type pairs we're expecting.
expectedTypes = dict()
expectedTypes['inFile'] = '<string>'
expectedTypes['outFile'] = '<string>'
expectedTypes['topText'] = '<string>'
expectedTypes['bottomText'] = '<string>'
# We don't specify the bottom text size parameter here, because it is optional.
# We will use ParseCommandLine_TypeAgnostic to check for it.

# Parse the command line arguments.
params = ParseCommandLine( expectedTypes, sys.argv ) # params now contains a
        # dictionary of the parameters initialized to values from the arguments.
inFile = params['inFile'] # The photo we wish to use for the lolcat image.
outFile = params['outFile'] # Where to save the lolcat image.
topText = params['topText'] # Text to place at the top of the image.
bottomText = params['bottomText'] # Text to place at the bottom of the image.

image = Draft.Image.ReadFromFile( inFile ) # Read in the photo to use for our LOLcat.

# Set up how you want the text to appear.
textInfo = Draft.AnnotationInfo()
textInfo.PointSize = 48

# Create the image for the top text.
topTextImage = Draft.Image.CreateAnnotation( topText, textInfo )

edgeOffset = 20.0 / image.height  # Position text 20 pixels from the top and bottom.

# Add the top text to the output image.
image.CompositeWithPositionAndAnchor( topTextImage, 0.5, 1.0 - edgeOffset,
        Draft.Anchor.North, Draft.CompositeOperator.OverCompositeOp )

# Check to see if we specified the optional bottom text size.
paramsA = ParseCommandLine_TypeAgnostic( sys.argv )
if ( 'bottomTextSize' in paramsA ) :
    # We must convert non-string types ourselves for type agnostic parameters.
```

```
    textInfo.PointSize = int( paramsA[ 'bottomTextSize' ] )

# Create the image for the bottom text.
bottomTextImage = Draft.Image.CreateAnnotation( bottomText, textInfo )

# Add the bottom text to the output image.
image.CompositeWithPositionAndAnchor( bottomTextImage, 0.5, edgeOffset,
        Draft.Anchor.South, Draft.CompositeOperator.OverCompositeOp )

image.WriteToFile( outFile )  # Save the completed LOLcat image.
```

Discussion

To run this script from the command line, assuming it is in a file called LOLcat.py, we can use:

```
> python lolcat.py inFile="Patches.jpg" outFile="PatchesLOL.jpg" topText="LOL" \
  bottomText="I AM PATCHES"
```

or:

```
> python lolcat.py inFile="Patches.jpg" outFile="PatchesLOL.jpg" topText="I AM CAT" \
  bottomText="HEAR ME ROAR!!!" 'bottomTextSize'=88
```

Valid types for the expected parameter types include `'<string>'`, `'<int>'` and `'<float>'`.

`DraftParamParser` contains other useful functions. For example, to convert the parameter from a string to a list of frames, use:

```
frames = FrameRangeToFrames( params['frameList'] )  # Get a list of individual frames
```

To create a filename from a filename pattern and frame number, use:

```
currFilename = ReplaceFilenameHashesWithNumber( filePattern, currFrameNum )
```

If we expect to be using the same parameter values multiple times, we can store them in a file and parse them using:

```
params = ParseParamFile( expectedTypes, paramFile )
```

5.3.12 Write an Image to File with a Maximum Bit Depth

Problem

You want to write an image to file and specify a maximum bit depth.

Solution

```
import Draft
from DraftParamParser import SplitDataType      # Import SplitDataType from the DraftParamParser libi

# Read an image from file
image = Draft.Image.ReadFromFile( '//path/to/in.exr' )

# Get the current file channel map
fileChannelMap = image.GetFileChannelMap()
```

```
# Set a max bit depth of 16
for key in fileChannelMap:
    splitDataType = SplitDataType( fileChannelMap[key] )
    bitDepth = int( splitDataType[0] )
    dataType = splitDataType[1]
    if bitDepth > 16:
        fileChannelMap[key] = '16' + dataType

# Set the file channel map
image.SetFileChannelMap( fileChannelMap )

# Write the image back to file
image.WriteToFile( '//path/to/out.exr' )
```

Discussion

If you want to limit the bit depth of an image file that was read from file to a maximum of 16, you need to first get that image's file channel map using:

```
fileChannelMap = image.GetFileChannelMap()
```

Then, fileChannelMap will store a dictionary containing the data type for each channel present in the image. The function `SplitDataType()` from the `DraftParamParser` library takes a valid data type string made of a strictly positive number followed by an optional group of letters and split this string in two, the first part being the bit depth and the second the optional group of letters. Looping through all the image channel, you can easily extract the bit depth of each channel by using:

```
splitDataType = SplitDataType( fileChannelMap[key] )
bitDepth = int( splitDataType[0] )
dataType = splitDataType[1]
```

and you can modify the channel bit depth using:

