Shiva Kernel Language Reference

Cyrille Berger

May 30, 2013

Abstract

This document describes the Shiva Language, which is a kernel-based image processing language.

Contents

Chapter 1

Introduction

1.1 Goal of Shiva

The goal of Shiva is to provide a programing language for graphical effects, that takes as input multiple pixels coming from multiple images and return one pixel. This type of

Part I

Language Reference

Chapter 2

Kernel

2.1 Structure

```
1  <
2  ...
3  >
4  kernel MyKernel
5  {
6    parameter float firstParameter;
7    parameter int secondParameter;
8    ...
9    void evaluatePixel(input image source1, input image source2,
10                      ..., output pixel result)
11   {
12     ...
13   }
14   region needed(region output_region, int input_index,
15                 region input_DOD[])
16   {
17     ...
18   }
19 }
```

The `< ... >` is the metadata associated with a `kernel` and is optional.

While the order of functions in the kernel is not important, if you want to use a function or a constant it needs to be define before its use. For instance the `needed` and `changed` functions can use the name of source of image to know the index of the image in the list of parameters, this require the `evaluatePixel` function to have been written before.

2.2 Functions

2.2.1 evaluatePixel

```
1  void evaluatePixel(input image source1, input image source2,
2                     ..., output pixel result)
```

```
{
  ...
}
```

2.2.2 needed

```
region needed(region output_region, int input_index,
              region input_DOD[])
{
  ...
}
```

The `needed` function indicates the Shiva interpreter which pixel in the input images are needed. For instance, if you program is a gaussian blur, it needs an extra pixel around the operated region.

`region output_region`	the region that will be operated by the kernel
`int input_index`	the index of the image for which we want to know the region that will be needed
`region input_DOD[]`	the list of input

Exemple for a gaussian blur:

```
region needed(region output_region, int input_index,
              region input_DOD[])
{
  output_region.outset(1,1)
}
```

2.2.3 changed

The `changed` function indicates the Shiva interpreter which pixels in the output images need to be recomputed when a given region of the image has been changed. As a special note, the changed function is usually called at the first run to compute the region of the output image.

```
region changed( region changed_input_region, int input_index,
                region input_DOD[])
{
  ...
}
```

For instance, for a kernel that apply a translation of $10 pixels$ on x and $15 pixels$ on y, the following `changed` function:

```
region changed( region changed_input_region, int input_index,
                region input_DOD[])
{
  return { changed_input_region.left + 10,
           changed_input_region.top + 15,
           changed_input_region.width,
           changed_input_region.height };
}
```

2.2.4 generated

The `generated` function indicate the Shiva interpreter which pixels are changed when this generator is applied on the image. Even if images are expected to be of infinite size, some generators can have a more limited area of effect.

```
region generated()
{
  ...
}
```

2.2.5 runTest

This function is used for `kernel` which are used for automatic testing of the language, and of the interpreter. It's a function that doesn't take any parameter and return the number of failed tests.

Exemple:

```
int runTest()
{
  int count = 0;
  if( (1+1) != 2) ++count;
  return count;
}
```

2.3 Metadata

A `kernel` can contains metadata before the its declaration. Metadata is divided in optional sections.

- `version` indicates the version of the Shiva spec used for this `kernel`, the current version is *0*
- `info` this section contains various information about the `kernel`, author, vendor, license... and it is basically a list of key / value.
- `parameters` this section allow to define parameters that can be adjusted by the user and will be given to the kernel as constant parameters before compilation.

