Pixel

Alex Anthony Cortes-Ose (ac4441)
Dillon Davis (dhd2121)
Jessica Kim (sk4711)
Jessica Peng (jp3864)

1 Introduction

Pixel is a language that is designed to process and manipulate images. The idea for our language was originally inspired by the concept of applying filters to images. On social media platforms such as Snapchat and Instagram, there are pre-set filters that people are able to apply to their photos. Every user has access to these filters, and applying these filters may alter each different photo in its own unique way depending on the state of the original photo.

We wanted to create a language that could not only create these filters, but perform the various tasks which are fundamental to basic image processing. With our language, it is easy to write algorithms that perform enhancement, transformation, edge detection, and basic analysis on images (see section 2, use cases). An *image* is its own primitive type, containing a *pixel* matrix; *pixel* is another primitive type. These types enable extended functionality and make available information useful in many image processing applications.

The syntax for our language draws from Python and JavaScript in several notable ways. The design of Python libraries and its array and matrix manipulation features are crucial for image processing, and we admire the simplicity of Python in this aspect. We were also inspired by the map, forEach, and reduce (which is equivalent to fold_left) methods for lists in JavaScript, the explicit code structure with semicolons and easy scoping with brackets, the language's arrow syntax for anonymous function declaration, and its support for higher order functions.

2 Use Cases

Our language can be used to write and evaluate many fundamental image processing algorithms and can achieve various image enhancements, transformations, and analyses. Some of these include:

- noise removal
- image sharpening
- Thresholding

- color shifting
- Blurring
- Edge and feature detection (texture analysis corners)
- blob and object detection
- contrast enhancement
- Rotation
- Enhanced mapping
- Altering rgb order
- Flipping pixel order (flip image)
- Convert image to black and white
- Convert image to grayscale
- Encrypt image
- Rank filters, thresholding
- Create an RGB representation of a gray-level image.

3 Language Details

3.1 Data Types

Data Type	Description	
`int`	4-byte signed integer type	`integer = 4`
`str`	Array of ASCII characters	`string = “hello world”`
`float`	8-byte floating point number	`fl = 3.5`

`matrix`	Mutable data structure storing multi-dimensions of objects	`mat =`
		`[[1 0 0 1`
		` 0 1 0 1`
		` 0 0 0 0]`
		`[0 1 1 1`
		` 1 0 0 1`
		` 1 0 1 0]]`

`pixel`	A pixel is a special tuple with functions to obtain r, g, b values, double precision color values, and supports indexing. Can be initialized as a single element for	`pix = (120, 70, 54);`
		`r = pix.red;`
		`g = pix.green;`
		`b = pix.blue;`

	grayscale images.	`db_list = pix.doubles; // [0.4705, 0.2745, 0.24]` `pix2 = (106);`

`image`	A specific type of matrix that represents an image consisting of each row as a list of pixels within an outer list	`img =`
		`[`
		
		`[[255,0,0],[0,25,100],[0,56,23]],`
		
		`[[205,12,56],[20,25,0],[82,3,1]],`
		
		`[[0,12,0],[111,25,90],[0,9,120]]`
		`]`

`list`	A standard array, consisting of elements of the same type	`li = [1, 2, 3, 4, 5]`

In our language, a boolean true or false value is represented by the integer values of either 1 or 0, respectively. This enables thresholding and binary image generation, a common image manipulation task.

3.2 Keywords

The following keywords are reserved for this language:

`fun, pixel, matrix, image, int, float, while, for, in, if, else, return, true, false, len, range, str, blob, self, list, void, object`

3.3 Operators

Operator	Operation
`+`	Scalar, matrix-scalar, element-wise matrix addition
`-`	Scalar, matrix-scalar, element-wise matrix subtraction

*	Scalar, matrix-scalar, matrix (dot product) multiplication
/	Scalar, matrix-scalar division
%	Scalar modulo
>, >=, <, <=, ==, !=	Scalar, boolean, matrix-scalar, element-wise matrix comparison
.*	Element-wise matrix multiplication
./	Element-wise matrix division
	Casts a given input into the specified type

Element-wise matrix operators `+`, `-`, `.*`, `./` perform the given operation with every corresponding element between matrices and return a matrix with their results in the same positions. The dot product between two matrices can be done with `*`. Comparison operators return either 1 for true or 0 for false.