```
if bitDepth > 16:
    fileChannelMap[key] = '16' + dataType
```

You can then write the image back to file. Note that even if, in this process, you set the file channel data type to '16ui' before writing an EXR image to file, the channel will still be written as '32ui' since an EXR unsigned integer always has a bit depth of 32.

See Also

You can consult the sections *Image File Channel Map* in Concepts and *Write an Image to File with a Specific File Channel Map* in the Basic Cookbook.

5.3.13 Embed a Timecode in a Video File

Problem

You want to extract a timecode from a EXR image sequences and embed it in a video file.

Solution

```
import Draft
from DraftParamParser import ReplaceFilenameHashesWithNumber    # Import ReplaceFilenameHashesWithNumb

# Create a Draft.ImageInfo object used to extract a Draft.Timecode object
imageInfo = Draft.ImageInfo()

# Main encoding loop
for currFrame in range( 1, 200 ):
    currFile = ReplaceFilenameHashesWithNumber( 'movie_frame####.exr', currFrame )
    frame = Draft.Image.ReadFromFile( currFile, imageInfo )

    # If first frame and a timecode is found, specify the timecode parameter when creating the encode
    if currFrame == 1:
        if( imageInfo.timecode ):
            encoder = Draft.VideoEncoder( '//path/to/video/save.mov', timecode = imageInfo.timecode )
        else:
            encoder = Draft.VideoEncoder( '//path/to/video/save.mov' )

    encoder.EncodeNextFrame( frame )

encoder.FinalizeEncoding()
```

Discussion

In order to embed a timecode in a video file, you can extract the timecode embedded in the first frame of an image sequence. To do so, you first need to check if there's one. If a *Timecode* is set in imageInfo object, the if clause will execute and we'll initialize the video encoder and embed the first frame's timecode using:

```
encoder = Draft.VideoEncoder( '//path/to/video/save.mov', timecode = imageInfo.timecode )
```

If no timecode is found, we'll have:

```
imageInfo.timecode = None
```

and the else clause will execute, initializing the encoder using:

```
encoder = Draft.VideoEncoder( '//path/to/video/save.mov' )
```

The remaining encoding steps are the same as usual, finishing with the line:

```
encoder.FinalizeEncoding()
```

to finalize and save the resulting video to file.

See Also

You can consult the sections *Timecode* in Concepts and *Embed a Timecode in an Image File* in the Basic Cookbook.

5.3.14 Display an Image Using PySide

Problem

You want to access a Draft.Image as a byte array and display it with PySide.

Solution

```
import Draft

# Required imports for using PySide
import sys
from PySide.QtCore import *
from PySide.QtGui import *

# Read an image from file and creates an ARGB32 string representing the image
img = Draft.Image.ReadFromFile( '//path/to/in.exr' )
imgAsByteArray = img.ToBytes()

# Displays the image using PySide
app = QApplication( sys.argv )
label = QLabel()

imgQT = QImage( imgAsByteArray, img.width, img.height, QImage.Format_ARGB32 )
pixMap = QPixmap.fromImage( imgQT )

label.setPixmap( pixMap )
label.show()

app.exec_()
```

Discussion

To display an image using PySide you first need to load the image in memory. Then, you can create a byte array representing the image using:

```
imgAsByteArray = img.ToBytes()
```

For more details on the syntax used by PySide, you can consult the PySide Documentation.

5.4 Advanced Operations

These operations target an audience that is already comfortable with writing Draft scripts.

5.4.1 Extract EXR Layers to Images

Problem

You rendered a sequence of multi-layer EXR and you need to work with each layer separately. You want to extract each layer and save each of them as a separate EXR image file. In addition, you would like Draft to identify each layer based on your multi-layer EXR's channel names and to write each layer to file automatically with a meaningful filename.

Solution

```
import Draft
from DraftParamParser import *

startFrame = 1
endFrame = 100
inFilePattern = "//path/to/multi-layer####.exr"
outFilePattern = "//path/to/output.exr"

# Define a dictionary holding channel name equivalences
channelMap = { 'red': 'R', 'green': 'G', 'blue': 'B', 'alpha': 'A',
               'r': 'R', 'g': 'G', 'b': 'B', 'a': 'A' }

# Get the first frame's channel names
currFile = ReplaceFilenameHashesWithNumber( inFilePattern, 1 )
firstFrame = Draft.Image.ReadFromFile( currFile )
channelNames = firstFrame.GetChannelNames()

# Create a dictionary of layers,
# with the layer name as the key, and the value as the list of channels
layers = {}
for name in channelNames:
    # Specific layer
    if string.rfind( name, '.' ) >= 0:
        ( layer, channel ) = string.rsplit( name, '.', 1 )
    # Basic RGB(A) layer
    else:
        ( layer, channel ) = ( None, name )
    if channel.lower() in channelMap:
        if layer in layers:
            layers[layer].append( channel )
        else:
            layers[layer] = [channel]
    else:
        print "Ignoring the following layer.channel: ", name, " (channel not recognized as RGBA)"


