Photon - Language Reference Manual

Akira Higaki (abh2171) - Manager
Calum McCartan (cm4114) - System Architecture
Franky Campuzano (fc2608) - Language Guru
Phu Pham (pdp2121) - Tester

February 2021

Contents

1 Introduction and Motivation

Photon is a language that is centered around modifying and editing images, similar to the functionality of Adobe Photoshop. The language is inspired by workflows in the visual effects industry, especially the node-based software Nuke. We aim to be able to provide functionality similar to that of Photoshop through a C-like syntax.

The main feature of the language is that takes advantage of an alpha layer in addition to the red, green, and blue layers to provide an efficient and easy way to combine and edit images and video.

The end goal is to have users upload JPG images (or multiple JPGs for videos) and put it through an automated and efficient editing pipeline through our language to output simple editing procedures quickly.

2 Types

2.1 Primitives

Type	Description
int	An integer
float	A floating-point number
string	A sequence of characters
boolean	True or False values
pint	A special type of Int whose value can only range from 0 to 255
null	Primitive that holds nothing

2.2 Structures

Type	Description
array	A single unit of multiple grouped values
pixel	A group of four Pint values
image	A matrix of pixels

2.3 Conventions

- Type int_to_float(x) convert x to a floating-point number.
- Type float_to_int(x) convert x to an integer.
- Type to_string(x) convert x to a string. Here, x can be of any primitive type.
- print(x) will convert x to string for output.
- Operation on numeric types will cast to the most precise one ($pint \rightarrow int \rightarrow float$).

3 Lexical Conventions

3.1 Identifiers

User-defined variables and functions will use camel case, i.e. flipImage(filePath).

3.2 Keywords

These are the keywords that are reserved by the language for conditional statements and function declarations.

Name	Description
func	Function declaration
return	Followed by a value that is returned to the caller
if	Beginning of conditional statement
elif	Conditional statement
else	Conditional statement
for	Iterative statement
while	Iterative statement

Below are keywords reserved for data types and structures in the language.

Name	Description
int	An integer
float	A floating-point number
string	A sequence of characters
boolean	True or False values
pint	A special type of Int whose value can only range from 0 to 255
null	Primitive that holds nothing
array	A single unit of multiple grouped values
pixel	A group of four Pint values
image	A matrix of pixels

Color aliases are also keywords reserved by the language - which can be referenced using a '_', such as _black

Alias	Value
_black	Pixel(0, 0, 0)
_white	Pixel(255, 255, 255)
_red	Pixel(255, 0, 0)
_green	Pixel(0, 255, 0)
_blue	Pixel(0, 0, 255)
_cyan	Pixel(0, 255, 255)
_magenta	Pixel(255, 0, 255)
_yellow	Pixel(255, 255, 0)

3.3 Operators & Logical Expressions

Type	Description
+	Appends primitives or images (point-wise)
-	Subtracts primitives or images (point-wise)
*	Multiplies primitives; images (point-wise with constants, images)
/	Divides non-zero primitives or images (point-wise)
==, <=, >=, <, >	Compares primitives
[ ]	Array creation and element calling
sqrt(arg)	Square root of a numeric type
dist(arg1, arg2)	Distance of array or image's elements
||	OR operator
&	AND operator
!	NOT operator

3.4 Precedence

Precedence	Operator	Description	Associativity
1	() [] .	Function call Array subscripting Structure and union member access	Left-to-right
2	!	Logical NOT	Right-to-left
3	sqrt()	Square root	Right-to-left
4	* /	Multiplication, division	Left-to-right
5	+ -	Addition and subtraction	Left-to-right
6	dist()	Image Distance	Right-to-left
7	< <= > >=	For relational operators < *and* <= respectively For relational operators > *and* >= respectively	Left-to-right
8	==	For relational =	Left-to-right
9	&	Logical AND	Left-to-right
10	||	Logical OR	Left-to-right
11	=	Simple assignment	Left-to-right

4 Syntax

4.1 Variable Declaration & Assignment

All types of variable can be declared, assigned and reassigned in the same way. A variable cannot be assigned before declaration.

```
<type> <name>;              # Declaration
<name> = <value>;           # Assignment / reassignment
<type> <name> = <value>;    # Declaration & assignment
```

For example...

```
int x = 42;
int y;
bool myBool = true;
y = x + 10;
x = 3;
string greeting = "Hello";
```

All variables declared without assignment will be assigned a default value. 0 for numeric types, false for the boolean type, and null for all other types.

