ArtC++

Archit Ganvir
CS21BTECH11005

Kushagra Gupta
CS21BTECH11033

Rahul Ramachandran
CS21BTECH11049

Suryaansh Jain
CS21BTECH11057

Contents

1 Introduction

ArtC++ is a unique image manipulation language that aims to make image editing fun and easy. It borrows inspiration from several languages and libraries like C/C++, Java, Python, NumPy, and Pillow. While it uses types, it supports several inbuilt functions, and behaves similarly to functional programming languages. In addition to this, ArtC++ supports several unique features, like printing images to the terminal. The language can also support basic video manipulation, and can play frames of the video in the terminal.

ArtC++ is compiled to C++, ensuring that the code is fast and efficient.

2 Lexical Specifications

2.1 Keywords

img	gray_img	num	real	and
or	spool	presuming	otherwise	break
continue	return	ink	void	bool
true	false			

2.2 Identifiers

As in several other languages, identifiers begin with a letter or underscore, and are followed by a sequence of letters, underscores or digits. Identifiers are case-sensitive. For example :-

```
    abcdef
    _23n32_
    343491h # Invalid identifier
    *()_  # Invalid identifier
```

2.3 Comments

Single-line comments start with a # symbol. For example :-

```
    # This is a single-line comment
```

Multi-line comments are enclosed by #∗ and ∗#. For example :-

```
    #* This
    is a
    multi-line comment *#
```

2.4 Whitespace

As in C++, whitespaces (including tabs and spaces) is ignored. It is however recommended to include them to improve readability. The newline character is however used to define statements, like in Python. Curly braces {} are used to define scope.

3 Datatypes

Data Type	Description
	Primitive
`img`	multicolor image
`gray_img`	grayscale image
`num`	`integer`
`real`	`float`
`bool`	`bool`
	Derived
`vid`	video, an array of images with a fps
`gray_vid`	grayscale video, an array of gray_images with a fps
`num[]`	array of integers
`real[]`	array of real numbers

4 Operators

Operator	Example	Description
Binary Arithmetic		
`+`	`a + b`	Here, `a` and `b` are either numbers (or reals), or have to be of the same image datatype (RGB or Grayscale). The image shapes have to broadcastable. If a value exceeds 255, it is clipped back to 255. In addition to this, `a` and `b` can be videos (of the same datatype and fps). If this is done, the frames are concatenated.
`-`	`a - b`	The operands are the same as for `+`. If a pixel value goes below 0, it is clipped back to 0.
`*`	`a * b`	The operands and behaviour are similar to `+`
`@`	`a @ b`	`@` is used for convolutions. `a` is an image or a grayscale image. `b` can be a image array of size 3 (this outputs an RGB image) or a single image (this outputs a grayscale image). The number of channels (color dimensions) in the right operand should equal the channels in the left operand.
`^`	`a ^ b`	`^` is used for XORing two images.
Unary Arithmetic		
`~`	`~a`	`a` is an image or a video. This will invert the colors of the image or the video.
`--`	`a--`	`a` is a number or a real. The post-decrement operator will decrease the value of `a`.
`++`	`a++`	`a` is a number or a real. The post-increment operator will increase the value of `a`.
Relational and Logical		
>, >=	a > b, a >= b	The greater-than and greater-than or equal-to operate on numbers (or reals), and are defined as in other programming languages
<, <=	a < b, a <= b	The less-than and less-than or equal-to operators also operate on numbers or reals
`and`	P_1 `and` P_2	Here, P_1 and P_2 are predicates, and equate to true or false.
`or`	P_1 `or` P_2	The `or` operator also operates on two predicates.
!	!P_1	`!` flips the truth value of the predicate it operates on
!=, ==	P_1 == P_2, P_1 != P_2	== and != are the standard equality and not-equality operators.

5 Statements

All statements in ArtC++ end with a newline ('\n') character. This keeps the code clean, and ensures that every statement is on a separate line.