3.3 Functions

Function	Description	Return type
`len(s)`	Returns the number of items in an object. `s` may be a string, bytes, tuple, list, dictionary, set, etc.	`int`
`print()`	Prints any given data type to stdout	`void`
`range(int start, int finish, int inc)`	Represents an immutable sequence of numbers from start (inclusive)to finish (exclusive) with an optional increment (default of 1)	`list`
`sum(list l)` OR `sum(matrix m)`	Returns a sum of each of the elements in either the provided list or provided matrix	`int, float`
`int(float f), int(str s)`	Casting a float or string into an int	`int`

`float(int i),` `float(str s)`	Casting an int or string into a float	`image`
`str(int i),` `str(float f),`	Casting an int or float into a string	`str`
`matrix(list l)`	Casting a list into a matrix	`matrix`
`image(matrix m)`	Casting a matrix into an image	`image`
`image_in(str s)`	Converts either a filepath string or base64 encoded string into an image type	`image`

`image_out(image i)`
`OR`
`image_out(matrix m)`
	Converts either an image or matrix into an image on the hard disk	`image`

3.4 Built-in Image Functions

Name	Description	Return Type
`image.metadata`	Returns an object containing relevant metadata for the image datatype (filename, size, file-size, compression type, color type, pixel size)	`object`
`image.matrix()`	Returns matrix of pixels representing image	`matrix`

3.5 Matrix Functionality

Name	Description	Return Type
`transpose()`	Transposes a given matrix	`matrix`
`zeros(int r, int c)`	Creates a matrix of zeros	`matrix`

	with r rows and c columns	
`ones(int r, int c)`	Creates a matrix of ones with r rows and c columns	`matrix`
`identity(int s)`	Creates an identity matrix with s rows and s columns	`matrix`

Name	Description
`matrix``.rows`	An int describing the number of rows of a given matrix
`matrix``.cols`	An int describing the number of columns of a given matrix
`matrix.KERNELS`	An object containing commonly used image processing kernels (various blurs, sharpens, and edge detection matrices), exists as a property of the matrix type

3.6 Features

- Single line comment: `//`
- Multi line (nestable) comment: `/* */`
- Array and matrix indexing is done in the style of numpy array slicing,

```
m = [
        [[10 10 10], [10 10 10], [10 10 10]],
        [[10 10 10], [10 10 10], [10 10 10]],
        [[10 10 10], [10 10 10], [10 10 10]]
   ];

first_and_second_row = m[0:2, :];
```

4 Code Samples

1. Convert an image to grayscale:

```
fun grayscale(image img) {
   m = img.matrix();
   row_size = m.rows;
   column_size = m.cols;

   new_img = matrix.zeroes(row_size, column_size);

   for pixel in m {
      avg = (pixel.r + pixel.g + pixel.b) / 3;
      New_img[pixel.x, pixel.y] = [avg, avg, avg];
   }
   return new_img;
}
```

2. Permute color channels of an image:

```
fun permute_color_channels(image img, list perm) {
   m = img.matrix();
   row_size = m.rows;
   column_size = m.cols;

   new_img = matrix.zeros(row_size, column_size);

   for row in range(row_size) {
      for column in range(column_size) {
          new_img[row, column, perm[0]] = m[column, row, 0];
          new_img[row, column, perm[1]] = m[column, row, 1];
          new_img[row, column, perm[2]] = m[column, row, 2];
      }
   }

   return new_img;
}
```

3. Encrypt an image with a key:

```
fun encrypt(image img, list key) {
   m = img.matrix();
   row_size = m.rows; #instead of matrix.shape[0]
   column_size = m.cols; #instead of matrix.shape[1]
   new_img = matrix.zeros(row_size, column_size);

   for row in range(row_size) {
      for column in range(column_size) {
          r = m[column, row, 0] ^ key[row];
          g = m[column, row, 1] ^ key[row];
          b = m[column, row, 2] ^ key[row];
          new_img[column, row, :] = (r, g, b);
      }
   }

   return new_img;
}
```