

ADOBE® PIXEL BENDER™

PIXEL BENDER REFERENCE



Copyright © 2008 Adobe Systems Incorporated. All rights reserved.

Adobe Pixel Bender Reference.

If this guide is distributed with software that includes an end user agreement, this guide, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. Except as permitted by any such license, no part of this guide may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior written permission of Adobe Systems Incorporated. Please note that the content in this guide is protected under copyright law even if it is not distributed with software that includes an end user license agreement.

The content of this guide is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this guide.

Please remember that existing artwork or images that you may want to include in your project may be protected under copyright law. The unauthorized incorporation of such material into your new work could be a violation of the rights of the copyright owner. Please be sure to obtain any permission required from the copyright owner.

Any references to company names in sample templates are for demonstration purposes only and are not intended to refer to any actual organization.

Adobe, the Adobe logo, After Effects, Flash, Photoshop, and Pixel Bender are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Apple, Mac, Mac OS, and Macintosh are trademarks of Apple Computer, Incorporated, registered in the United States and other countries. Sun and Java are trademarks or registered trademarks of Sun Microsystems, Incorporated in the United States and other countries. UNIX is a registered trademark of The Open Group in the US and other countries.

All other trademarks are the property of their respective owners.

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA. Notice to U.S. Government End Users. The Software and Documentation are "Commercial Items," as that term is defined at 48 C.F.R. §2.101, consisting of "Commercial Computer Software" and "Commercial Computer Software Documentation," as such terms are used in 48 C.F.R. §12.212 or 48 C.F.R. §227.7202, as applicable. Consistent with 48 C.F.R. §12.212 or 48 C.F.R. §§227.7202-1 through 227.7202-4, as applicable, the Commercial Computer Software and Commercial Computer Software Documentation are being licensed to U.S. Government end users (a) only as Commercial Items and (b) with only those rights as are granted to all other end users pursuant to the terms and conditions herein. Unpublished-rights reserved under the copyright laws of the United States. Adobe Systems Incorporated, 345 Park Avenue, San Jose, CA 95110-2704, USA. For U.S. Government End Users, Adobe agrees to comply with all applicable equal opportunity laws including, if appropriate, the provisions of Executive Order 11246, as amended, Section 402 of the Vietnam Era Veterans Readjustment Assistance Act of 1974 (38 USC 4212), and Section 503 of the Rehabilitation Act of 1973, as amended, and the regulations at 41 CFR Parts 60-1 through 60-60, 60-250, and 60-741. The affirmative action clause and regulations contained in the preceding sentence shall be incorporated by reference.

Contents

	Preface	5
1	Pixel Bender Language Overview	6
	Syntax and Program Structure	6
	Using filters defined with Pixel Bender	7
	Preprocessor directives	7
	Predefined preprocessor symbols	8
	Using Pixel Bender with Flash Player	8
2	Kernel Specifications	9
	Kernel syntax	9
	Kernel metadata	9
	Kernel members	10
	Kernel declarations	11
	Parameter metadata	12
	Kernel function definitions	13
	Statements in kernel functions	15
3	Pixel Bender Data Types	17
	Scalar types	17
	Conversions between scalar types	17
	Implementation notes	17
	Vector types	18
	Selecting and reordering elements	18
	Conversions between vector types	19
	Matrix types	20
	Other types	21
	Region type	21
	Image types	21
	Array types	21
	Void return type	22
	Operators	22
	Operations on multiple-value types	23
4	Pixel Bender Built-in Functions	24
	Mathematical functions	24
	Geometric functions	26
	Region functions	28
	Sampling functions	29

Intrinsic functions	29
5 Pixel Bender Graph Language	31
Graph elements	31
Graph syntax	31
Graph header	32
Graph element reference	33
graph	33
metadata	34
parameter	35
inputImage	35
outputImage	36
kernel	36
node	37
evaluateParameters	37
connect	38
Version identification in graphs	39

Preface

The Pixel Bender technology delivers a common image and video processing infrastructure which provides automatic runtime optimization on heterogeneous hardware.

Pixel Bender is a high-performance graphics programming tool intended for image processing. You can use the Pixel Bender kernel and graph languages to implement image processing algorithms in a hardware-independent manner.

This document, *Adobe Pixel Bender Reference*, is a reference manual and specification for the Pixel Bender kernel and graph languages.

The reference is intended for programmers who wish to use Pixel Bender to develop image filters for Adobe products. It assumes a basic knowledge of programming and image processing, as well as familiarity with the intended target application.

A companion document, *Pixel Bender Developer's Guide*, provides an introduction to the Pixel Bender Toolkit, including the Pixel Bender Toolkit IDE, an integrated development environment for Pixel Bender, as well as tutorials and examples of how to use Pixel Bender to develop filters.

1 Pixel Bender Language Overview

Pixel Bender is a high-performance graphics programming technology intended for image processing. This document is a complete reference and specification for the Pixel Bender *kernel* and *graph* languages.

Syntax and Program Structure

The Pixel Bender kernel language is designed for hardware-independent description of image-processing algorithms. It is designed to compile efficiently for both GPU and CPU back ends, including multi-core and multiprocessor systems. Since efficient execution on modern high-performance hardware requires parallel processing, the Pixel Bender programming model is explicitly parallel.

Pixel Bender kernel language is a C-like language with extensions for image processing. It is based on GLSL, which in turn is based on C. The basic syntax of the language should be familiar to any C programmer. If you have an OpenGL/GLSL background, you may recognize that a Pixel Bender program is analogous to a fragment shader (although there is no notion of geometry or vertex shading in Pixel Bender).

The basic unit of image processing in Pixel Bender is the *kernel*. Each Pixel Bender kernel language program defines one kernel. The kernel is an object that defines the result of one output pixel as a function of an arbitrary number of input pixels, which can be from a single image or from multiple input images.

The Pixel Bender run-time engine executes a kernel's defined pixel operation in parallel over all desired output pixels, to generate an output image. This parallel model means there are no interactions between the individual invocations of a kernel function for each output pixel; state cannot be shared between pixels. This is known as a *strict gather* model: a kernel gathers multiple input pixel values to produce a single pixel output. Notice that the output defined for each kernel is a single pixel, but the result of running the kernel is a complete image.

- A Pixel Bender program defines a named `kernel` object by specifying an `evaluatePixel()` function, which operates on input image data to produce a result pixel. Each kernel must contain this function definition. Additional helper functions can be defined. See [Chapter 2, "Kernel Specifications."](#)
- A `kernel` can take any number of parameters of arbitrary types. It can define parameters and variables to be used in its functions, and can import external function libraries. Pixel Bender is a strongly typed language. See [Chapter 3, "Pixel Bender Data Types."](#)
- The Pixel Bender kernel language provides many built-in functions for common pixel-manipulation operations. See [Chapter 4, "Pixel Bender Built-in Functions."](#)
- A complex image-processing algorithm may require multiple passes using different kernels. The Pixel Bender graph language allows you to define a *graph*, a set of kernels to be executed in a defined sequence to produce the desired result. See [Chapter 5, "Pixel Bender Graph Language."](#)

Using filters defined with Pixel Bender

Adobe provides the Pixel Bender Toolkit IDE, an integrated development environment for developing programs in the Pixel Bender kernel language. The Toolkit is documented in the *Pixel Bender Developer's Guide*.

During development, you can run your programs in the Pixel Bender Toolkit IDE, which supplies a parameter interface for you. For information on the Pixel Bender Toolkit IDE, see *Pixel Bender Developer's Guide*.

The Pixel Bender run-time engine is integrated into *client applications*, which include Adobe Photoshop®, After Effects®, and Flash® Player.

