COMS W4115 Programming Languages and Translators

Pixelish Language Reference Manual

Jahyun Kim
jk3111@columbia.edu

Contents

1 Introduction

Pixelish, Pixel + -ish from English, is a simplified image processing language which specifically deals with pixels of image. Pixelish provides a built-in image data type that can be used to easily manipulate the pixels of an image to realize various effects such as turning into grayscale, creating a filter effect similar to that of Instagram, vignetting, and warping of shapes.

2 Lexical Conventions

Identifiers, keywords, separators, operators, string literals, and constants consist the tokens of a Pixelish program. Whitespaces, tabs, and newlines are ignored unless they are served as separators. Comments are also ignored.

2.1 Line Terminator

A semi-colon is used as a line terminator (;).

2.2 Tokens

There are six types of tokens: identifiers, keywords, separators, operators, string literals, and constants.

2.3 Comments

Only a multiple-line comment is supported. A comment can be written inside the block that begins with /* and ends with */. Nested comments are not supported.

2.4 Identifiers

An identifier is a sequence of alphanumeric characters including the underscore. The leading term of an identifier cannot be a number. The upper case letters are differentiated from the lower case letters.

2.5 Keywords

The list of keywords:

image	int	float
string	array	function
true	false	bool
for	if	else
while	break	continue
height	width	hue
sat	val	

2.6 Separators

Tokens are separated by whitespace, tab, and newline. Expressions are separated by comma (,) and semi-colon (;).

2.7 Operators

The list of operators:

```
+     -     *     /
<     >     ==    =
>=    <=    !     !=
.     [     ]
(     )     ||    &&
```

2.8 String Literals

String literals are inside double quotes (“”).

2.9 Constants

Pixelish supports constants for three primitive types.

2.9.1 Integer

An integer constant is a sequence of digits between 0 and 9.

2.9.2 Float

A float constant is a sequence of digits between 0 and 9 and a ‘.’ between the characters or at the end. A sequence after ‘.’ indicates the fraction part. The fraction part can also be stated with the exponent symbol ‘e’ or ‘E’.

2.9.3 Hexadecimal

A hexadecimal constant is a sequence of hexadecimal digits prefaced by #. A hexadecimal value can also be represented as a string literal without # at front.

```
   #000000
or “000000”
```

3 Types

Pixelish supports two categories of types: primitive and object.

3.1 Primitive Types

Primitive types of the language are: int, float, and bool.

3.2 Object Types

Object types contain additional properties and metadata. The primary object type offered by Pixelish is image.

3.1.1 Image

Image is the object of two-dimensional image. This is similar to Mat in OpenCV. Image saves the pixels of the input image in a two-dimensional array. Image saves the height and width of the given input as its properties. Depending on the color space, the image object has three channels: either red, blue, and green, or hue, saturation, and value. By default, when an image object is created by imread function, the object is in rgb-color space. Conversions from rgb to hsv and hsv to rgb can be easily done by built-in functions such as toRGB() and toHSV associated with the image object.

```
image example;

/* Read the image file. */
example = imread(“./image2.jpg”);

/* Check if the image is in RGB space.
  * This returns true if in RGB, false otherwise.
  */
example.isRGB();

/* Check if the image is in HSV space.
  * This returns true if in HSV, false otherwise.
  */
example.isHSV();

/* Access red, blue, and green channels. */
/* Increase the Red level by 50% */
example.red = example.red*1.5;

/*
  * Increase the Blue level by 10.
  * If the resulting value is greater than 256, the error message will appear.
  */
 example.blue = example.blue + 10;

/* Convert RGB to HSV */
example.toHSV();

/* Convert HSV to RGB */
 example.toRGB();
```

4 Names

4.1 Variable Names

A previously unused identifier can be used as a variable name when the identifier is placed on the left for the assignment operation.

4.2 Function Names

A previously unused identifier can be used as a function name when a function is newly declared.

5 Expressions

The primary expression is in the form of identifier, string literal, constant, or function call.

5.1 Function Call
Function Call expression is used to call either a built-in function or user-defined function.

<*identifier*> (<*function-parameters*>)

5.2 Additive Operators
Additive operators are '+' and '-' for addition and subtraction. These are binary operators that take two primitive types such as int or float and return the result of the same type. If one of the arguments is in float, the result is in float.

