IML: Image Manipulation Language

Steven Chaitoff
Zach van Schouwen
Cindy Liao
Eric Hu

Abstract

IML is an image manipulation programming language. With keywords and operators it can be used for a variety of image modifications such as cropping, resizing, shearing, perspective transformations, and other oft-found operations in graphical editing programs. IML incorporates flow-control instructions (conditionals and iterators) allowing automated processing, and simplifying otherwise tedious manipulations. IML also provides user-definable functions alongside built-in ones; we intend that users may create powerful functions in few lines of code.

IML as an interpreted command-line language which takes image files as input. It is isolated from file formats, and allows constructions to be written without the overhead of file processing; that is, IML works with images as raw bitmap data, and the interpreter acts as an interface between raw blocks of pixels and formatted files.

Data Types

The Pixel is the fundament native type. It may be manipulated using built-in operations. Scalar types – integers, doubles, and strings – are implemented loosely and strings may be freely converted (in a style much like Perl), making data manipulation simple within the language. IML supports loose arrays of these data types, and also the Pixel type.

Because images are so fundamental to the aim of the language, two-dimensional arrays of Pixels are treated as "first-class citizens" under the Image data type. Individual Pixel data can then be accessed as in a typical array: `Image[0][0]`, for example. However, Images also have built-in operations that cannot be applied to scalar arrays. Images themselves may be put into arrays.

Pixels may also be used to uniquely represent a (color, gamma) ordered pair. Several of these color values will be built in as language constants, such as `redPx, bluePx, greenPx, whitePx, blackPx, purplePx, orangePx, brownPx, yellowPx, greyPx`

Syntax

IML is not internally constructed as an object-oriented language, but because programmers are often manipulating distinct entities (Image, Pixel), in this document we use a Java-like *Object.member* / *Object.operation* syntax, where only certain members and operations can be applied to certain objects. We firmly associate data and operations with the types to which they are appropriate.

For example:

```
x.opacity = 100
x.red += 20
x.saturation = y.saturation / 2
```

Built-in Operations

The Pixel type implements multiple color models, including RGB and HSV. These values can be manipulated transparently based on the user's preference, and the appropriate color values of the pixel are reliable at all time. Pixels will also store an opacity value, making blending images simpler and more reasonable.

The Image type can be acted on similarly, manipulating the individual pixels in predefined way.

```
im.opacity = 50
im.gradient(redPx, bluePx, 0, 0, 100, 100)
```

The Image type has several predefined operations:

```
crop(amt)  crop(top, bottom, left, right)
crop(vert, horiz)
enlarge(amt)   (increases the canvas size)
scale(amt) scale(x, y)
rotate(dir, amt)
mirror()
shear()
combine(vert, horiz)  (lay 2 images together horizontally or vertically)
desaturate()
```

User Interaction from Interpreter

IML acts as an interpreted language, since our main goal in this project is to ensure that outputted IML programs can process images transparently, without being concerned with format conversion, compression, importing and exporting, and manipulation of binary data.

The interpreter will be written in Java and executed as a command-line tool.

The IML interpreter takes filenames on the command line, which are passed into the program as global variables, e.g.:

```
java iml myscript.iml --portrait img1.jpg img2.png --landscape img3.gif img4.jpg
```

In this case, global array variables `portrait` and `landscape` (specified by the programmer at the preprocessing level) are filled with the processed contents of these files. (`<img1.jpg img2.png>`, `<img3.gif img4.jpg>`, respectively).

Output files may be specified -- if they are not, the interpreter overwrites the input files (backing up based on the VERSION_CONTROL environment variable). The user may specify --sdtin FORMAT or --stdout FORMAT to cause the interpreter to act like a pipe, inputting and outputting data in the specified format (one of PNG, GIF, JPG, BMP, etc.).

Control Structures

We will implement the standard control structures necessary for Turing-completeness, and various native control structures designed for the manipulation of images.

Including:

* `if - else` statements for conditionals
* `for` loops
* `do...while` loops
* `foreach` loops designed for rows, columns, and pixels, as well as a native `foreach` loop for loose types (`foreach_row, foreach_column, foreach_pixel`)

Examples

`Image x` (declares an Image called x)
`Pixel y` (declares an Pixel called y)

Resize an image such that its largest dimension is 100 pixels, whether it is a portrait or landscape image

```
operation constrained_fit: Image x
    if x.width > x.height
        x.width = 100
        x.height = 100 * x.height / x.width
    else
        x.height = 100
        x.width = 100 * x.width / x.height
```

Create an attractive fading visual effect in just a few lines of code

```
operation downwards_fade: Image x
    for_each_row i in x
        for_each_column j in x
            x[i][j].opacity = 1 - i / x.rows
```

Add a colored border around an image

```
operation add_border: Image x
    for_each_column j in x
        x[0][j] = blackPx
        x[x.rows - 1][j] = blackPx
    for_each_row i in x
        x[i][0] = blackPx
        x[i][x.columns - 1] = blackPx
```

Process several images using stacked function calls

```
operation add_border_to_multiple_images: Array(Image) x
    for_each im in x
        im.add_border
```