- ▶ A Pixel Bender program is saved to a file with the extension `.pbk`. These PBK files can be loaded and used as filters by Adobe image-manipulation programs After Effects and Photoshop.
- ▶ You can compile a PBK file (with certain limitations) to the PBJ format. A PBJ file can be loaded and used as a filter by Flash Player.

The client application uses information from the filter definition to decide how to present a UI in which the user can select a filter or effect, and set kernel parameters. Different clients may use the supplied information differently.

- ▶ The client application typically uses the `name` attribute supplied in the `kernel` or `graph` element to create a menu or palette item that identifies and invokes the filter.
- ▶ You can supply a description string as metadata for a kernel, which a client might use to supply a tooltip.
- ▶ When you choose a Pixel Bender filter from the client application's filter or effect menu or palette, the application provides an interface (such as a dialog) in which the user can enter required parameters. When you define parameters, you supply metadata (such as minimum, maximum, and default values) that help the client choose and configure appropriate UI controls.

For more details of how Pixel Bender filters are integrated into each application, and what extensions or limitations apply for each client, see the *Pixel Bender Developer's Guide*.

Preprocessor directives

A C-style preprocessor is available with the following keywords:

```
#if
#ifdef
#define
#endif
#elif
#define
```

Predefined preprocessor symbols

Several predefined preprocessor symbols are available at compile time.

If the compilation is done on a machine with an ATI graphics card, every Pixel Bender program acts as though it is prefixed by the following lines:

```
#define AIF_ATI 1
#define AIF_NVIDIA 0
```

If the compilation is done on a machine with an nVidia graphics card, the prefix looks like this:

```
#define AIF_ATI 0
#define AIF_NVIDIA 1
```

Both symbols always are defined. To mark card-specific code, use `#if`, not `#ifdef`.

- If the compilation will result in the production of byte code for Flash Player, every Pixel Bender program acts as though it is prefixed by the following line:

```
#define AIF_FLASH_TARGET 1
```

If the compilation is for a non-Flash Player target, every Pixel Bender program acts as though it is prefixed by the following line:

```
#define AIF_FLASH_TARGET 0
```

Using Pixel Bender with Flash Player

Because Flash Player software must run on a wide variety of hardware, only a subset of Pixel Bender is available for use in Flash Player. Limitations are indicated in each applicable feature description. This is a summary of supported functionality when Pixel Bender is used in Flash Player:

- The preprocessor symbol `AIF_FLASH_TARGET` is defined to be 1.
- Flash Player always uses 1x1 square pixels. The function `pixelSize()` always returns (1.0, 1.0), and `pixelAspectRatio()` always returns 1.0.
- The selection operator (`? :`) can be used only to select between two constants or variables.
- Pixel Bender images have 32 bits per channel, but graphics in Flash Player 10 have only 8 bits per channel. When a kernel is run in Flash Player, the input image data is converted to 32 bits per channel and then converted back to 8 bits per channel when kernel execution is complete.
- The only available flow-control statements are `if` and `else`.
- The following are not supported:
 - Region functions .
 - Custom support functions and libraries.
 - Dependent values.
 - Arrays.
 - The Pixel Bender graph language.

2 Kernel Specifications

Kernel syntax

Each Pixel Bender program is specified by one string, which contains a set of metadata enclosed in angle brackets that describes the kernel, and a set of members enclosed in curly braces that define the filtering operation.

```
<languageVersion : 1.0;>
kernel name
<
  kernel metadata pairs
>
{
  kernel members
}
```

Every kernel must begin with the `languageVersion` statement, which identifies the version of the Pixel Bender kernel language in which this kernel is written, followed by the kernel definition.

Kernel metadata

The first portion of the kernel definition is the *kernel metadata*, a series of name-value pairs enclosed in angle brackets:

```
<
  name1 : value1;
  name2 : value2;
  ...
>;
```

These metadata values are predefined:

<code>namespace</code>	Required. A string, the namespace within which this kernel is defined. The <code>namespace</code> value is used in combination with the other filter identifiers to determine the actual namespace, so it need not be globally unique. You can use it, for example, to distinguish categories of kernels.
<code>vendor</code>	Required. A string, the vendor supplying this kernel.
<code>version</code>	Required. An integer value, the version number of this implementation of this kernel. This is distinct from the kernel language version specified in the <code>languageVersion</code> element.
<code>description</code>	Optional. A string describing the purpose of this kernel. Applications that integrate with Pixel Bender have access to this value, and can use it to create menu items, tooltips, or other UI elements.

For example:

```
<
  namespace : "Pixel Bender IDE";
  vendor : "Adobe";
  version : 1;
  description: "A sample filter";
>
```

AFTER EFFECTS NOTE: After Effects defines two additional kernel metadata properties, both of which are optional:

displayname	An effect name to show in the Effects and Presets panel. If not specified, the kernel name is used.
category	The category of the effect. Default is the 'Pixel Bender' category.

Kernel members

The second part of the kernel definition is a set of *kernel members* enclosed in curly braces. Members are declarations and function definitions. The kernel must contain at least an `evaluatePixel()` function definition; all other members are optional:

```
{
  [declarations]
  [support functions]
  void evaluatePixel()
  {
    statements
  }
}
```

- Declarations usually include a declaration of input images and output pixels. They can include parameters, dependent variables, and constants to be used in the functions, and can be used to import function libraries. See ["Kernel declarations" on page 11](#).
- The main function, `evaluatePixel()`, is applied to the input image or images, performing the transformations that result in the output pixel. It can use helper functions of particular predefined types, and you can also define arbitrary helper functions. The functions have specified types of access to the supplied parameters, variables, and constants. See ["Kernel function definitions" on page 13](#).

Kernel declarations

Before the `evaluatePixel()` function, a kernel definition can contain these declarational members:

Declaration	Syntax
parameter (kernel contains zero or more)	<pre>parameter type name < name1 : value1; name2 : value2; ... ></pre>
Parameters are set before a kernel is executed and are read-only within kernel functions. Parameters can be of any type except <code>image</code> and <code>region</code> . Float arrays used as parameters must have sizes determined at compile time.	
A parameter can have optional metadata attached to it, as one or more name-value pairs enclosed in angle brackets. See "Parameter metadata" on page 12 .	
dependent (kernel contains zero or more)	<pre>dependent type name</pre>
Dependent variables are accessible within any kernel function, but can be written to only in the <code>evaluateDependents()</code> function. Float arrays used as dependents must have sizes determined at compile time.	
FLASH PLAYER NOTE: Flash Player does not support dependent variables.	
const (kernel contains zero or more)	<pre>const type name=compile-time_expression;</pre>
The value of the constant is determined at compile time.	
The C-preprocessor directives <code>#define</code> , <code>#undef</code> , <code>#ifdef</code> , and <code>#if</code> are provided to support conditional compilation. The use of <code>const</code> is recommended for constant definitions.	
input (kernel contains zero or more)	<pre>input type name;</pre>
An image to use as input to the <code>evaluatePixel()</code> function.	
The type must be <code>image1</code> , <code>image2</code> , <code>image3</code> , or <code>image4</code> .	
output (kernel contains one or more)	<pre>output type name;</pre>
The output pixels that contain the results of the <code>evaluatePixel()</code> function.	
The type must be <code>pixel1</code> , <code>pixel2</code> , <code>pixel3</code> , or <code>pixel4</code> .	

Parameter metadata

A parameter specification can include metadata that describes the parameter and places constraints on its value. This metadata is made available to the client application after the compilation, and helps the client determine how to present the UI that allows users to set the parameter value.