4.2 Naming

Variables and functions must be named any character a-z, followed by any number of characters a-z, digits 0-9, or underscores '_' (Characters may be upper or lower case). Variables and functions may not use reserved words as names.

```
int GoodName123;
float other321_name;
```

4.3 String literals

String literals are any number of ASCII characters enclosed in a pair of single or doubles quotes. Literals enclosed in single quotes may not contain any single quotes, and literals enclosed in double quotes may not contain any double quotes.

```
string x = "hello 'friend'";
string y = '"hello" friend';
```

4.4 String concatenation

Use the + operator to add a string variable to another string variable

```
string x = "hello";
string y = "world";
string z = x+y; # hello world
```

4.5 Comments

Line comments are denoted by the hash character '#'. Hash characters inside string literals are unaffected.

4.6 White space

Comments are terminated by newline characters. Other than this, the language is not sensitive to white space / indentation.

4.7 Pixel

The pixel type can be created with 4, 3, 1, or 0 arguments corresponding to RGBA values.

```
# RGBA = (arg1, arg2, arg3, arg4)
Pixel myWeakGreenPixel = Pixel(0, 255, 0, 32);
# RGBA = (arg1, arg2, arg3, 255)
imgWithABlueCorner[0][0] = Pixel(0, 0, 255);
# RGBA = (arg1, arg1, arg1, 255)
Pixel grey = Pixel(128);
# RGBA = (0, 0, 0, 0)
Pixel blankPixel = Pixel();
```

Pixels can also be created using the aliases specified in the keywords section. Example uses of the aliases are shown below.

```
# Set the pixel at (10, 4) to green
myImage[10][4] = _green;
# Set the red component to 0 for all pixels
imgWithNoRed = myImage - _red;
```

When a binary operator is used with a pixel and an image, the pixel will be treated as if it was an image where all the pixels contain the same value.

4.8 Image

The image type must be created with a width and height.

```
Image img = Image(600, 400);
```

The width and height attribute is accessible.

```
int pixelCount = myImg.width * myImg.height;
```

Pixels in an image can be accessed and set like so.

```
Pixel favouritePixel = myImg[12][18];
myImg[22][27] = Pixel(128, 255, 0);
```

4.8.1 Flipping an Image

Images can be flipped with the built-in function like so:

```
Image myImg = load("myImage.jpg");
myImg = myImg.flip
```

4.8.2 Rotating an Image

Images can be rotated right 90 degrees with the built-in function like so:

```
Image myImg = load("myImage.jpg");
myImg = myImg.rotate; #myImg is now rotated 90 degrees right
myImg = myImg.rotate; #myImg is now upside down
```

4.8.3 Adding Images

Below is the code snippet for adding two images:

```
newImg = img1 + img2;
```

Images are added with the following equation:

```
newImg.pixels[x][y].red =
img1.pixels[x][y].red * img1.pixel[x][y].alpha / 255 +
img2.pixels[x][y].red * img2.pixel[x][y].alpha / 255
```

This operation occurs with all pixels in both images. Adding two images of different size will result in the resulting image being the maximum height and width of the two images, and the addition will be aligned to the top left corner of both images. Keep in mind that the assigned value is a pixel with a maximum value of 255.

4.8.4 Multiplying Images

Below is the code snippet for multiplying two images:

```
newImg = img1 * img2;
```

Images are multiplied with the following equation:

```
newImg.pixel[x][y].red =
img1.pixel[x][y].red * img2.pixel[x][y].red / 255
```

This operation occurs with all pixels in both images. Multiplying two images of different size will result in the resulting image being the maximum height and width of the two images, and the operation will be aligned to the top left corner of both images. Keep in mind that the assigned value is a pixel with a maximum value of 255.

4.8.5 Dividing Images

Below is the code snippet for dividing two images:

```
newImg = img1 / img2
```

Images are divided with the following equation:

```
newImg.pixel[x][y].red =
img1.pixel[x][y].red * 255 / img2.pixel[x][y].red
```

This operation occurs with all pixels in both images. Dividing two images of different size will result in the resulting image being the height and width of the divisor image to avoid dividing by zero, and the operation will be aligned to the top left corner of both images. Keep in mind that the assigned value is a pixel with a maximum value of 255.

4.8.6 Subtracting Images

Below is the code snippet for subtracting two images:

```
newImg = img1 - img2;
```

Images are subtracted with the following equation:

```
newImg.pixel[x][y].red =
img1.pixel[x][y].red * img1.pixel[x][y].alpha / 255 -
img2.pixel[x][y].red * img2.pixel[x][y].alpha / 255
```

This operation occurs with all pixels in both images. Subtracting two images of different size will result in the resulting image being the maximum height and width of the two images, and the subtraction will be aligned to the top left corner of both images. Keep in mind that the assigned value is a pixel with a maximum value of 255.

4.8.7 Loading an Image

A new image is loaded with the built-in function like so:

```
Image myImg = load(filePath);
```

Here, filePath can be either absolute (with root element and complete directory list to the file) or relative to the current directory.

