"it" - IceMan

The IceMan imaging model is a sophisticated framework for expressing general computations on image data in a variety of representations. IceMan's execution engine employs dependency analysis, lazy-evaluation, and multi-threading to ensure efficient execution of your IceMan scripts. Here are a few features of the IceMan environment:

1. Speed: substantial effort has been expended to optimize operations.
2. Quality: sub-pixel accurate positioning, full floating point (or double!) precision available.
3. Multiple data types.
4. "Lazy" evaluation of image processing expressions: optimized evaluation occurs only when necessary.
5. User-transparent multiprocessing on machines with multiple CPUs.

The IceMan imaging functions are available through the ice python module and images themselves are represented but the python object type ice.Image.

In conventional object-oriented fashion, image operators are methods of the *IceImage* object. To invoke an operation, one must first construct an instance of *IceImage* and then call the method with appropriate arguments. In most cases, the return value is (a reference to) another *IceImage*: this rule is implied in all the documentation in this section. A handful of operators do return non-*IceImage* values, and special mention is made when this is the case.

Overview

To script image processing operations within "it" requires writing expressions using the "ice" Python module.

In IceMan, each image is represented by an object that has a type called IceImage (ice.Image). Images can use any of over one hundred image processing methods IceImage has to transform and create new IceImages.

For example, to add two images together:

```
result = img1.Add(img2)
```

In this example, img1 is added to img2, and the new IceImage is then saved to result.

Note that images in It's Catalog are represented by a type called It3ImageElement. In order to retrieve the underlying IceImage instance, you can call It3ImageElement's GetImage() method. For example, if we are interested in manipulating an image that was just rendered from Maya, we can do the following:

```
elem = it.GetCurrentElement()
img = elem.GetImage()
result = img.Gamma([2.2])
it.AddImage(result)
```

The first line retrieves the current element from the Catalog. The second line retrieves the element's IceImage instance. The third line calls the Gamma() operator on the IceImage instance (note that Gamma() takes a Python list as its input parameter), and saves the new image to result. Finally, we add the new image to the Catalog.

IceMan Image Operators

In conventional object-oriented fashion, image operators are methods of the *IceImage* object. To invoke an operation, one must first construct an instance of *IceImage* and then call the method with appropriate arguments. In most cases, the return value is (a reference to) another *IceImage*: this rule is implied in all the documentation in this section. A handful of operators do return non-*IceImage* values, and special mention is made when this is the case.

The IceMan Imaging Model

The IceMan Coordinate system

Images in IceMan exist conceptually in an infinite two-dimensional plane specified by integer coordinates. Conforming to standard computer and image-processing convention, x increases from left to right, and y from top to bottom.

Images can be combined in various ways and numbers to give rise to new images, which occupy some part of the 2D image plane determined by rules of composition to be defined later in this document.

Internally, operations may be performed with non-integral coordinates: however, the result is always an image occupying a rectangle specified in integer coordinates. Thus a half-pixel move to the right results in an image which is offset correctly by half a pixel from the original image, but the boundaries of the image still lie on the integer grid.

Each image maintains two rectangles, or boxes, that specify its position in the 2D plane. The data box is the region for which there is real allocated memory, and which can be both read from and written to. The request box is typically larger than the data box, and specifies the rectangular region from which data can be read. For example, images by default have a request box of infinite extent, and reading data from anywhere outside the image always results in a constant (zero) value for all channels.

IceMan Data Types

Images have the type of **ice.Image**. Points and boxes are stored as the standard python tuple type.

Point

A point is a pair of integers, or a pair of real numbers held in a tuple. It is typically used to represent Cartesian coordinates, but occasionally as just an array of two values.

Box

A box is a tuple of four numbers, typically used to represent a rectangle in cartesian image coordinates. When used as a rectangle the elements of the list are ordered:

(minimum_x,maximum_x,minimum_y,maximum_y)

This is the same order that the RIB CropWindow call uses and is notably different from prior version of "it" that used (x, y, width, height).