Example of metadata:

```
<
  version: 0;
  category: <
    label: Miscaelenous;
    key: misc;
    description: Miscaelenous kernels;
  >;
  info: <
    author: "Joe Doe; Joe Doe Jr";
    vendor: <
```

```
11          name: "DoeGraphics";
12          address: "1242 Main Street";
13        >;
14        license: "LGPLv2+";
15      >;
16      parameters: <
17        param1: <
18          type: int;
19          label: "Param 1";
20          minValue: 0;
21          maxValue: 100;
22          defaultValue: 50;
23          description: "This is the first parameter";
24        >;
25        category2: <
26          description: "This is a category of parameters";
27          param2: <
28            label: "Param 2";
29            type: curve;
30            defaultValue: {{0,0},{1,1}};
31          >;
32          param3: <
33            label: "Param 3";
34            type: color;
35            defaultValue: {1,0,0};
36          >;
37        >;
38      >;
39    >
```

2.3.1 category

The *category* group of the metadata is used by kernel collection management system to group together a set of kernel. All kernels with the same key are assumed to be in the same category, even if the *label* and *description* fields are different, those two are only suggestion, it is left to the collection management system to pick the *label* and *description*.

If no category is specified, the kernel is assumed to belong to the `misc` category.

List of suggested default category, with labels and descriptions:

key	label	description
misc	Miscellaneous	Miscellaneous kernels

2.3.2 info

2.3.3 parameters

Supported types of parameters:

- `int` integer value, if no range is specified it is defaulted to $[0, 100]$
- `float` float value (32bits), if no range is specified it is defaulted to $[0.0, 1.0]$

- `curve` (mapped to `float[][2]`) this allow to pass a curve to a `kernel`, values are expected to between $(0, 0)$ and $(1, 1)$.
- `color` a three components vector of float representing a color

Chapter 3

library

Libraries are used to write reusable code for Shiva Kernel, if you have function that you use often, you might want to put them in a library.

Libraries are started with the `library` keyword and can contain constants, and functions:

```
1 library MyLibrary {
2   const int value = 2;
3   int addValue(int v )
4   {
5     return v + value;
6   }
7 }
```

The keyword `import` allows to use a library in a kernel:

```
1   import mylibrary;
2   ...
3   int v = 3;
4   v = MyLibrary::addValue( v );
```

Chapter 4

Types

4.1 Primitive types

4.2 Arrays

4.3 Matrixes

4.4 Vectors

4.4.1 Vectors types

The four scalar types `bool`, `int`, `float` and `pixel` are declined in three sizes of vector, from two elements to four : `bool2`, `bool3`, `bool4`, `int2`, `int3`, `int4`, `float2`, `float3`, `float4`, `pixel2`, `pixel3` and `pixel4`.

There is also `booln`, `intn` and `floatn` whose length is decided at compile time.

4.4.2 Vectors initialisation

Vectors are initialized using a list of elements `{ first element, second element, ...}`. For instance:

```
1  float3 v = { 1.0, 2.0, 3.0 };
```

Alternatively vectors can be initilized using their constructor:

```
1  typeN::typeN(el0, el1, ..., elN)
```

For instance:

```
1  float3 v( 1.0, 2.0, 3.0 );
2  float3 v;
3  v = float3(1.0, 2.0, 3.0);
```

4.4.3 Access vectors elements

Vector elements can be access using the classical subscript operator `[]`. For instance:

```
1 float3 v( 1.0, 2.0, 3.0 );
2 float v0 = v[0];
```

The elements of the vector can be access as named elements using the following sequences :

- r g b a
- x y z w
- s t p q

4.4.4 Specific members for floating points vector

length This function return the length of the vector:

```
1 floatN floatN::length( )
```

For instance:

```
1 float2 float2::length( )
2 {
3   return sqrt( this[0] * this[0] + this[1] * this[1]);
4 }
```

normalize This function return the normalized version of the vector:

```
1 floatN floatN::normalize( )
2 {
3  return this / this.length();
4 }
```

4.5 Complex types

4.5.1 Images types

Shiva defines five specific image types:

- `image1` one channel image
- `image2` two channels image
- `image3` three channels image
- `image4` four channels image
- `image5` five channels image

And the generic image type `image` whose number of channels is defined at compile time.

Members

sampleNearest and sampleLinear Those two functions are used to access the pixel value in the image:

```
1 void imageN::sampleNearest(float2 coord)
2 void imageN::sampleLinear(float2 coord)
```

`sampleNearest` return the value of the pixel whose coordinates are the closest to the coordinates given as argument. This function does no interpolation of the pixel data. While `sampleLinear` performs a bilinear interpolation.