Metadata values are enclosed in angle brackets following the parameter specification:

```
parameter type name
<
  name1 : value1;
  name2 : value2;
  ...
>
```

The names are strings. The values are constants of any valid Pixel Bender type. For `int`, `float`, and `bool`, the type is deduced automatically. For other types, specify a constant of the correct type (such as `float2 (1.0, -1.0)`), or a string delimited by double quotes. For example:

```
parameter int angle
<
  minValue : 0;
  maxValue : 360;
  defaultValue : 30;
  description : "measured in degrees";
>;
```

These parameter metadata values specify constraints on the parameter value:

<code>minValue</code>	The minimum allowed value.
<code>maxValue</code>	The maximum allowed value.
<code>defaultValue</code>	The default value

These parameter metadata values describe how the parameter should be presented in the UI, as determined by the client application:

<code>description</code>	A descriptive string that a client application can use in any way. Not used in After Effects.
<code>aeDisplayName</code>	The display name for After Effects. Ignored by other client applications.

The default display name for a kernel parameter is the parameter name. Because Pixel Bender parameter names cannot contain spaces or other reserved characters, you can use this metadata value to specify a user-friendly string that After Effects will use to refer to this parameter anywhere it appears in the UI.

aeUIControl	<p>For After Effects only, what kind of UI control to use to allow user input for this parameter. The value depends on the parameter's data type:</p> <hr/> <p>Parameter data type Allowed control values</p> <hr/> <p>int aeUIControl: "aePopup"</p> <p style="padding-left: 40px;">Requires an additional metadata value to specify the popup menu values, as a string with individual items separated by the pipe character:</p> <p style="padding-left: 40px;">aePopupString: "Item 1 Item 2 Item 3"</p> <hr/> <p>float aeUIControl: "aeAngle" aeUIControl: "aePercentSlider" aeUIControl: "aeOneChannelColor"</p> <hr/> <p>float2 aeUIControl: "aePoint"</p> <p style="padding-left: 40px;">Tells After Effects that this parameter represents a point on the image, such as the center of a circle that is being drawn on the image. If you wish to specify a default point, use the following additional metadata value:</p> <p style="padding-left: 40px;">aePointRelativeDefaultValue: float2(x, y)</p> <p style="padding-left: 40px;">These are not pixel coordinates, but are relative to the image's size. For instance, aePointRelativeDefaultValue: float2(0.5, 0.5) sets the default position to the middle of the image, however large it is. Relative coordinates (1.0, 1.0) set the default position to the bottom right of the image, (0.0, 0.0) to the upper left.</p> <hr/> <p>float3 aeUIControl: "aeColor"</p> <p>float4 In float4 color values, the alpha channel is always 1, because in After Effects, color controls only support opaque colors.</p>
-------------	--

Kernel function definitions

The kernel definition can contain these function definitions.

- All except `evaluatePixel()` are optional.
- The region functions (`needed()`, `changed()`, `generated()`) can read parameters and dependents, but cannot sample kernel input images. They can, however, call the built-in `dod()` function on kernel input images.

```

evaluatePixel()      void evaluatePixel()
                        {
                            statements
                        }

```

Defines the processing to be performed, in parallel, at each pixel of the output image or images. This function and all functions that it calls have:

- read-only access to all parameters and dependent variables;
- read-only access to all input images;
- write access to all output pixels.

```

evaluateDependents() void evaluateDependents()
                        {
                            statements
                        }

```

Writes values to variables declared in *dependent* statements. These values can be written only during the execution of this function or within functions that it calls.

```

needed()             region needed(region outputRegion, //requested output region
                                imageRef inputIndex) // reference to an image
                        {
                            statements
                        }

```

outputRegion—The requested output region. This is the size and position in world coordinates of the image to be calculated.

inputIndex—The input image. If there are multiple input images, this distinguishes the one for which this function determines a needed region.

Finds the region of each input image that is needed to correctly calculate all of the pixels in the requested output region. The result is the region of a given input image in which pixels must be considered. Pixels outside this region of the input image are not processed by the `evaluatePixel()` function.

Called once for each input image, before any calls to `evaluatePixel()`.

```

changed()           region changed(region inputRegion, //input region that changed
                                imageRef inputIndex) //reference to an image
                        {
                            statements
                        }

```

inputRegion—The region of an input image within which pixels have changed.

inputIndex—If there are multiple input images, this distinguishes the one for which the input region has changed.

Finds the region within the output image in which pixels must be recomputed when any pixels change in a given input region. This function is used to compute the bounds (domain of definition) of the output image.

Called once for each input image, before any calls to `evaluatePixel()`.

```

generated()           region generated()
                        {
                            statements
                        }

```

Creates and returns a `region`. Finds the region of the output image where non-zero pixels will be produced even if all image inputs are completely empty.

```

other functions      returnType name([arguments])
                        {
                            statements
                        }

```

You can define zero or more additional kernel functions. These take access restrictions from their calling parent; for example, only functions called from `evaluateDependents()` can write to dependent variables.

The argument syntax is:

```
[in|out|inout] type name
```

The default qualifier is `in`. The argument is passed by value into the function. If a variable is used, any changes that the function makes to the value are not reflected in the variable when the function returns.

The `out` qualifier indicates that the argument is a return value, a variable that is passed by reference, uninitialized upon entry to the function.

The `inout` qualifier indicates that the argument is a variable, initialized to the caller's value on entry and passed by reference. Any changes that the function makes to the value are available in the variable upon return.

Functions can be named according to the usual C conventions. All functions names that start with an underscore (`_`) are reserved and cannot be used.

All functions are overloaded; that is, matched by argument types as well as names. Unlike C++, no implicit type conversion is performed when matching overloaded functions. All functions must be defined before calling; there are no forward declarations. Pixel Bender does not support recursive function calls.

FLASH PLAYER NOTE: Flash Player does not support custom function definitions.

Statements in kernel functions

The following flow-control constructs are supported in Pixel Bender, with the usual C syntax:

```

if (scalar_expression) true_statement
if (scalar_expression) true_statement else false_statement
for (initializer; condition; incremental) statement
while (condition) statement
do statement while (condition);
break;
continue;
return expression;

```

FLASH PLAYER NOTE: When Pixel Bender is used in Flash Player, the only flow-control statements available are `if` and `else`.

Within the `evaluatePixel()` function and functions called from `evaluatePixel()`, Pixel Bender does not support `return` statements inside the body of a conditional statement or loop.

A statement can be an expression or a variable declaration. A variable declaration can be initialized or not:

```
expression
type name;
[const] type name=expression;
```

Variables can be declared anywhere inside a function and have scope inside the enclosing set of braces.

- As in C++, variables also can be declared inside the initializer of a `for` loop or the conditional test of a `while` loop, but not within the conditional test of an `if` statement.
- Variables can hide other variables of the same name in outer scopes.
- The `const` qualifier can be applied only if an expression is a compile-time constant.
- Variables can be named according to the usual C conventions. All variable names starting with an underscore (`_`) are reserved and cannot be used.

As in C, a statement also can be a sequence of statements of the types above, inside braces:

```
{
    statement
    [statement...]
}
```

3 Pixel Bender Data Types

Pixel Bender is strongly typed. There are no automatic conversions between types, with the single exception of integral promotion during construction of floating-point vector and matrix types. There are several classes of types, each defined over a particular set of operators and intrinsic functions.

Scalar types

Pixel Bender supports these basic numeric types:

<code>bool</code>	A Boolean value
<code>int</code>	An integer value
<code>float</code>	A floating-point value
<code>pixel11</code>	Represents the value of one channel of an image. The name distinguishes this single-element pixel from a pixel that contains multiple channels. Pixel values are assumed to be 32-bit floating point numbers.

All of these numeric types can participate in arithmetic operations. All the usual arithmetic operators are defined over the scalar types; see ["Operators" on page 22](#).