<*expression*> + <*expression*>
<*expression*> - <*expression*>

5.3 Multiplicative Operators
Multiplicative operators are '*' and '/' for multiplication and division. These are binary operators that take two primitive types such as int or float and return the result of the same type. If one of the arguments is in float, the result is in float.

<*expression*> * <*expression*>
<*expression*> / <*expression*>

5.4 Relational Operators
Relational operators are '<', '>', '<=', and '>=' for comparisons. These are binary operators that take two expressions and return the result of the comparisons.

<*expression*> < <*expression*>
<*expression*> > <*expression*>
<*expression*> <= <*expression*>
<*expression*> >= <*expression*>

5.5 Logical Operators
Logical operators are '!', '&&', and '||'.

5.5.1 Not (!)
Not operator negates the evaluation of a Boolean expression.

!<*Boolean-expression*>

5.5.2 And and Or (&& and ||)
Logical operators are '!', '&&', and '||' for Boolean comparisons. These are binary operators that take two Boolean expressions and return true or false based on evaluations on both expressions.

<*Boolean-expression*> && <*Boolean-expression*>
<*Boolean-expression*> || <*Boolean-expression*>

5.6 Equality Operators

Equality operators are '==' and '!='. These are binary operators that compare the operands for equality. '=='returns true if two expressions are equal to each other, and false otherwise. '!=' returns true if two expressions are not equal to each other.

<*expression*> == <*expression*>
<*expression*> != <*expression*>

5.7 Access Operators

Access operators are '[]' and '.'.

5.7.1 [and]

[and] operators are used for accessing each element in an array or 2D array.

<*array-name*>[*integer-for-array-position*]
<*2D-array-name*>[*integer-for-array-position*][*integer-for-array-position*]

5.7.2 .

A '.' operator is used to access the fields or functions inside object type (image). This operator can be paired up with (and).

<*image-type-name*>.<*field-name*>
<*image-type-name*>.<*function-name*>
<*image-type-name*>.(*width-position-for-pixel*, *height-position-for-pixel*)

5.8 Assignment

Assignment operator is '='. There must be a variable on the left side of the assignment operator, which can be modified. The assignment operator is right associative and returns the result of the expression on the right side of the operator to the left side variable.

<*variable-name*> = <*expression*>

5.9 Operator Precedence

Operators	Associativity
[] () .	Left to Right
!	Right to Left
* /	Left to Right
+ -	Left to Right
< > <= >=	Left to Right
== !=	Left to Right
&& ||	Left to Right

6 Statements

6.1 Block Statements
A sequence of statements can be grouped into a block of statements. A function is a named block statement with a given input.

```
{
   <statement 1>
   …
   <statement n>
}
```

6.2 Expression Statements
An expression statement is an expression followed by a semi-colon.

<expression>;

6.3 If-Else Statements
If-Else statements are conditional statements.

if (*<conditional-expression>*) *<statement>*
if (*<conditional-expression>*) { *<statement-list>* }
if (*<conditional-expression>*) { *<statement-list>* } else { *<statement-list>*}

6.4 For Statements
Pixelish supports for-loops.

for(*<expression>*; *<conditional-expression>*; *<expression>*) *<statement>*
for(*<expression>*; *<conditional-expression>*; *<expression>*) { *<statement-list>* }

6.5 While Statements
Pixelish supports while-loops.

while (*<conditional-expression>*) *<statement>*
while (*<conditional-expression>*) { *<statement-list>* }

6.6 Break Statement
Break statement breaks out from the inner loop.

break;

6.7 Continue Statement
Continue statement skips to the next iteration of the inner loop.

continue;

6.8 Variable Declaration
Variable can be either declared on its own or accompanied with assignment.

<type> *<variable-name>*;
<type> *<variable-name>* = *<expression>*;

6.9 Function Declaration
Function can be declared as follows:

```
function <function-name> (<function-parameters>) {
   <statement 1>
   …
   <statement n>
}
```

Function parameters are separated by comma (,). There is no return type in Pixelish. Instead of copying the values of input parameters, Pixelish directly modifies the values if variables are passed on as parameters.

7 Program

There must be a main function for a program. The main function does not take any arguments.

```
function main() {

   <statement 1>
   …
   <statement n >

}
```