5.1 Declarations

- Numbers:
  - `num num_name = value`
    - Here, `value` is a 32 bit integer.
  - `real real_name = value`
    - Here, `value` is a 32 bit floating point value.
  - `num[] array_name = []`
    - Declaring an array results in `array_name` being associated with an empty list. The list can be populated with `nums` using operators and methods like `.append()` (see below).
  - `real[] array_name = []`
    - Declaring an array results in `array_name` being associated with an empty list. The list can be populated with `reals` using operators and methods like `.append()` (see below).
- RGB Images:
  - `img img_name<h, w>`
    - Here, `h` is the height of the image and `w` is the width of the image.
    - Declaring an image automatically allocates memory for the image, which is stored as a 3D array (height, width, 3). Each element is initialized to 0 (the image is initially black).
  - `img img_name<h, w, hex_code>`
    - This declaration is similar to the previous one, except for the hex-code. The format of the hex-code is `'0xRRGGBB'`, where `RR`, `GG`, and `BB` are hexadecimal numbers representing the red, green, and blue components of the color respectively. Every pixel in the image is initialized to this color.
  - `img img_name<'./img.bmp'>`
    - Here, `'./img.bmp'` is the path to the image file. (Only '.bmp' files are supported)
    - Declaring an image automatically allocates memory for the image, which is stored as a 3D array (height, width, 3). The corresponding RGB values of the image are stored.
  - `img img_name = expr`
    - This combines a declaration and an initialization. `expr` is an expression that evaluates to an image, and is assigned to `img_name`.
- Grayscale Images: `gray_img img_name<h, w>` Grayscale images are stored as 2D arrays (height, width). The declaration is similar to RGB images.
  - `gray_img img_name<h, w>`
  - `gray_img img_name<h, w, c>` where `c` is a value from 0 to 255.
  - `gray_img img_name<'./img.bmp'>`
  - `gray_img img_name = expr`

- RGB Videos (Image Arrays): `vid vid_name<h, w, f>` or `vid vid_name<h, w>`
  - Here, `h` is the height of each frame, `w` is the width of each frame, and `f` is the frame rate in frames per second (fps). It is optional to specify the frame rate, and the default value is 30.
  - Declaring a video results in `vid_name` being associated with an empty list. The list can be populated with images using operators and methods like `.append()` (see below).
- Grayscale Videos (Grayscale Image Arrays): `gray_vid vid_name<h, w, f>` or `gray_vid vid_name<h, w>`
  - This is identical to the declaration of RGB videos, except that each frame is stored as a 2D array (height, width). As before, each element is initialized to 0.

5.2 Expressions

Expressions (The `RHS` of a statement) must always evaluate to a type that is castable to the `LHS`. `real` and `num` are castable to each other and are castable to `bool` but `bool` cannot to typecasted to `num` or `real`. (`gray_img` and `img`), (`gray_vid` and `vid`) are also castable to each other. When an `img` is casted to a `gray_img`, it is converted to grayscale. Conversely, when a `gray_img` is casted to an `img`, the single color dimension is replicated. Examples of valid expressions are:

```
# a,b,c are images/gray_images
a = b + c
b = (a + c)/2
a = a @ b * ~c #** if one operand of @ is gray and the other is RGB, casting to RGB
  takes place *#

a = a or a and b # a, b are real numbers
b = a++
a[2] = b[5] - i*i  #** a and b are arrays of nums/reals and i is a num/real. This
  also works with videos *#
a.append(b[5])  # a and b are arrays of nums/reals
a = a >= b # a gets the value 1 or 0
```

Some invalid expressions are:

```
a = b + c # a is a vid but b is an img
a = a @ b * ~c # b has 2 frames

b = a++ # a is a bool
a[2] = b[5] - i*i  # b is a vid and a in a num/real array
a.append(b[5])  # a is an img
```

5.3 Loops : spool

We have implemented only one type of loop that incorporates both `for` and `while` loops, using the keyword `spool`. The syntax is as follows:-

Type 1

```
# Similar to for loops in C++
spool (initialization expression; condition expression; update expression)
{
    <body of the loop>
}

```

```
# Example
spool (num i = 0; i < 10; i++)
{
    <body of the loop>
}
```

Type 2

```
spool (predicate)
{
    statements
}

# Example
spool (i < 10)
{
    statements
}
```

5.4 Conditionals

Conditional statements will follow the following syntax:-

```
presuming (expression)
{
    statements
}
otherwise presuming (expression)
{
    statements
}
otherwise
{
    statements
}
```

The "presuming otherwise" and "otherwise" parts of the conditional statement are optional.