Conversions between scalar types

The types `bool`, `int`, and `float` can be converted from one to another, using the usual C-style truncation and promotion rules, with the following cast syntax:

```
type(expression)
```

For example:

```
int a=int(myfloat)
```

The `pixel11` type can be used interchangeably with `float`.

Implementation notes

The `int` type has at least 16 bits of precision (not including the sign), but an implementation can use more than 16 bits. An implementation can convert an `int` to a `float` to operate on it. When the result of an `int` operation (including a conversion from `float`) cannot be represented as an `int`, the behavior is undefined.

The `float` type matches the IEEE single-precision floating-point definition for precision and dynamic range. The precision of internal processing is not required to match the IEEE floating-point specification for floating-point operations, but does meet the guidelines for precision established by the OpenGL 1.4 specification.

Vector types

Pixel Bender supplies 2-, 3-, and 4-element vectors for each of the scalar types:

```
float2    bool2  int2   pixel2
float3    bool3  int3   pixel3
float4    bool4  int4   pixel4
```

AFTER EFFECTS NOTE: After Effects allows only 4-channel input and output images.

Initialize any of the vector types, including pixels, using this constructor syntax:

```
vectorType(element1 [, element2...])
```

For example:

```
float3(0.5, 0.6, 0.7)
```

This expression results in a value of the named type, which can be assigned to a variable or used directly as an unnamed result. A shorthand syntax sets all elements to the same value; these two statements are equivalent:

```
float3(0.03);
float3(0.03, 0.3, 0.3);
```

Most scalar arithmetic operators are defined over vectors as operating component-wise; see [“Operators” on page 22](#).

You can access a vector element by index or names.

- Use the subscript operator with a zero-based integer index:

```
vectorValue[index]
```

- Use dot notation to retrieve named elements in these sequences:

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

Each of these names corresponds to an index from zero to three.

For example, to retrieve the first value of a vector `myVectorValue`, you can use any of these notations:

```
myVectorValue[0]
myVectorValue.r
myVectorValue.x
myVectorValue.s
```

Selecting and reordering elements

Pixel Bender allows “swizzling” to select and re-order vector elements. For a vector value with n elements, up to n named indices can be specified following the dot operator. The corresponding elements of the vector value are concatenated to form a new vector result with as many elements as index specifiers. This syntax can be used to re-order, remove, or repeat elements; for example:

```
float4 vec4;
```

```
float3 no_alpha=vec4.rgb;    // drop last component
float3 no_r=vec4.gba;      // drop first component
float4 reversed=vec4.abgr;  // reverse order
float4 all_red=vec4.rrrr;   // repeated elements
float4 all_x=vec4.xxxx;     // same as all_red
```

Indices from separate sequences cannot be combined:

```
float4 vec4;
float3 no_alpha=vec4.rgz;    // Error
```

Index specifiers also can be applied to variables on the left side of an assignment. In this case, indices cannot be repeated. This functionality is used to implement write-masking. The correct number of elements must be supplied on the right-hand side.

```
float3 vec3;
float2 vec2;
vec3.xy=vec2;                // assign vec2's elements to vec3[0] and vec3[1]
vec3.xz=vec2;                // assign vec2's elements to vec3[0] and vec3[2]
```

Interactions

Swizzling and write-masking can be used simultaneously on both sides of an expression:

```
vec3.xz=vec4.wy;
```

There is a potentially troublesome interaction between swizzling and the assignment operations. Consider the following expression:

```
g.yz *= g.yy;
```

A naive expansion of this would look like this:

```
g.y *= g.y;
g.z *= g.y;
```

The problem with this is that the value of `g.y` used in the second expression has been modified. The correct expansion of the original statement is:

```
float2 temp=g.yz * g.yy;
g.yz=temp;
```

That is, the original value of `g.y` is used for both multiplications; `g.y` is not updated until after both multiplications are done.

Conversions between vector types

Conversions between vector types are possible, provided the dimensions of the vectors are equal. Convert (as for scalar types) using C-style truncation and promotion rules, with the following cast syntax:

```
type(expression)
```

For example:

```
float3 fvec3;
int3 ivec3;
fvec3=float3(ivec3);
```

Matrix types

These matrix types are available:

```
float2x2
float3x3
float4x4
```

Generate matrix value with constructor syntax, using `float` vectors describing the column values, or `float` values indicating each element in column-major order, or a mixture of vectors and floats:

```
float2x2( float2, float2 )
float2x2( float, float,
          float, float )

float3x3( float3, float3, float3 )
float3x3( float, float, float,
          float, float, float,
          float, float, float )

float4x4( float4, float4, float4, float4 )
float4x4( float, float, float, float,
          float, float, float, float,
          float, float, float, float,
          float, float, float, float )
```

You can also initialize a matrix from a single float, which defines the elements on the leading diagonal. All other elements are set to zero.

```
float2x2( float )
float3x3( float )
float4x4( float )
```

To access matrix elements, use double subscripts, column first:

```
matrix[ column ][ row ]
```

If the row subscript is omitted, a single column is selected, and the resulting type is a `float` vector of the appropriate dimension:

```
matrix[ column ]
```

A small set of scalar operators are defined for matrices, which perform component-wise, matrix/matrix, and matrix/vector operations. See [“Operators” on page 22](#).

Other types

Region type

The region type is declared as follows:

```
region
```

A rectangular region can be constructed from a `float4` representing the left, top, right, and bottom bounds:

```
region( float4_bounds )
```

There are no operators defined for regions; instead, regions are manipulated through a set of specialized functions. See [Chapter 4, “Pixel Bender Built-in Functions.”](#)

Image types

Pixel Bender supports images of up to four channels.

```
image1
image2
image3
image4
```

AFTER EFFECTS NOTE: After Effects allows only 4-channel input and output images.

Images cannot be constructed or used in expressions; however, they can be passed as arguments to user-defined functions or passed as an argument to the `dod()` built-in function.

The `imageRef` type allows the `needed()` and `changed()` functions to select the input image for which they are being run. There are only two uses for an `imageRef` variable:

- It can be compared for equality or inequality to an input image.
- It can be passed to the `dod()` built-in function.

Array types

Pixel Bender has some support for arrays. The following one-dimensional arrays are allowed:

- Constant-size arrays of `floats` declared as kernel parameters.
- Constant-size arrays of `floats` declared as kernel dependents.

NOTE: Pixel Bender 1.0 supports only arrays of floats, and the array size is a compile-time constant. Future versions of Pixel Bender will allow arrays to be declared with a size based on kernel parameters, which will enable parameter-dependent look-up table sizes.

Declare and access arrays using C syntax:

```
type name[ size ];
name[ subscript ]
```

Attempting to access an array with a subscript less than 0 or greater than the declared size minus 1 causes a run-time error.

FLASH PLAYER NOTE: When Pixel Bender is used in Flash Player, arrays are not available .

Void return type

Functions that do not return a value must be declared with the `void` return type. There is no other legal use for `void` within the Pixel Bender kernel language.

Operators

Pixel Bender defines the following arithmetic operators over the scalar types, with their usual C meanings, in order of highest to lowest precedence. Parentheses can be used to override precedence.

<code>.</code>	Member selection
<code>++ --</code>	Postfix increment or decrement
<code>++ --</code>	Prefix increment or decrement
<code>- !</code>	Unary negation, logical not
<code>* /</code>	Multiply, divide
<code>+ -</code>	Add, subtract
<code>< > <= >=</code>	Relational
<code>== !=</code>	Equality
<code>&&</code>	Logical and
<code>^^</code>	Logical exclusive or
<code> </code>	Logical inclusive or
<code>= += -= *= /=</code>	Assignment
<code>?:</code>	Selection FLASH PLAYER NOTE: When Pixel Bender is used in Flash Player, you can only use the selection operator to select between two constants or variables. You cannot place a general expression on the right-hand side of the selection.