# Process each of the frames in the list of frames
for currFrame in range ( startFrame, endFrame + 1 ):
    # Read in the frame.
    currFile = ReplaceFilenameHashesWithNumber( inFilePattern, currFrame )
    frame = Draft.Image.ReadFromFile( currFile )

    # Extract the layers from the image, saving each as a separate image.
    for ( layer, channels ) in layers.items():
        prefix = ''
        if layer is not None:
            prefix = layer + '.'
        channelList = [ prefix + channel for channel in channels ]
        imgLayer = Draft.Image.CreateImage( frame.width, frame.height, channelList )
        imgLayer.Copy( frame, channels = channelList )

        # Rename the channels to RGB(A).
        for channel in channels:
            if prefix + channel != channelMap[channel.lower()]:
                imgLayer.RenameChannel( prefix + channel, channelMap[channel.lower()] )

        # Add the layer name to the filename.
        currOutFile = ReplaceFilenameHashesWithNumber( outFilePattern, currFrame )
```

```
        if layer is not None:
            # Specific layer
            if string.rfind( currOutFile, '_' ) >= 0:
                ( first, second ) = string.rsplit( currOutFile, '_', 1 )
                currOutFile = first + '_' + layer + '_' + second
            # Basic RGB(A) layer
            else:
                ( root, ext ) = os.path.splitext( currOutFile )
                currOutFile = root + '_' + layer + ext

        # Write the layer to file as an image
        imgLayer.WriteToFile( currOutFile )
```

Discussion

In order to be able to extract layers from a wide variety of multi-layer EXR, it's convenient to first define a dictionary of channel name equivalence:

```
channelMap = { 'red': 'R', 'green': 'G', 'blue': 'B', 'alpha': 'A',
               'r': 'R', 'g': 'G', 'b': 'B', 'a': 'A' }
```

In that case, 'R', 'red' and 'r' will be treated as being equivalent. This dictionary can be customized to fit your specific needs.

Then, we can create a dictionary of layers based on the channel names of the first frame. This code snippet assumes that all images in the sequence have the same layers as the first frame but this assumption is not mandatory. You can also create a dictionary of layers for each frame by moving its creation in the main loop.

Once the dictionary of layers is created, we are ready to extract the layers from each frame, saving each one as a separate image. For each layer, we get the corresponding `channelList` from the dictionary of layers and we use it to extract the layer in the following lines of code:

```
imgLayer = Draft.Image.CreateImage( frame.width, frame.height, channelList )
imgLayer.Copy( frame, channels = channelList )
```

Note that the method *`Copy()`* only copy over to `imgLayer` the channels in `channelList`. This is where the actual layer extraction takes place.

Then, the `imgLayer`'s channel names are renamed to RGB(A) and the output filename is renamed to include the current layer's name. Finally, `imgLayer` is written to file.

To summarize, a set of images, one image per EXR layer, is written to file for each frame. For instance, let's suppose the list of channel names in your multi-layer EXR image is:

```
{Reflect.R, Reflect.G, Reflect.B, Refract.R, Refract.G, Refract.B, Shadow.R, Shadow.G, Shadow.B, R, G
```

Then, this code snippet will create the following EXR image files associated with the sequence's first frame:

```
output_Reflect_0001.exr, output_Refract_0001.exr, output_Shadow_0001.exr, output_0001.exr
```

where `output_0001.exr` contains the basic RGB(A) layer. A similar set of images will be written to file for each frame in the sequence.

5.5 Deadline Integration

The Deadline section focusses specifically on how to connect your Draft scripts to Deadline and how to take advantage of things like custom parameters and Deadline Job meta-data.

5.5.1 Convert a Frame String to Frame List

Problem

Deadline passes frames to Draft in the form of a Frame Range (such as `"1-100"`). Converting this to a List of Frame Numbers can be non-trivial, given how complex Frame Strings can get (e.g., `"105,200-400x3,500-600step4,1-100"` is a valid Frame String).

Solution

Fortunately, we have added a function in Draft's `DraftParamParser` helper script specifically to alleviate this problem:

```
import Draft
from DraftParamParser import * #Needed to use the utility function

#Sample input
frameRange = "1-10x2,11-15" #Equivalent to 1,3,5,7,9,11,12,13,14,15

frameList = FrameRangeToFrames( frameRange )

#You can now iterate over the Frame List
for frameNumber in frameList:
    #Do something with the FrameNumber
    print frameNumber
```

Discussion

In addition to the obvious functionality of expanding a Frame Range string into a list of Frames, the `FrameRangeToFrames()` function also sorts the frames in ascending order and ensures that there are no duplicate frames in the List.

Simple Frame Ranges

It should be noted that Deadline also passes the First and Last frames in the given Frame Range as separate arguments to Draft (`startFrame` and `endFrame`, respectively). If you're not interested in supporting complex Frame Ranges in your Template, you can simply use these values to generate a list of frames as follows:

```
for frameNumber in range( startFrame, endFrame + 1 ):
    #Do something with frameNumber
    print frameNumber
```

5.5.2 Substituting Frame Padding for a Frame Number

Problem

You have a filename with Frame Padding, but need to get the filename for a specific frame in order to Read it in with Draft.

Solution

Simply use the `ReplaceFilenameHashesWithNumber()` helper function provided in Draft's ParamParser helper script:

```
import Draft
from DraftParamParser import * #Needed to use the utility function

#sample inputs for this example
inFile = r"X:\project\shot\frame_list_####.png"
frameNumber = 24

swappedFileName = ReplaceFilenameHashesWithNumber( inFile, frameNumber )

frameImage = Draft.Image.ReadFromFile( swappedFileName )
```

Discussion

The `ReplaceFilenameHashesWithNumber()` utility function does exactly what its name implies. It replaces hashes (`'#'`) in the given file name (the first argument) with the given frame number (the second argument). The function will automatically pad the frame number with 0's in order to match the length of the padding string.

It should be noted that Deadline will always try to pass the `inFile` argument to the Draft template as a hash-padded filename. If the Draft input is taken from the output of a rendering application that does not normally use `'#'` as a padding character (e.g., Maya), Deadline should detect this and swap it to `'#'` whenever possible

5.5.3 Setting Up Custom Command Line Parameters

Problem

If you are re-using the same code in all your templates to fetch some extra information, it might be worth considering adding a custom command line parameter to Draft. This is a slightly more advanced technique since it requires making and maintaining changes to some built-in Deadline scripts, but it can potentially save you a lot of work in the long run.

Solution

There are a total of three possible places from where a Draft Job can be submitted to Deadline:

1. **The independent Draft Submission Script in the Monitor**
    - Relevant Script: scripts/Submission/DraftSubmission/DraftSubmission.py
2. **The Job right-click Draft script in the Monitor**
    - Relevant Script: scripts/Jobs/JobDraftSubmission/JobDraftSubmission.py
3. **The Draft Event Plugin**
    - Relevant Script: events/Draft/Draft.py

The three different scripts listed above do slightly different things throughout, but the creation of the Draft job itself is more or less the same across all of them. The area of interest for this particular Recipe is near the end of the script, where the Draft arguments are being created. It should look something like this (keep in mind variable names might be slightly different):

```
#prep the script arguments
args = []
args.append( 'username="%s" ' % scriptDialog.GetValue( "UserBox" ) )
args.append( 'entity="%s" ' % scriptDialog.GetValue( "EntityBox" ) )
args.append( 'version="%s" ' % scriptDialog.GetValue( "VersionBox" ) )
```

To add another argument to Draft, add another line similar to the `args.append( ... )` lines in this section, and supply it with the value you wish to pass to your template:

```
#prep the script arguments
args = []
args.append( 'username="%s" ' % scriptDialog.GetValue( "UserBox" ) )
args.append( 'entity="%s" ' % scriptDialog.GetValue( "EntityBox" ) )
args.append( 'version="%s" ' % scriptDialog.GetValue( "VersionBox" ) )

#New custom Draft parameter!
newArgValue = "Hello World!"
args.append( 'argumentName="%s" ' % newArgValue )
```

Now, whenever a new Draft job is submitted through this particular script, Deadline will pass another parameter to your scripts named `'argumentName'`, with the string value `"Hello World!"`. Keep in mind that you will have to modify all three scripts listed above if you want your change to apply to all types of Draft submissions.

Discussion

Note that even though many other (non-Draft) submitters have Draft sections in them, all of these use the Draft Event Plugin to actually do the submission to Deadline.

Argument Types

While you could technically pass any arbitrarily formatted string to Draft as a parameter, if you're making use of the `DraftParamParser` utility functions, Draft will expect parameters to be formatted as `'argumentName=<argumentValue>'`.

The argument's value can only be of the following types:

- **String**
  - Format Code `'%s'`
  - Be sure to add quotes around the value
- **Integer**
  - Format Code `'%d'`
  - No quotes needed
- **Float (decimal value)**
  - Format code `'%f'`
  - No quotes needed
- **A list combining multiple of the above types**
  - Lists are to be enclosed in parentheses, and elements should be separated by commas
  - E.g.: `(1, "string value", 3.5, "another string")`

Here is some code to show what each of these might look like, in the context of the example above:

(NOTE: *Pay special attention to how the '%' format codes change based on the type of value it is!*)

```
#String value
strValue = "Hello World!"
args.append( 'stringValue="%s" ' % strValue )

#Integer value
intValue = 25
args.append( 'intValue=%d ' % intValue )

#Float value
floatValue = 3.141593
args.append( 'floatValue=%f ' % floatValue )