4.8.8 Saving an Image

Modification to an image is saved to a file system with the built-in function like so:

```
Image myImg = load(filePath);
myImg.save("New changes made.");
```

A new image can also be created and saved to a file system containing the new changes with the built-in function *save_as*:

```
Image myImg = load(filePath);
Image newImg = myImg.save_as(filePath, "New changes made.");
```

4.8.9 Retrieving a Pixel

An image can be treated as a 2 dimension array of pixels. A pixel could be accessed as follow:

```
Pixel myPixel = myImg[4][16];
```

4.9 Attributes

Attributes of an object (arrays, Pixel, Image) can be accessed using the name of variable and the name of the attribute separated by a dot.

```
int len = myArray.length;
pint redValue = pixel.red;
```

4.10 Arrays

4.10.1 Initialization

Arrays of any type can be declared in either 1 or 2 dimensions.

```
string[] greetings;      # 1D
float[][] numberGrid;    # 2D
```

Arrays can be created with specified values (where size is implicit), or with a given size (where values are initialised to the type's default value).

```
greetings = ["hello", "hi"];    # Size is implicit
numberGrid = float[100][200];   # Size is explicit
```

4.10.2 Element Retrieval

Array values are assigned and retrieved like so.

```
numberGrid[4][7] = 22;
print(greetings[1]);
```

4.10.3 Length

Array size are retrieved using the length keyword.

```
greetings = ["hello", "hi"];
print(greetings.length); #output 2
```

4.11 Functions

Functions are declared using the following syntax.

```
func <return_type> <name>(<arg1type> <arg1name>) {
    return <value>;
}
```

For example a function can be declared and called like so.

```
func int add(int val1, int val2) {
    return val1 + val2;
}

int sum = add(3, 5);
```

More function examples included in section 6.

4.12 Control Flow

Control flow statements ignore white space. Scope is defined by the usage of {} brackets.

Conditional blocks can be created as shown below.

```
    if(conditional statement is true){
        do this ;
    }

    #using blocks
    if(x>0) {
        print("x is positive");
    }
    elif(x<0) {
        print("x is negative");
    }
    else { print("x equals 0"); }
```

Loops are created as shown below.

```
    for( value initialization; conditional statement;
    optional increment) {
        do this until conditional statement is false
    }
```

```
    while(conditional statement is true) {
        do this until conditional statement is false
    }

    #example of an implemented while loop
    while(x>0) {
        print(x);
        x = x-1;
    }
```

5 Standard Library Functions

Built-in functions in Photon.

5.1 min, max Functions

Min() and max() functions will each take two ints as arguments, and return the smallest/largest value, respectively.

```
    int x = min(int_one, int_two);
    int y = max(int_one, int_two);
```

5.2 Printing

To print in Photon, use print(string). Printing a non-string data type will cast the argument as a string, allowing for anything in Photon to be printed.

```
    print("Hello World!")
    print(int)
    print(string)
    print(array[element])
```

5.3 Destroy

Non-primitive data types can be manually destroyed using destroy(object).

```
    func null destroy(object) {
        /* underlying undefined mechanism */ }
```

5.4 Image Functions

Image functions are built into the language.

```
    Image myimage = Image(600,400);

    Image secondimage = load(filepath);
    secondimage = image.flip;
    secondimage = image.rotate;

    secondimage.save("changes made");
    myimage.save_as(filepath);
```

6 Example Code

6.1 Maximum of an array

This is a simple subroutine that finds the largest element in an array.

```
func int maxElement(inArray) {
    int max = 0;
    for (int i = 0; i < inArray.length; i = i + 1) {
        if (inArray[i] > max) {
            max = inArray[i];
        }
    }
    return max;
}
```

6.2 Example Image Modification Flow

Below is a subroutine that uses the built-in functions to modifying an image. This demonstrates loading/saving in an image.

```
func string flipAndRotateImage(filePath) {
    image testImage = load(filePath);
    testImage = testImage.flip;
    testImage = testImage.rotate;
    returnPath = filePath + "_new";
    Image newImage =  testImage.save_as(returnPath,
    "modified image");
    return returnPath;
}
```

6.3 Image addition

Below is a subroutine that uses the alpha values to combine two images together.

```
func image halfHalf(image1, image2) { # alphas = 255 by
    default
    for (int i = 0; i <= image1.width/2; i = i + 1) {
        for (int j = 0; j <= image1.height; j = j + 1) {
            image1[i][j].alpha = 0;
            image2[i+image1.width/2][j].alpha = 0;
        }
    }
    image3 = image1 + image2;
    return image3; #left side of image1, right side of
    image 2
}
```

7 References

VSCOde - Fall 2018 Project
Coral - Fall 2018 Project
Nuke - Video Editing Software