Short-circuit evaluation for logical AND, and logical inclusive OR is not provided. If you require short-circuit evaluation to be present (or absent), you must explicitly code it.

The following are undefined:

```
n/0
sqrt(-n)
tan(pi/2)
```

Operations on multiple-value types

The standard arithmetic operators (+, -, *, /) can be used with combinations of vectors, matrices, and scalars.

A binary operator can be applied to two vector quantities only if they have the same size. The operation behaves as though it were applied to each component of the vector. For example:

```
float3 x, y, z;
z=x + y;
```

This operation is equivalent to:

```
z[ 0 ]=x[ 0 ] + y[ 0 ];
z[ 1 ]=x[ 1 ] + y[ 1 ];
z[ 2 ]=x[ 2 ] + y[ 2 ];
```

Combining a scalar with a vector also is possible. For example:

```
float3 x, y;
float w;
x=y * w;
```

This operation is equivalent to:

```
x[ 0 ]=y[ 0 ] * w;
x[ 1 ]=y[ 1 ] * w;
x[ 2 ]=y[ 2 ] * w;
```

Important exceptions to this component-wise operation are multiplications between matrices and multiplications between matrices and vectors. These perform standard linear algebraic multiplications, not component-wise multiplications:

```
float2x2 * float2x2
float3x3 * float3x3
float4x4 * float4x4
```

Linear-algebraic matrix multiplication

```
float2x2 * float2
float3x3 * float3
float4x4 * float4
```

Column-vector multiplication

```
float2 * float2x2
float3 * float3x3
float4 * float4x4
```

Row-vector multiplication

4 Pixel Bender Built-in Functions

Pixel Bender supports a variety of built-in functions over different data types.

Mathematical functions

As with arithmetic operators, mathematical functions can be applied to vectors, in which case they act in a component-wise fashion. Unless stated otherwise, all angles are measured in radians.

<pre>float radians(float degrees) float2 radians(float2 degrees) float3 radians(float3 degrees) float4 radians(float4 degrees)</pre>	Converts degrees to radians.
<pre>float degrees(float radians) float2 degrees (float2 radians) float3 degrees (float3 radians) float4 degrees (float4 radians)</pre>	Converts radians to degrees.
<pre>float sin(float radians) float2 sin(float2 radians) float3 sin(float3 radians) float4 sin(float4 radians)</pre>	Returns the sine of the input.
<pre>float cos(float radians) float2 cos(float2 radians) float3 cos(float3 radians) float4 cos(float4 radians)</pre>	Returns the cosine of the input.
<pre>float tan(float radians) float2 tan(float2 radians) float3 tan(float3 radians) float4 tan(float4 radians)</pre>	Returns the tangent of the input. Undefined if $x = \pi/2.0$.
<pre>float asin(float x) float2 asin(float2 x) float3 asin(float3 x) float4 asin(float4 x)</pre>	Returns the arc sine of the input. The result is in the range $[-\pi/2.. \pi/2]$.
<pre>float acos(float x) float2 acos(float2 x) float3 acos(float3 x) float4 acos(float4 x)</pre>	Returns the arc cosine of the input. The result is in the range $[0.. \pi]$.
<pre>float atan(float y_over_x) float2 atan(float2 y_over_x) float3 atan(float3 y_over_x) float4 atan(float4 y_over_x)</pre>	Returns the arc tangent of the input. The result is in the range $[-\pi/2.. \pi/2]$.
<pre>float atan(float y, float x) float2 atan(float2 y, float2 x) float3 atan(float3 y, float3 x) float4 atan(float4 y, float4 x)</pre>	Returns the arc tangent of y/x . The result will be in the range $[-\pi.. \pi]$.

<pre>float pow(float x, float y) float2 pow (float2 x, float2 y) float3 pow (float3 x, float3 y) float4 pow (float4 x, float4 y)</pre>	Returns x^y .
<pre>float exp(float x) float2 exp(float2 x) float3 exp(float3 x) float4 exp(float4 x)</pre>	Returns e^x .
<pre>float exp2(float x) float2 exp2(float2 x) float3 exp2(float3 x) float4 exp2(float4 x)</pre>	Returns 2^x .
<pre>float log(float x) float2 log(float2 x) float3 log(float3 x) float4 log(float4 x)</pre>	Returns the natural logarithm of x .
<pre>float log2(float x) float2 log2(float2 x) float3 log2(float3 x) float4 log2(float4 x)</pre>	Returns the base-2 logarithm of x .
<pre>float sqrt(float x) float2 sqrt(float2 x) float3 sqrt(float3 x) float4 sqrt(float4 x)</pre>	Returns the positive square root of x . Undefined if $x < 0$.
<pre>float inverseSqrt(float x) float2 inverseSqrt(float2 x) float3 inverseSqrt(float3 x) float4 inverseSqrt(float4 x)</pre>	Returns the reciprocal of the positive square root of x . Undefined if $x < 0$.
<pre>float abs(float x) float2 abs(float2 x) float3 abs(float3 x) float4 abs(float4 x)</pre>	If $x \geq 0$, returns x , otherwise returns $-x$.
<pre>float sign(float x) float2 sign(float2 x) float3 sign(float3 x) float4 sign(float4 x)</pre>	If $x < 0$, returns -1 If $x == 0$, returns 0 If $x > 0$, returns 1
<pre>float floor(float x) float2 floor(float2 x) float3 floor(float3 x) float4 floor(float4 x)</pre>	Returns y , the nearest integer where $y \leq x$.
<pre>float ceil(float x) float2 ceil(float2 x) float3 ceil(float3 x) float4 ceil(float4 x)</pre>	Returns y , the nearest integer where $y \geq x$.
<pre>float fract(float x) float2 fract(float2 x) float3 fract(float3 x) float4 fract(float4 x)</pre>	Returns $x - \text{floor}(x)$.

<pre>float mod(float x, float y) float2 mod(float2 x, float2 y) float3 mod(float3 x, float3 y) float4 mod(float4 x, float4 y)</pre>	Returns $x - y * \text{floor}(x/y)$.
<pre>float min(float x, float y) float2 min(float2 x, float2 y) float3 min(float3 x, float3 y) float4 min(float4 x, float4 y)</pre>	If $x < y$, returns x , otherwise returns y .
<pre>float max(float x, float y) float2 max(float2 x, float2 y) float3 max(float3 x, float3 y) float4 max(float4 x, float4 y)</pre>	If $x > y$, returns x , otherwise returns y .
<pre>float step(float x, float y) float2 step(float2 x, float2 y) float3 step(float3 x, float3 y) float4 step(float4 x, float4 y)</pre>	If $y \leq x$, returns 0.0, otherwise returns 1.0
<pre>float clamp(float x, float minval, float maxval) float2 clamp(float2 x, float2 minval, float2 maxval) float3 clamp(float3 x, float3 minval, float3 maxval) float4 clamp(float4 x, float4 minval, float4 maxval)</pre>	If $x < \text{minval}$, returns minval If $x > \text{maxval}$, returns maxval otherwise returns x .
<pre>float mix(float x, float y, float a) float2 mix(float2 x, float2 y, float2 a) float3 mix(float3 x, float3 y, float3 a) float4 mix(float4 x, float4 y, float4 a)</pre>	Returns $x * (1.0 - a) + y * a$ (that is, a linear interpolation between x and y).
<pre>float smoothStep(float edge0, float edge1, float x) float2 smoothStep(float2 edge0, float2 edge1, float2 x) float3 smoothStep(float3 edge0, float3 edge1, float3 x) float4 smoothStep(float4 edge0, float4 edge1, float4 x)</pre>	If $x \leq \text{edge0}$, returns 0 If $x \geq \text{edge1}$, returns 1, otherwise performs smooth hermite interpolation.