#List of values
args.append( 'listValues=(%s,%d,%f) ' % (strValue,intValue,floatValue) )
```

5.5.4 Using Deadline Job Values

NOTE: The following was written for Deadline 5, and needs to be updated for Deadline 6/7. For now, see the "simple_slate_h264_*" scripts in yourRepository/draft/Samples/Encode/ for updated examples.

Problem

When a Draft job is submitted to Deadline using either the Draft Event Plugin or the Job right-click Submission Script, it is tied to another Deadline Job. This 'original' Job is generally the Job that was responsible for creating the input for Draft, and a lot of info from this Job is passed onto Draft. However, Deadline does not pass *all* of the Job Properties, and getting those into your Draft script will require either parsing the XML Job file or making calls to DeadlineCommand and parsing its output.

Solution

Fortunately, we have recently added a helper function to the `DraftParamParser` utility script to do this complicated work for you. The `GetDeadlineJobProperties()` function takes the path to the Deadline Repository and a Job ID as parameters, and returns a dictionary of Job Properties for that particular Job:

```
from DraftParamParser import *

#Sample input
deadlineRepoPath = r"\\repoServer\DeadlineRepository" #Deadline repository
deadlineJobID = "999_050_999_4253fd64" #The ID of the Deadline Job to parse

#Call the utility function
jobProps = GetDeadlineJobProperties( deadlineRepoPath, deadlineJobID )

#The function returns a dictionary, you can now access any Job property:
print jobProps["Name"] #prints out the original Job's name

#Some properties are lists:
print jobProps["AuxiliaryFileNames"][0] #prints out the first aux filename

#ExtraInfoKeyValues is a dictionary:
print jobProps["ExtraInfoKeyValues"]["DraftTemplate"]
```

Discussion

As a general rule, values in the Dictionary will be returned as strings (or `None` if there is no value). Exceptions to this rule are for any property containing lists (which are returned as lists of strings), and ExtraInfoKeyValues, which are returned as a Dictionary. Available properties generally correspond to the Job properties listed here, minus the `'Job'` prefix.

To confirm the name of a given property, you can check by simply opening a .job file found in the Deadline Repository and inspecting the Tag names.

Getting the Deadline Job ID

If the Draft job is tied to another Deadline job, as described in this entry's Problem description, Deadline will pass the ID of this 'original' job as a parameter to your Draft template, as `'deadlineJobID=<job ID>'`. You can get the actual value from the template parameters by using the Param Parser utility function, as shown in the earlier Cookbook entry *Setting Up Custom Command Line Parameters*.

Cross-Platform Path Considerations

You may have noted that the `GetDeadlineJobProperties()` function expects you to provide it with the path to the Deadline Repository. If you have a cross-platform render farm, that path might be different based on which OS the Slave is running on. Aside from setting up an OS-specific Group for Draft jobs, there would be no way to guarantee which OS the Draft Job might run on. In that case, it would be best to specify the Repository Path based on the current OS, as follows:

```
import sys

dealdineRepoPath = ""
if sys.platform.startswith( 'linux' ): #Linux
    deadlineRepoPath = r"/mnt/DeadlineRepository"
elif sys.platform.startswith( 'darwin' ): #Mac OSX
    deadlineRepoPath = r"/Volumes/DeadlineRepository"
elif sys.platform.startswith( 'win32' ): #Windows
    deadlineRepoPath = r"\\repoServer\DeadlineRepository"
```

5.5.5 Parsing File Names to Extract Shot Information

Problem

Many studios embed shot/project information within their file structure. Parsing out this information can seem daunting if you're new to Python.

Solution

Python has many built-in string/filename parsing utilities that make this problem seem somewhat trivial once you're used to them. Let us assume that each entity we're looking to parse from the filename consists of a directory (e.g. X:\project_name\sequence_name\shot_name\image_sequence_####.png). In this case, we can use Python's `os.path.split()` function to walk through the directory structure:

```
import os

#Sample input
```

```
inFile = r"X:\\project\\sequence\\shot\\image_sequence_####.png"

(head, tail) = os.path.split( inFile )
#tail is now 'image_sequence_####.png' and
#head is now 'X:\\project\\sequence\\shot'

#do it again (on head this time) to go back another level
(head, tail) = os.path.split( head )
#head is now 'X:\\project_name\\sequence'
#tail is now 'shot' - our shot name!
shotName = tail

#keep walking backwards to get more info!
(head, tail) = os.path.split( head )
#head is now 'X:\\project'
#tail is now 'sequence' - our sequence name!
sequenceName = tail

#one more...
(head, tail) = os.path.split( head )
#head is now just 'X:'
#tail is now 'project' - our project name!
projectName = tail
```

Note that if you have more than one entity name in your directories, you can still use the above approach, but you will need the generic Python's String version of `split()`, to divide things up even more.

Let's say for the sake of example that one of your directories is the sequence name AND shot name, separated by an underscore: `"sequence_shot"`. You can then use `split()` to split the string by the underscore, which acts as a separator:

```
#Sample input
directoryName = "sequence_shot"

splitResult = directoryName.split( "_" ) #split on underscores
sequenceName = splitResult[0] #first part of the split is the sequence
shotName = splitResult[1] #second part of the split is the shot
```

This is obviously extendable to any number of splits (`splitResult` will have one element for each split it makes), and different separators (simply change the split argument to be your delimiter).

Discussion

It is important to be aware that sometimes, for whatever reason, the paths passed into Draft may not match your expectations. In that event, it is important to know how each of the above code snippets will behave.

The first example should be fairly robust; in the worst-case scenario, `'shotName'`, `'sequenceName'`, and `'projectName'` will just be blank (or the incorrect value). However, in the second example, if there are not enough results in `'splitResult'`, you might get an `IndexError` when trying to get a value you expect to be there. To make this more robust, you can add a try-except block around this code to prevent it from crashing your Draft template, as follows:

```
#Sample input
directoryName = "sequence_shot"

#Initialize our output variables, in the worst-case they will be empty
sequenceName = ""
shotName = ""
```

```
try:
    splitResult = directoryName.split( "_" ) #split on underscores
    sequenceName = splitResult[0] #first part of the split is the sequence
    shotName = splitResult[1] #second part of the split is the shot
except IndexError:
    pass
```

5.6 Color Operations

This section focuses on color management within Draft and how to manipulate color spaces.

5.6.1 Apply a Gamma Correction

Problem

Your image or video has the wrong gamma. You need to apply a gamma correction.

Solution

Use the *Draft.Image.ApplyGamma()* method:

```
import Draft

inFile = '/path/to/input.exr'
outFile = '/path/to/output.exr'

img = Draft.Image.ReadFromFile( inFile )
img.ApplyGamma( 2.2 )
img.WriteToFile( outFile )
```

Discussion

The gamma correction may be the opposite of what you expect. For example, the 2.2 gamma in our example will make grays darker. If you want the opposite behaviour, please use:

```
img.ApplyGamma( 1.0 / gamma )
```

Where `gamma` is the gamma correction you want to apply.

5.6.2 Create Cineon Images from EXR Images

Problem

You have a linear color .EXR image, and you want to create a Cineon .DPX image from it.

Solution

You can use the following script:

```
import Draft

inFile = '/path/to/input.exr'
outFile = '/path/to/output.dpx'

lut = Draft.LUT.CreateCineon()
img = Draft.Image.ReadFromFile( inFile )
lut.Apply( img )
img.WriteToFile( outFile )
```

Discussion

By default, Draft writes linear DPX files. To create a DPX file with Cineon color, we must create a Cineon LUT and apply it to the image before we *WriteToFile()*. This line creates a Cineon LUT:

```
lut = Draft.LUT.CreateCineon()
```

And this line applies the LUT to our image:

```
lut.Apply( img )
```

5.6.3 Convert a QuickTime's Color Space

Problem

You have a QuickTime movie that is not in the desired color space. For example, the QuickTime file has linear color but you want sRGB instead.

Solution

Use the *Draft.LUT* class:

```
import Draft

inFile = '/path/to/input.mov'
outFile = '/path/to/output.mov'

dec = Draft.VideoDecoder( inFile )
enc = Draft.VideoEncoder( outFile, dec.fps, dec.width, dec.height )

lut = Draft.LUT.CreateSRGB()

img = Draft.Image.CreateImage( 1, 1 )

while dec.DecodeNextFrame( img ):
    lut.Apply( img )
    enc.EncodeNextFrame( img )

enc.FinalizeEncoding()
```

Discussion

In our script above, two lines are repsonsible for performing the color conversion. First, we call *Draft.LUT.CreateSRGB()* to create the LUT. The returned LUT will convert images from Linear color to sRGB.

Next, we call *Apply()* to apply our LUT to every frame in the movie before we encode it into our output movie.

Draft includes several LUTs in addition to sRGB:

LUT	Draft command
Cineon	*Draft.LUT.CreateCineon()*
Alexa V3 Log C	*Draft.LUT.CreateAlexaV3LogC()*
sRGB	*Draft.LUT.CreateSRGB()*
Rec. 709	*Draft.LUT.CreateRec709()*
Gamma correction	*Draft.LUT.CreateGamma()*

See Also

For the inverting a LUT please review *Bake a Color Transform*.

5.6.4 Bake a Color Transform

Problem

You have an image file with an Alexa LogC LUT. You want to bake the Alexa LUT to prepare the files for display.

Solution

Use the *Draft.LUT* class:

```
import Draft

inFile = '/path/to/input.exr'
outFile = '/path/to/output.exr'

img = Draft.Image.ReadFromFile( inFile )

lut = Draft.LUT.CreateAlexaV3LogC().Inverse()
lut.Apply( img )

img.WriteToFile( outFile )
```

Discussion

To create the LUT, we call *Draft.LUT.CreateAlexaV3LogC()*. The resulting LUT will convert an image from Linear to Alexa V3 Log C. This is the opposite of what we want, so we use the *Inverse()* method. This method returns a new LUT that will convert an image from Alexa V3 Log C to Linear, which is what we need.

We apply the LUT to the image:

```
lut.Apply( img )
```

before we *WriteToFile()*.

Draft includes several LUTs in addition to Alexa V3 Log C:

LUT	Draft command
Cineon	*Draft.LUT.CreateCineon()*
Alexa V3 Log C	*Draft.LUT.CreateAlexaV3LogC()*
sRGB	*Draft.LUT.CreateSRGB()*
Rec. 709	*Draft.LUT.CreateRec709()*
Gamma correction	*Draft.LUT.CreateGamma()*

See Also

For the applying a Color Transform please review *Convert a QuickTime's Color Space*.

CHAPTER

SIX

RELEASE NOTES

6.1 Draft 1.6.0

This version of Draft will not work with Deadline 6 or earlier.

6.1.1 What's New

New License File Required

- This build requires a new version 1.6 license. Please contact sales@thinkboxsoftware.com for an updated Draft 1.6 license file.

Compatibility

- Scripts written for previous versions of Draft should still work fine with Draft 1.6.

Licensing

- Improved license checkout process when Draft is used outside of Deadline.

6.1.2 Bug Fixe

- Fixed a bug in Draft.VideoDecoder.DecodeNextFrame(). That functions was not handling file channel mapping correctly.

6.2 Draft 1.5.2

This version of Draft will not work with Deadline 6 or earlier.

6.2.1 What's New

New License File Required

- This build requires a new version 1.5 license. Please contact sales@thinkboxsoftware.com for an updated Draft 1.5 license file.

Compatibility

- Scripts written for previous versions of Draft should still work fine with Draft 1.5.

Images

- Added support for specifying the channel bit depth when writing an image to file.
- Added support for embedding a timecode when writing an image to file.
- Added support for accessing an image in memory as a byte array that is compatible with the QByteArray in PySide.

Videos

- Added support for embedding a timecode when encoding a video.
- Added support for video concatenation.

6.2.2 Bug Fixes

- Fixed a bug that was causing ImageMagick to read image file twice.
- Fixed a bug that was preventing Draft to encode a video with codec DNxHD and format MXF for some frame rates.
- Fixed a bug that was causing some ImageMagick warnings to throw runtime errors.
- Fixed bugs in Draft.Image.RenameChannel() and Draft.Image.SetToColor(). Those functions were not handling file channel mapping correctly.
- Fixed a bug that was causing overflows when assembling big OpenEXR images with the TileAssembler.

6.3 Draft 1.4.3

This version of Draft will not work with Deadline 6 or earlier.

6.3.1 What's New

New License File Required

- This build requires a new version 1.4 license. Please contact sales@thinkboxsoftware.com for an updated Draft 1.4 license file.

Compatibility

- Scripts written for previous versions of Draft should still work fine with Draft 1.4.
- Added support for OS X El Capitan (version 10.11).

Images

- Added support for controlling image compression and quality settings.
- Updated to ImageMagick 6.9.1.

Videos

- Updated to FFmpeg 2.8.

General Improvement

- Writing an image to file with an undefined compression doesn't issue a warning anymore.
- Improved Draft robustness.

6.3.2 Bug Fixes

- Draft was always writing PNG and TIFF files with an alpha channel.
- The TileAssembler was setting final data window with width and height one pixel bigger than necessary, temporarily carrying useless black pixels.
- Successive crops was not giving the correct results (resulting image was shifted).
- Fixed a bug in deepcopy.
- Changed Magick::Color("black") to Magick::Color(0, 0, 0) so Draft won't rely on finding colors.xml.
- Draft was not closing files properly (bug introduced when ffmpeg was updated to version 2.3).
- It was impossible to resize a one pixel image (bug introduced when support for exr windows was added).
- Fixed a bug that prevented Draft Tile Assembler to assemble tiles with empty data window.

6.4 Draft 1.3.2

Draft 1.3.2 is included with Deadline 7.1.0.35.

This version of Draft will not work with Deadline 6 or earlier.

6.4.1 What's New

New License File Required

- This build requires a new version 1.3 license. Please contact sales@thinkboxsoftware.com for an updated Draft 1.3 license file.

Compatibility

- Scripts written for previous versions of Draft should still work fine with Draft 1.3.

EXR Images

- Added support for EXR data and display windows (previously data windows were set to the same size as the display windows).
- Updated to OpenEXR 2.2.0.

LUT Support

- Added ACES 1.0 LUTs to the included ocio-configs folder.
- Improved the robustness of the Draft ASCCDL Reader. The reader can now handle different syntax in its input file.

