Glimmer

by Terry Tai and Vida Ha

Introduction

Glimmer is a new and exciting programming language for manipulating images and creating special imaging effects, designed to be easy to learn. Extensive programming experience is not a prerequisite for learning Glimmer; instead, we hope even a novice programmer can quickly pick up Glimmer's commands and dive straight into image modifications. To this end, Glimmer natively supports images as first-order data types, along with a set of basic image operations, from which one can construct advanced filters and effects generators. This language will allow more advanced control logic than that provided by standard image editing software packages such as Adobe Photoshop. Glimmer is not designed to be an advanced language or to be used for all general purposes. While Glimmer has limited support for matrix operations, because image manipulation techniques do rely heavily on matrix mathematics, Glimmer is not the best matrix manipulation language around.

Justification

So what is it that Glimmer provides, really?

Even very experienced programmers will enjoy the low ramp-up time for learning Glimmer and the *ease* with which someone can edit an image in a programming language setting. While stronger languages such as C can handle the parsing of the bitmap and perform the same numeric operations, the looping constructs needed would be clumsy and prone to error. Glimmer has loop mechanisms specifically tailored for images. With Glimmer, it is possible to do a lot more image manipulation with fewer lines of code.

In addition, Glimmer would be a great language to use to introduce programming to younger students who maybe aren't yet ready to learn a more powerful language. There's no type-checking, objects, or other abstract programming paradigms to learn first. And students will appreciate the visual aspect of imagining. A teacher could even use interesting visual patterns to teach elementary algorithms.

Our hope is that users will have fun using Glimmer for image processing and find interesting applications for this language!

Language Features

- User (non-reserved) identifiers start with any alphabet character, and contain alphanumeric characters union {_}
- All function parameters are pass by reference.

Native Types

- `int`
- `float`
- `pixel` – contains a red, green, blue, transparency value for a point in an image
- `image` – contains the pixels in a matrix format along with basic image information

Numerical Operations

- Addition, Subtraction, Multiplication, Division (`+`, `-`, `*`, `/`) for `int's` and `float's`
- Modulus (`%`) for `int's`
- Binary Comparators (>, <, <=, >=, ==, `!=`) return 1 if true, 0 otherwise
- A Set of intrinsic library math functions, like rand - to produce random integers, sine, cosine, etc.

Pixel Operations

- Addition, Subtraction (`+`, `-`): Add/Subtract pixel values
- `a`: get the transparency component of the pixel
- `red`: get the red component of the pixel
- `green`: get the green component of the pixel
- `blue`: get the blue component of the pixel
- Access the surrounding pixels: pixel[-1:-1] means the pixel to the left and up of pixel.

Image Operations

- Addition (+, -): Add/Subtract all pixel values in the image.
- Concatenation of two images: image3 = [image1 image2];
- Overlaying (<<): Placing one image on top of the other.
- Rotating an image (^): flips the image 90 degrees counterclockwise.
- `load`, `save`: read or write an image from or to disk
- `scan_pixels`: iterate over all the pixels of an image (row-major), assigning the row number to the variable `<imagename>.rownum`, the column number to `<image_name>.colnum`, and the pixel itself to `<image_name>.pixel.`
- `scan_rows`: iterate over the rows of an image, assigning the row number to the variable `<imagename>.rownum` and the row itself to `<imagename>.row`
- `scan_cols`: iterate over the columns of an image, assigning the column number to the variable `<imagename>.colnum` and the row itself to `<imagename>.col`

If Statements

- `if, else` statements.

Functions

- Functions can be defined using pass by reference exclusively. Functions can take any number of arguments and don't return anything.

Example Commands

Creating an image bitmap

```
// Load a bitmap from a file.
pregenerated_image = load("/home/vida/image.bmp");
// Manually create a bitmap, where each pixel is explicitly
defined
test_image = [ 0xFF0000, 0x00FF00, 0x0000FF];
// Create a bitmap that is 50 wide and 250 tall where each pixel
is black.
black_image = bitmap(50, 250, 0x000000);
// Create a copy of another bitmap - modifying one will not
modify the other.
pregenerated_image_copy = [pregenerated_image];
// Create an alias to a portion of another bitmap.  (Same place
in memory).
bottom_of_pregenerated_image =
pregenerated_image[height(pregenerated_image)/2:,:];
// Create a new bitmap with two images side by side.
wide_black_image = [black_image black_image];
// Concatenate two images - one on top of the other.
long_black_image = [black_image; black_image];
```

Ways to manipulate an image

```
// Rotates the image one quarter turn counterclockwise.
^test_image;
// Paste one bitmap on top of another starting at pixel 50, 100.
test_image[50:,100:] << front_image;
```

Defining a function

```
function make_white_pixels_red(image) {
  scan_pixels(image) {
    if (image.pixel == "0xFFFFFF") {
      image.pixel = "0xFF0000";
    }
  }
}
```

Example Programs

Converting a color image to grayscale

```
img = load("/home/glimmer/color.bmp", "bmp");
scan_pixels(img) {
  avg = .3 * red(img.pixel) + .59 * green(img.pixel) +
  .11 * blue(img.pixel);
  img.pixel = [avg, avg, avg, a(img.pixel)];
}
save(img, "/home/glimmer/grayscale.bmp", "bmp");
```

Double the width and height of an image

```
expanded_image = bitmap(img.width*2, img.height*2);
scan_pixels(img) {
  expanded_image[2*img.row:, 2*img.col:] + bitmap(2, 2,
img.pixel);
}
```

Compute the 1 - Nth Fibonacci numbers

```
fib_seq = [0, 1, n];
scan_rows(fib_seq) {
   if (fib_seq.row[2,0] > 0) {
     fib_seq = [fib_seq; fib_seq.row[1,0], fib_seq.row[0,0] +
fib_seq.row[1,0], fib_seq.row[2,0] - 1];
   }
}
one_nth_fib_seq = fib_num[2, -1];
```