Geometric functions

These functions operate on vectors as vectors, rather than treating each component of the vector individually.

<pre>float length(float x) float length(float2 x) float length(float3 x) float length(float4 x)</pre>	Returns the length of the vector x .
<pre>float distance(float x, float y) float distance(float2 x, float2 y) float distance(float3 x, float3 y) float distance(float4 x, float4 y)</pre>	Returns the distance between x and y .
<pre>float dot(float x, float y) float dot(float2 x, float2 y) float dot(float3 x, float3 y) float dot(float4 x, float4 y)</pre>	Returns the dot product of x and y .

<code>float3 cross(vector3 x, vector3 y)</code>	Returns the cross product of <code>x</code> and <code>y</code> .
<code>float normalize(float x)</code> <code>float2 normalize(float2 x)</code> <code>float3 normalize(float3 x)</code> <code>float4 normalize(float4 x)</code>	Returns a vector in the same direction as <code>x</code> but with a length of 1. Undefined if <code>length(x) == 0</code> .

These functions perform component-wise multiplication (as opposed to the `*` operator, which performs algebraic matrix multiplication):

<code>float2x2 matrixCompMult(float2x2 x, float2x2 y)</code> <code>float3x3 matrixCompMult(float3x3 x, float3x3 y)</code> <code>float4x4 matrixCompMult(float4x4 x, float4x4 y)</code>	Returns the component-wise product of <code>x</code> and <code>y</code> .
--	---

These functions compare vectors component-wise and return a component-wise Boolean vector result of the same size.

<code>bool2 lessThan(int2 x, int2 y)</code> <code>bool3 lessThan(int3 x, int3 y)</code> <code>bool4 lessThan(int4 x, int4 y)</code> <code>bool2 lessThan(float2 x, float2 y)</code> <code>bool3 lessThan(float3 x, float3 y)</code> <code>bool4 lessThan(float4 x, float4 y)</code>	Returns the component-wise compare of <code>x < y</code> .
<code>bool2 lessThanEqual(int2 x, int2 y)</code> <code>bool3 lessThanEqual(int3 x, int3 y)</code> <code>bool4 lessThanEqual(int4 x, int4 y)</code> <code>bool2 lessThanEqual(float2 x, float2 y)</code> <code>bool3 lessThanEqual(float3 x, float3 y)</code> <code>bool4 lessThanEqual(float4 x, float4 y)</code>	Returns the component-wise compare of <code>x <= y</code> .
<code>bool2 greaterThan(int2 x, int2 y)</code> <code>bool3 greaterThan(int3 x, int3 y)</code> <code>bool4 greaterThan(int4 x, int4 y)</code> <code>bool2 greaterThan(float2 x, float2 y)</code> <code>bool3 greaterThan(float3 x, float3 y)</code> <code>bool4 greaterThan(float4 x, float4 y)</code>	Returns the component-wise compare of <code>x > y</code> .
<code>bool2 greaterThanEqual(int2 x, int2 y)</code> <code>bool3 greaterThanEqual(int3 x, int3 y)</code> <code>bool4 greaterThanEqual(int4 x, int4 y)</code> <code>bool2 greaterThanEqual(float2 x, float2 y)</code> <code>bool3 greaterThanEqual(float3 x, float3 y)</code> <code>bool4 greaterThanEqual(float4 x, float4 y)</code>	Returns the component-wise compare of <code>x >= y</code> .
<code>bool2 equal(int2 x, int2 y)</code> <code>bool3 equal(int3 x, int3 y)</code> <code>bool4 equal(int4 x, int4 y)</code> <code>bool2 equal(float2 x, float2 y)</code> <code>bool3 equal(float3 x, float3 y)</code> <code>bool4 equal(float4 x, float4 y)</code> <code>bool2 equal(bool2 x, bool2 y)</code> <code>bool3 equal(bool3 x, bool3 y)</code> <code>bool4 equal(bool4 x, bool4 y)</code>	Returns the component-wise compare of <code>x == y</code> .

<pre>bool2 notEqual(int2 x, int2 y) bool3 notEqual(int3 x, int3 y) bool4 notEqual(int4 x, int4 y) bool2 notEqual(float2 x, float2 y) bool3 notEqual(float3 x, float3 y) bool4 notEqual(float4 x, float4 y) bool2 notEqual(bool2 x, bool2 y) bool3 notEqual(bool3 x, bool3 y) bool4 notEqual(bool4 x, bool4 y)</pre>	Returns the component-wise compare of $x \neq y$.
---	--

These vector functions operate only on vectors of Boolean type:

<pre>bool any(bool2 x) bool any(bool3 x) bool any(bool4 x)</pre>	True if any element of x is true.
<pre>bool all(bool2 x) bool all(bool3 x) bool all(bool4 x)</pre>	True if all elements of x are true.
<pre>bool2 not(bool2 x) bool3 not(bool3 x) bool4 not(bool4 x)</pre>	Element-wise logical negation.

Region functions

These functions manipulate the opaque `region` type:

<code>region nowhere()</code>	Returns the empty region.
<code>region everywhere()</code>	Returns an infinite region. ► After Effects does not support this function. If you have a kernel that uses this built-in function, to use it in After Effects you must modify it to produce output in a bounded region.
<code>region transform(float2x2 m, region r)</code>	Performs a linear transformation on region r .
<code>region transform(float3x3 m, region r)</code>	Performs an affine transformation on region r .
<code>region union(region a, region b)</code>	Returns the union of a and b .
<code>region intersect(region a, region b)</code>	Returns the intersection of a and b .
<code>region outset(region a, float2 amount)</code>	Expands region a by the given amount at each edge.
<code>region inset(region a, float2 amount)</code>	Contracts region a by the given amount at each edge.
<code>float4 bounds(region r)</code>	Returns $(leftX, topY, rightX, bottomY)$.

<code>bool isEmpty(region r)</code>	Returns true if region <code>r</code> is empty.
<code>region dod(image1)</code> <code>region dod(image2)</code> <code>region dod(image3)</code> <code>region dod(image4)</code> <code>region dod(imageRef)</code>	Returns the domain of definition of the supplied image. This call can be made only within the <code>needed()</code> and <code>changed()</code> functions.

FLASH PLAYER NOTE: Region functions are not available when Pixel Bender is used in Flash Player.

Sampling functions

Each sampling function takes an image of a particular number of channels and returns a pixel with the same number of channels. All pixels outside the image's domain of definition are treated as transparent black.

<code>pixel1 sample(image1 im, float2 v)</code> <code>pixel2 sample(image2 im, float2 v)</code> <code>pixel3 sample(image3 im, float2 v)</code> <code>pixel4 sample(image4 im, float2 v)</code>	Handles coordinates not at pixel centers by performing bilinear interpolation on the adjacent pixel values.
<code>pixel1 sampleLinear(image1 im, float2 v)</code> <code>pixel2 sampleLinear(image2 im, float2 v)</code> <code>pixel3 sampleLinear(image3 im, float2 v)</code> <code>pixel4 sampleLinear(image4 im, float2 v)</code>	Same as <code>sample()</code> functions.
<code>pixel1 sampleNearest(image1 im, float2 v)</code> <code>pixel2 sampleNearest(image2 im, float2 v)</code> <code>pixel3 sampleNearest(image3 im, float2 v)</code> <code>pixel4 sampleNearest(image4 im, float2 v)</code>	Performs nearest-neighbor sampling.

Intrinsic functions

Pixel Bender includes these functions that allow access to the system's compile-time or run-time properties.