For instance the following function will copy the source image into the output:

```
1 kernel Copy
2 {
3   void evaluatePixel(input image source, output pixel result)
4   {
5     result = source.sampleNearest( result.coord() );
6   }
7 }
```

For instance the following function will translate the source image by $(-4, -2)$:

```
1 kernel Translate
2 {
3   void evaluatePixel(input image source, output pixel result)
4   {
5     result = source.sampleNearest( result.coord() + float2(4,2) );
6   }
7 }
```

4.5.2 Pixeles types

Shiva defines five specific pixel types:

- `pixel1` one channel pixel
- `pixel2` two channels pixel
- `pixel3` three channels pixel
- `pixel4` four channels pixel
- `pixel5` five channels pixel

And the generic pixel type `pixel` whose number of channels is defined at compile type.

`pixelN` can be cast to the corresponding vector of `float`.

Members

data Give access to the vector containing the data.

coord Give access to the coordinates of the pixel, the type of this field is `float2`.

setAlpha The `setAlpha` function allows to set the current alpha channel of a pixel. It takes a `float` as argument.

When the image doesn't have an alpha channel, this function does nothing.

alpha The `alpha` functions allows to get the value of the alpha channel of a pixel.

For instance:

```
1 pixel src, dst;
2 dst.setAlpha( 0.5 * src.alpha() );
```

When the image doesn't have an alpha channel, this function return 1.0.

4.5.3 Region types

```
1 struct Region {
2   float x;
3   float y;
4   float width;
5   float height;
6 }
```

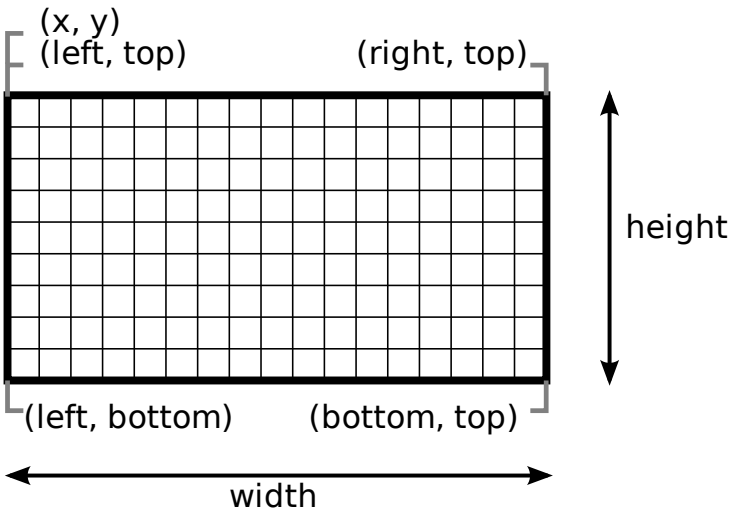

Figure 4.1: Region.

Members

left, right, top and bottom The function `left`, `right`, `top` and `bottom` return, respectively, the horrizontal coordinate of the left pixels and of the right pixels, and the vertical coordinate of the top pixels and bottom pixels.

```
region reg = { 1, 2, 10, 20 };
reg.left() == 1;
reg.left() == reg.x
reg.top() == 2;
reg.top() == reg.y
reg.right() == 20;
reg.right() == reg.x + reg.width - 1
reg.bottom() == 11;
reg.bottom() == reg.y + reg.height - 1
```

Intersect The `intersect` function will intersect the current region with an other region.

```
region reg1 = { 0, 0, 2, 2 };
region reg2 = { 1, -1, 3, 2 };
reg1.intersect( reg2 );
// Now reg1 == { 1, 0, 1, 1 };
```

Union The `union` function will unify the current region with an other region.

```
region reg1 = { 0, 0, 2, 2 };
region reg2 = { 1, -1, 3, 2 };
reg1.union( reg2 );
// Now reg1 == { 0, -1, 4, 3 };
```