6 Programming Constructs

6.1 Functions

6.1.1 Image Manipulation

Method	Example	Description
`blur()`	`a.blur()`	Adds blur to the image (or blurs every frame of the video)
`noise()`	`a.noise()`	Adds random gaussian noise to the image (or adds noise to every frame of the video)
`gray()`	`a.gray()`	Converts the image to grayscale and changes the shape (or convert the video to grayscale. Only works on RGB images or videos)
`vflip()`, `hflip()`	`a.vflip()` or `a.hflip()`	Flips the image vertically or horizontally (or every frame of the video)
`get()`, `set()`	`a.get(x, y, c)` or `a.set(x, y, color_hex)`	Gets the pixel value at (x, y, c) or sets the pixel value at (x, y) to the specified color

6.1.2 Displaying Images and Videos

Method	Example	Description
`paint()`	`a.paint()`	Displays the image `a` in the terminal
`frame()`	`a.frame('img.bmp')`	Saves the image `a` to the specified path (only .bmp)
`play()`	`a.play()`	Plays the video `a` in the terminal (resolution might be lowered)

6.1.3 Other Methods

Method	Example	Description
`len()`	`a.len()`	Returns the number of frames in video `a`
`append()`	`a.append(b)`	Appends image `b` or video `b` to the video `a`. The types should match-up
`height()`	`a.height()`	Returns the height of an image or the images in video `a`
`width()`	`a.width()`	Returns the width of an image or the images in video `a`

6.1.4 Drawing on Images

`draw()` is a method that can be used to draw shapes on images. It supports the following shapes :-

```
# Draws circle with center (x\_c, y\_c) and radius r on image 'a'
a.draw('circle', x_c, y_c, r, color_hex)

# Draws a line from (x1, y1) to (x2, y2) on image 'a'
a.draw('line', x1, y1, x2, y2, color_hex)

# Draws a rectangle with center (x\_c, y\_c) and dimensions h and w
a.draw('rectangle', x_center, y_center, h, w, color_hex)

# Draws a triangle with vertices (x1, y1), (x2, y2), (x3, y3)
```

```
a.draw('triangle', x1, y1, x2, y2, x3, y3, color_hex)

# Draws an ellipse with center (x\_c, y\_c) and radii a and b
a.draw('ellipse', x_center, y_center, a, b, color_hex)
```

6.1.5 User-defined Functions

The programmer must define a `ink main()->void` function. All user defined functions must have a `return` statement in them. They must also only return an object of the declared return type.

```
ink func_name (parameters)->return_type
{
    statements
}
```

Functions are provided with a comma separated list of parameters, just like in C/C++

```
ink func_name (img i1, num a1, vid v1)->return_type
{
    statements
}

```

6.2 Slicing

We allow the slicing of videos. This returns a new video with the specified frames. The syntax is as follows :-

```
a = b[1:10] # This returns a video with frames 1 to 9 of b
a = b[1:10:2] # This returns a video with frames 1, 3, 5, 7, 9 of b
a = b[1:] # This returns a video with frames 1 to the end of b
```

7 Sample Programs

Example 1

```

ink bar (img p, img q, img r)->img {
  num h = p.height()
  num w = p.width()
  img s <h, w>
  num a = 2
  real b = 7.8

  presuming (a + b > 10) {
    s = p * (q + r)
  } otherwise presuming (a + b == 10) {
    s = q * (p + r)
  } otherwise {
    s = r * (p + q)
  }

  return s
}

ink func(img i1, img i2, img i3)->img {
  num i

```

```
    spool (i = 0; i < 10; i++) {
      i1 = i1 + i2 * i3
    }

    return i1
  }

  ink main()->void {
      img i1.load('./i1.bmp')
      img i2.load('./i2.bmp')
      img i3.load('./i3.bmp')

      i2.draw('rectangle', 50, 40, 50, 40, 0xff0000)
      i2.draw('triangle', 0, 0, 5, 5, 3, 4, 0x00ffff)

      i3 = func(bar(i1, i2, i3), i1, i2)

      i3.frame('./i_mod.bmp')

      i3.paint()
      return
}
```

Example 2

```
  ink foo(img a, img b, img c)->img {
   img res = a*b + c
   return res
  }

  ink foo1(img a, img b)->img {
      return b*a.blur()
  }

  ink main()->void {
      img i1.load('./i1.bmp')
      img i2.load('./i2.bmp')
      img i3.load('./i3.bmp')

      i3 = foo1(foo(i1, i2, i3), i1)

      i2.draw('circle', 50, 40, 20)
      i2.draw('line', 0, 0, 2, 3)

      i2.paint() # displays i2 in the terminal
      i2.frame('./i2_mod.bmp')

      i3.frame('./i_res.bmp')

      i3.paint() # displays i3 in the terminal
      return
  }
```