<code>float2 outCoord()</code>	Returns the coordinate of the midpoint of the output pixel currently being evaluated, as an <code>(x,y)</code> pair within a <code>float2</code> object. This call can be made only within the <code>evaluatePixel()</code> function or a function called by <code>evaluatePixel()</code> .
<code>int arrayVariable.length()</code>	Returns the number of elements of an array.

These functions access the pixel-size and aspect ratio of individual pixels or of images:

<pre>float2 pixelSize(image1) float2 pixelSize(image2) float2 pixelSize(image3) float2 pixelSize(image4)</pre>	<p>Returns the pixel size of an input or output image (which applies to all pixels in that image). The returned vector is (x,y), for the horizontal and vertical size.</p> <p>The standard pixel size is (1,1). Pixels are not necessarily square; many video applications use non-square pixels.</p>
<pre>float2 pixelSize(pixel1) float2 pixelSize(pixel2) float2 pixelSize(pixel3) float2 pixelSize(pixel4)</pre>	<p>Returns the pixel size of an individual pixel declared as output of a kernel. The returned vector is (x,y), for the horizontal and vertical size.</p>
<pre>float pixelAspectRatio(image1) float pixelAspectRatio(image2) float pixelAspectRatio(image3) float pixelAspectRatio(image4) float pixelAspectRatio(pixel1) float pixelAspectRatio(pixel2) float pixelAspectRatio(pixel3) float pixelAspectRatio(pixel4)</pre>	<p>Returns the aspect ratio of an input or output image, or of an individual pixel.</p> <p>For a square pixel the aspect ratio is 1:</p> <pre>pixelAspectRatio(i) == pixelSize(i).x / pixelSize(i).y</pre>

FLASH PLAYER NOTE: The `pixelSize()` and `pixelAspectRatio()` functions are available in Flash Player; however Flash Player always uses 1 x 1 pixels. The function `pixelSize()` always returns (1.0, 1.0), and `pixelAspectRatio()` always returns 1.0.

5 Pixel Bender Graph Language

The Pixel Bender graph language allows you to connect multiple Pixel Bender kernels into a *processing graph* that can be treated as a single entity, in order to create more sophisticated image processing effects.

A Pixel Bender graph is a directed acyclic graph (DAG), in which no loops are allowed. There is a single input and a single output node. The single output node can have multiple output images; however the output images all have the same size and position in world coordinate space. Multiple output images are treated as a single image with many channels.

FLASH PLAYER NOTE: Graphs are not supported in Flash Player.

Graph elements

The Pixel Bender graph language is an XML-based language that describes the structure of a graph. It allows you to declare a set of nodes, specify the connections between those nodes, and supply parameters.

A Pixel Bender graph definition contains these XML elements:

graph	The top-level container of a graph definition, with header information.
metadata	The namespace and graph version information.
parameter	A named value to be entered by the user, with optional constraints.
inputImage outputImage	The single input and output images for the graph.
kernel	A complete kernel definition, written in the Pixel Bender language.
node evaluateParameters	Defines a unique <i>instance</i> , or application, of one of the embedded kernels.
connect	Specifies one connection in the sequence of application of nodes between input and output.

Graph syntax

```
<?xml version="1.0" encoding="utf-8"?>
<graph name="graphName"
  languageVersion="1.0"
  xmlns="http://ns.adobe.com/PixelBenderGraph/1.0">

  <!-- Graph metadata -->
  <metadata name="namespace" value="graphNamespace" />
  <metadata name="vendor" value="value" />
  <metadata name="version" type="int" value="value" />

  <!-- Graph parameters (zero or more) -->
  <parameter type="dataType" name="name" >
    <metadata name="defaultValue" type="dataType" value="value"/>
  </parameter>
</graph>
```

```

        <metadata name="maxValue" type="dataType" value="value" />
        <metadata name="minValue" type="dataType" value="value" />
    </parameter>

    <!-- Image inputs and outputs (one or more inputs, one or more outputs) -->
    <inputImage type="imageType" name="name" />
    <outputImage type="imageType" name="name" />

    <!-- Embedded kernels (one or more) -->
    <kernel>
        <![CDATA[
            <languageVersion : 1.0;>
            kernel name
            < namespace: "kernelNamespace";
              vendor: "value";
              version: kernelVersion;
            >
            {
                input imageType name;
                output imageType name;
                [...parameters ...]
                void evaluatePixel()
                {
                    ...function definition...
                }
                [...other functions...]
            }
        ]]>
    </kernel>

    <!-- Nodes (one or more) -->
    <node id="nodeID" name="kernelName" vendor="kernelVendor"
        namespace="kernelNamespace" version="kernelVersion" />

    <!-- Connections (two or more) -->
    <connect fromImage="graphInputImage"
        toNode="nodeName" toInput="nodeInputImage" />
    <connect fromNode="nodeName" fromOutput="nodeOutputImage"
        toNode="nodeName" toInput="nodeInputImage" />
    <connect fromNode="nodeName" fromOutput="nodeOutputImage"
        toImage="graphOutputImage" />

</graph>

```

Graph header

```

<?xml version="1.0" encoding="utf-8"?>
<graph name="graphName"
    languageVersion="1.0"
    xmlns="http://ns.adobe.com/PixelBenderGraph/1.0">

```

Pixel Bender graphs always use version 1.0 of XML and are UTF-8 encoded.

The `graph` element is the top-level container for the graph. It must specify the name of the graph, the version of the graph language, and the XML namespace.

Graph element reference

The elements that can be contained in a `graph` element are presented here in order of appearance.

graph

```
<graph name="graphName" languageVersion="1.0"
  xmlns="http://ns.adobe.com/PixelBenderGraph/1.0">
```

name	Required. The name of this graph. The graph name is typically used as the filter name by Adobe applications that use this graph as a filter definition.
languageVersion	Required. The version of the graph language implementation in which this graph is written. For the meanings of version values in different contexts, see "Version identification in graphs" on page 39 .
xmlns	Required. The XML namespace for the graph language. This is constant, and different from the graph namespace. The XML namespace for version 1.0 is always "http://ns.adobe.com/PixelBenderGraph/1.0".

The `graph` element contains all the other elements in these sections:

Graph metadata	A set of metadata elements that supplies the namespace and graph version information.
Graph parameters	Zero or more parameter elements that supply named values to be entered by the user, with optional constraints.
Graph input and output images	The inputImage and outputImage elements that specify one or more input and one or more output images.
Embedded kernels	One or more complete kernel definitions, written in the Pixel Bender language. The kernels define their own namespaces, parameters, and input and output images.
Graph nodes	One or more node elements. Each node specifies a unique <i>instance</i> , or application, of one of the embedded kernels.
Graph connections	A set of connect elements that specify the sequence of application of nodes between input and output.

metadata

```
metadata name="propName" value="propValue" [type="dataType"] />
```

name	Required. The property key.
value	Required. The property value. To initialize vector or matrix metadata values, separate the individual values with commas and optional whitespace. For example: <pre><metadata name="preferredSize" type="int2" value="100, 150"/></pre>
type	Optional. The data type of the value. If not supplied, the value is assumed to be a UTF-8 string. Available data types are: <pre>int, int2, int3, int4 float, float2, float3, float4 float2x2, float3x3, float4x4 bool, bool2, bool3, bool4 pixel1, pixel2, pixel3, pixel4</pre>

The metadata section, along with the name specified in the graph header, provides a globally unique way of identifying a kernel or a graph. Three metadata elements are predefined and required for graphs:

```
<metadata name="namespace" value="value" />
<metadata name="vendor" value="value" />
<metadata name="version" type="int" value="value" />
```

- The `vendor` value should be the name of the company producing the graph, or perhaps a domain name if there is no obvious or unique company name.
- The `namespace` value is used to distinguish between different teams or products within a single company or vendor. For example, Adobe might use product names such as Photoshop and After Effects as namespace values.
- The `version` value is for this graph definition, and does not refer to either the language version or any related product version.