Outset The `outset` function expand each edge of a given amount.

```
region reg = { 0, 1, 2, 3 };
reg.outset(1);
// Now reg == { -1, 0, 4, 5 };
```

Inset The `inset` function contract each edge of a given amount.

```
region reg = { 0, 1, 2, 3 };
reg.inset(1);
// Now reg == { 1, 2, 0, 1 };
```

4.6 Special types

4.6.1 Void

Functions that do not return a value are declared with the type `void`. For instance:

```
void addition(input int a, input int b, output int c)
{
  c = a + b;
}
```

Chapter 5

Function declaration

Chapter 6

Statements

6.1 Blocks

6.1.1 For loop

6.1.2 While loop

6.1.3 if else

6.2 Expressions

6.2.1 Operators

`.`	member selection
`++ --`	postfix increment and decrement
`++ --`	prefix increment and decrement
`- not`	unary negation and logical not
`* /`	multiplication and division
`+ -`	addition and substraction
`< > <= >=`	relational operators
`== !=`	equality and different
`and`	logical and
`xor`	logical exclusive or
`or`	logical inclusive or
`?:`	selection
`= += -= *= /=`	assignement

6.2.2 Pixel arithmetic

Pixel arithmetic is defined depending on the type of the channel, for floating point channels the arithmetic is the same as for numbers. But for integer channels, the arithmetic is defined by the following formulas:

MAX is the maximum value of an integer channel, $MAX = 255$ for 8bits, $MAX = 65535$ for 16bits and $MAX = 4294967295$ for 32bits.

- `+` : $(a + b)/MAX$

- `-` : $(a - b)/MAX$
- `*` : $(a * b)/MAX$
- `/` : $(a * MAX)/b$

To avoid overflow, operations are computed at a higher level of bit depth, 8bits operations are done in 16bits, 16bits operations in 32bits, and 32bits operation in 64bits.

6.2.3 Type conversion

Type conversion between primary types is implicit. They follow the following priority, from low to highest : bool, unsigned int, int, float, when a low priority is added to a high priority type, then the result is in the high priority type. Pixels types can be converted as a vector of float in the range $[0, 1]$ for integer pixels, or directly for float pixels. The opposite conversion is also possible from float vectors to pixels.

Chapter 7

Grammar

7.1 Keywords

- and
- or
- not
- const
- void
- int
- int2
- int3
- int4
- float
- float2
- float3
- float4
- bool
- bool2
- bool3
- bool4
- kernel
- if
- else

- while
- for
- return
- import
- struct
- true
- false
- image
- region
- pixel
- size
- output
- input

Reserved keywords:

- unsigned
- short
- half
- half2
- half3
- half4

Part II

Standard libray

7.2 Constants

7.3 Hints

Hints are special constants that are available to the Kernel writters (but not in libraries), they are called hints since they just provide an approximate information, since Kernels are expected to operate on an infinite plane, images sizes should just be used to provide a reasonnable scaling, and bounding to the user.

Constant name	Type	Description
`IMAGE_WIDTH`	`float`	hint of the width of the image
`IMAGE_HEIGHT`	`float`	hind of the height of the image
`IMAGE_SIZE`	`float2`	vector containing the width and height of the image

A typical use case of those hints is to control the center of an effect:

```
1  <
2    parameters: <
3      xcenter: <
4        label: "X center";
5        type: float;
6        minValue: 0;
7        maxValue: 1;
8        defaultValue: 0.5;
9      >;
10     ycenter: <
11       label: "Y center";
12       type: float;
13       minValue: 0;
14       maxValue: 1;
15       defaultValue: 0.5;
16     >;
17 >;
18 kernel MyKernel
19 {
20   const float2 center = { IMAGE_WIDTH * xcenter,
21                           IMAGE_HEIGHT * ycenter };
22   void evaluatePixel(output pixel result)
23   {
24     ...
25   }
26 }
```

7.4 Vector functions

distance

dot

cross