Draft Tile Assembler

- Added support for assembling big images by exposing a new class in Python called TileAssembler. Most of the logic of an assembly job can now be handled internally.

6.4.2 Bug Fixes

- Fixed a bug when encoding an image with VideoEncoder. The VideoEncoder was applying a bit of scaling to the image.

- Fixed a bug on Mac OS X when encoding with certain dimensions (ie: 640 x 480) was causing a memory error crash.

6.5 Draft 1.2.3

Draft 1.2.3 is included with Deadline 7.0.0.54.

This version of Draft will not work with Deadline 6 or earlier.

6.5.1 What's New

New License File Required

- This build requires a new version 1.2 license. Please contact sales@thinkboxsoftware.com for an updated Draft 1.2 license file.

Compatibility

- Updated Windows builds for compatibility with Deadline 7's installer (updated from VC90 to VC100 runtime library).
- Updated Python compatibility to 2.7.
- Updated simple_slate_h264_burnins_with_proxy.py and simple_slate_h264_with_proxy.py sample scripts so that they work properly with Deadline 7.
- Scripts written for previous versions of Draft should still work fine with Draft 1.2.

Encoding and Decoding Video

- Updated FFmpeg to version 2.3.
- Added support for webm files: vp8 video codec, vorbis audio.

OpenColorIO

- Use config.ocio and ColorSpaces / Roles to create OCIO color processors for color correcting images.
- Create OCIO color processors directly from your favourite LUT files... see http://opencolorio.org/FAQ.html for the full list of LUT formats supported.
- New OCIO lut sample script in the samples directory.

ASC CDL

- A fully standard-compliant implementation of ASC CDL LUTs. (The clamping steps in OCIO's ASC CDL implementation is not currently standard-compliant.)
- New ASC CDL sample script in the samples directory.

Unicode

- Draft now supports unicode filenames and text annotations.
- Note: We need to modify the DraftParamParser.py library so that unicode strings aren't mangled in the Deadline / Draft boundary, but once they're in, Draft handles them properly.

Licensing Improvements

- Draft licences are now more flexible. Most Draft features require only that a license be present. Actual checkout of licensees now happens only while videos are being encoded or decoded.
- "Lost connection to license server" no longer pops up dialog boxes on Windows.

6.5.2 Bug Fixes

- Fixed a crash when encoding movie with audio.
- Improved error messages when trying to open an exr file that isn't there.
- Fixed error messages so they no longer appear as "unidentifiable C++ exception" in Mac OS 10.8.

6.6 Draft 1.1.1

Draft 1.1.1 is included with Deadline 6.2.0.32.

6.6.1 Bug Fixes

- Fixed a Draft.VideoEncoder bug that resulted in crazy framerates when a framerate of 23.976 was requested.
- Fixed a bug where spaces in the PATH environment variable would result in Draft failing to import.

6.7 Draft 1.1.0

Draft 1.1.0 is included with Deadline 6.1.54655.

6.7.1 What's New

New License File Required

- This build requires a new version 1.1 license. Please contact sales@thinkboxsoftware.com for an updated Draft 1.1 license file.

Removed Support for Mac OS X 10.5

- Draft now requires Mac OS X 10.6 or later.

EXR Images

- Can now write tiled EXR files (see "Working with Tiled Images", below).
- Draft now uses ZIPS (single scanline ZIP) compression when writing scanline EXR files. Previously, Draft used ZIP (16-scanline ZIP) compression instead.

Image Channels

- Added operations for working with arbitrary Image channels (see "Working with Image Channels", below).
- Draft.Image.SetChannel(channel, value) will now create the specified channel if it does not already exist.
- Draft no longer adds an 'A' (alpha) channel to all images.
- Now, an image will only have an 'A' channel if there was one in the original image, or if you add one yourself by using Draft.Image.SetChannel('A', 1.0).

Error Messages

- Draft.Image.ReadFromFile() no longer reports warnings as errors.
- Improved error message when attempting to write movie files using Draft.Image.WriteToFile().

6.7.2 Bug Fixes

- Fixed Draft.VideoEncoder crash with some frame sizes.

6.7.3 Working with Tiled Images

Tile settings are controlled using the new Draft.ImageInfo class:

- Draft.ImageInfo can be passed to Draft.Image.ReadFromFile(filename, imageInfo) to retrieve tile settings.
- Draft.ImageInfo can be passed to Draft.Image.WriteToFile(filename, imageInfo) to control the tile settings for the written file.

6.7.4 Working with Image Channels

We added or changed the following Draft.Image methods to work with arbitrary Image channels:

- Draft.Image.CreateImage(width, height, channels)
- Draft.Image.GetChannelNames()
- Draft.Image.HasChannel(channel)
- Draft.Image.Copy(image, left, bottom, channels)
- Draft.Image.RemoveChannel(channel)
- Draft.Image.RenameChannel(oldChannel, newChannel)

For a description of these new methods, see the *Image* class documentation.

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X