You can define additional metadata properties for a graph as needed.

The `metadata` element can also be contained in a [parameter](#) element, in which case it supplies the parameter constraints. These are typically:

```
<metadata name="minValue" type="dataType" value="value" />
<metadata name="maxValue" type="dataType" value="value" />
<metadata name="defaultValue" type="dataType" value="value" />
```

parameter

```
<parameter type="dataType" name="paramName" />
```

type	Required. The data type of the parameter. Available data types are: int, int2, int3, int4 float, float2, float3, float4 float2x2, float3x3, float4x4 bool, bool2, bool3, bool4 pixel1, pixel2, pixel3, pixel4
name	Required. The unique identifying name.

The optional parameters section defines non-image parameters that must be supplied to the graph. Graph parameters can be accessed by name and used to set kernel parameters, using the [evaluateParameters\(\)](#) function in a [node](#) definition.

Like kernel parameters, a graph parameter can contain optional `metadata` elements that describe the constraints. Generally, numeric parameters should provide default, maximum and minimum values to assist the host application in displaying an appropriate UI for the user to enter values.

For example:

```
<parameter type="float" name="scale" >
  <metadata name="minValue" type="float" value="0.0" />
  <metadata name="maxValue" type="float" value="100.0" />
  <metadata name="defaultValue" type="float" value="1.0" />
</parameter>
```

inputImage

```
<inputImage type="dataType" name="imageName" />
```

type	Required. The data type of the image. Available data types are: image1, image2, image3, image4
name	Required. The unique identifying name.

A graph must have at least one input image.

outputImage

```
<outputImage type="image4" name="dst" />
```

type	Required. The data type of the image. Available data types are: image1, image2, image3, image4
name	Required. The unique identifying name.

A graph must have at least one output image. Unlike a kernel, which can output a pixel or an image, a graph always outputs images.

A graph can have any number of input and output images, but it can only have one output *node*. All of the output images must come from the same output node.

kernel

```
<kernel>
  <![CDATA[
    <languageVersion : 1.0;>
    kernel name
    < namespace: "value";
      vendor: "value";
      version: value;
    >
    {
      input type name;
      output float4 dst;
      void evaluatePixel()
      {
        ...
      }
    }
  ]]>
</kernel>
```

All of the kernels that a graph requires must be embedded in the graph program. The syntax for the kernel definition is the same as that of a standalone kernel definition, except that it is wrapped in a `kernel` tag and `![CDATA]` statement.

Each kernel has its own identifying information (name, namespace, vendor, and version). You should take care to make the kernel identification globally unique.

node

```
<node
  id="nodeID"
  name="kernelName"
  vendor="kernelVendor"
  namespace="kernelNamespace"
  [version="kernelVersion"] >
  <evaluateParameters>
    <![CDATA[
      void evaluateParameters()
      {
        ...
      }
    ]]>
  </evaluateParameters>
</node>
```

id	Required. A unique identifying string for this node.
name	The name and other identifying metadata of the kernel that is invoked in this node.
vendor	The <code>version</code> attribute is optional; if not supplied, the highest available version is used.
namespace	
version	

A node is a unique instance of a kernel; that is, one application of the kernel's operation within the graph sequence. If the sequence requires that a kernel be called multiple times, you must define multiple nodes for that kernel.

If no kernel with matching identification is found in the graph, a load error occurs.

If the kernel being instantiated by a node requires parameters, the node must contain an [evaluateParameters](#) element.

evaluateParameters

```
<evaluateParameters>
  <![CDATA[
    void evaluateParameters()
    {
      nodeName::paramName=statement;
      ...
    }
  ]]>
</evaluateParameters>
```

The `evaluateParameters` child element contains an `evaluateParameters()` function definition.

The `evaluateParameters()` function is the graph's equivalent to the kernel's `evaluateDependents()` function. This function can use all of the non-image related syntax available in Pixel Bender. It has access to the graph's declared parameters, and can use them to set values for the kernel's parameters.

All kernel parameters must be set by the node's `evaluateParameters()` function, even if they have default values associated with their metadata.

connect

```
<connect fromImage="imgName" | fromNode="nodeID" fromOutput=" "
        toImage="imgName" | toNode="nodeID" toInput=" " />
```

fromImage	Specifies the source of the connection as one of the graph's input images, declared in the graph's inputImage element.
fromNode fromOutput	Specifies the source of the connection as a node, and specifies which output image of that node is passed to the destination. fromNode: The node ID of a node whose output supplies the image. fromOutput: A specific node output image. This is the name of an output image variable declared in the node's kernel definition.
toImage	Specifies the destination of the connection as the graph's output image, declared in the graph's outputImage element.
toNode toInput	Specifies the destination of the connection as a specific node, and specifies which input image of that node receives the source image. toNode: The node ID of a node that performs this operation. toInput: A specific node input image. This is the name of an input image variable declared in the node's kernel definition.

Each connection in the graph is specified by one `connect` element. The connection has a "from" side and a "to" side. There are three kinds of connections:

Graph input image to node input image	<ul style="list-style-type: none"> ➤ The "from" side is <code>fromImage</code>, an image declared in the graph's inputImage element. ➤ The "to" side is a node, which performs a kernel operation and produces an image result. Because a node can produce multiple image results, you must specify both <code>toNode</code> and <code>toInput</code>.
Node output image to node input image	<ul style="list-style-type: none"> ➤ The "from" side is a specific image output from a node, specified by <code>fromNode</code> and <code>fromOutput</code>. ➤ The "to" side is another node, which performs an operation and produces a specified image result. You specify both <code>toNode</code> and <code>toOutput</code>.
Node output image to graph output image	<ul style="list-style-type: none"> ➤ The "from" side is a specific image output from a node, specified by <code>fromNode</code> and <code>fromOutput</code>. ➤ The "to" side is <code>toImage</code>, an image declared in the graph's outputImage element.

A connection can have only one source or destination node. However, because a node can accept multiple input images and produce multiple output images, you must specify the image being passed, as well as the node. Images are referenced by the variable names assigned in the [kernel](#) definition.

The graph is a directed acyclic graph (DAG); it must not contain loops.

Version identification in graphs

A `version` attribute appears in various elements in a Pixel Bender graph file, meaning different things depending on the context. This section provides a short guide to the different version contexts:

```
<?xml version="1.0" encoding="utf-8"?>
```

This indicates that the graph file conforms to version 1.0 of the Extensible Markup Language specification.

```
<graph name="scale"
  languageVersion="1.0"
  xmlns="http://ns.adobe.com/PixelBenderGraph/1.0">
```

The `languageVersion` attribute of the `graph` tag refers to the version of the Pixel Bender graph language in which this program is written.

```
<!-- Graph metadata -->
<metadata name="namespace" value="AIF Test" />
<metadata name="vendor" value="Adobe" />
<metadata name="version" type="int" value="1" />
```

The `version` metadata property refers to the version of the filter defined by this graph. As you develop and improve your filter, you can use this to identify the most recent iteration or release of the code.

```
<kernel>
  <![CDATA[
    <languageVersion : 1.0;>
    kernel Blend
    < namespace: "Pixel Bender Tutorial";
      vendor: "Adobe";
      version: 1;
    >
    . . .
```

- The `languageVersion` within a kernel refers to the version of the Pixel Bender language that the kernel is written in.
 - The `version` metadata property in a kernel refers to the version of the filter defined by this kernel.
-