Pixel Type

Four pixel types are supported in IceMan: all have a nominal range of 0-1, but offer various precision and over- and under-range capabilities. At the script level the types are referred by their values from the **ice.constants** module. The various types are described in the table below.

Type	Size (bytes)	Range	Use
ice.constants.FRACTIONAL	1	0-1	Speed
ice.constants.FIXED_POINT	2	-8.0 - 7.9997	Speed/precision
ice.constants.FLOAT	4	*	Precision/speed
ice.constants.DOUBLE	8	*	High precision

*The floating point types have very large machine dependent ranges and are stored as IEEE float and double respectivelyImage pixel types are not directly visible to the scripting interface. Images of various types can be created using the pre-defined enumerated types in the table above, but the few functions that access pixel values directly pass them as lists of numbers

Image

The image object **ice.Image** is the central entity in IceMan. Instances of the Image object are not just images in the conventional sense: they are roots of (possibly unevaluated) image processing expressions, or graphs. For most purposes, however, it is convenient to treat them as simple images, and it is possible to query them for many important characteristics: pixel type, number of channels (or ply) and data- and request-boxes.

The Image object offers a large number of image processing methods, each of which takes zero to three Image instances as input and returns another Image instance. With very few exceptions, methods do not modify the image they are operating on; instead they create and return a new instance.

Internally, image data is stored in uniformly sized tiles. This storage format is not exposed to the script programmer. In general, the internal data never needs to be accessed directly.

Card

A card is not another type but special kind of **ice.Image** that represents a single color sample. A card may be finite or infinite in extent, and is used to represent either an area of constant color or a single pixel. Cards are, for the most part, indistinguishable from images but have a special creation function **i ce.Card()**

IceMan Semantics

Operations on Images have certain semantics that are outlined in this section.

Ownership of Data

When a pixel exists as data stored internal to a particular Image instance, that Image is said to own the pixel. An Image can always refer to data that it owns. Data that is owned by another Image can be read or written by sub-Images, or read (only) via surround behavior.

Sub-images

Sub-Images are special Image objects that do not own any data, but instead reference a sub-rectangle of the data of another Image. The Image that is referenced by a sub-Image is called the parent of that sub-Image, because the relationship between the two can be viewed as a sort of inheritance. Sub-Images can refer to other sub-Images, ad infinitum.

Edge Behavior, Surround and Abyss

It is useful to define what happens when pixels outside the data box of the image are accessed. This is of particular interest for operations that operate on a neighborhood of pixels (filtering, convolution, etc.).The region of an image outside the data box but within the request box is called the abyss. Many possible surround behaviors are conceivable:

- Card surround: accesses outside the image return pixels of a single color.
- Image surround: accesses outside the image return pixels from another image (which might have its own surround behavior).
- Reflect: accesses outside the image return pixels from the image "mirrored" about the appropriate axis.
- Wrap: accesses outside the image are wrapped around.

Only the first two types of surround behavior (card and image surround) are currently supported by IceMan, though the internals are capable of being extended to support the other behaviors.

By default, images have Card surround with infinite extent, and the card contains all zeros.

Images and Cards

Cards are important because they can be used instead of a full-fledged Image as an argument to any operator. Because cards can be used in this way, there is no need for special operators to manipulate images and constants. For example, there is no multiplication operator that takes a constant: there's only one that takes an Image and that serves to multiply constants as well.

Rule of Composition

In any operation that takes more than a single image as input, it is necessary to determine the dimensions and location of the output. This is called the rule of composition and is different (in general) for each operator. For most operators, however, the rule is that the resulting data box is the union of the input data boxes intersected with the intersection of the input request boxes.

Another notion in IceMan that is useful to keep in mind is that each operator produces an output image that is large enough to contain all the pixels that "matter". A blur operation, for example, produces an output image larger than the input by the width of the filter kernel.