ADOBE® PIXEL BENDER®

PIXEL BENDER DEVELOPER'S GUIDE

Copyright © 2010 Adobe Systems Incorporated. All rights reserved.

Adobe Pixel Bender Developer's Guide.

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, Flex, 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 6

1 **Overview of the Pixel Bender Toolkit** 8
Installing the Pixel Bender Toolkit 8
Contents of the Pixel Bender Toolkit package 9
Getting started 9
Samples 10
Pixel Bender Concepts 10
File formats 11
Kernel program parts 11
Built-in functions 12
Using Pixel Bender filters and effects 12
Developing for Photoshop 13
Developing for After Effects 14
Developing for Flash 15

2 **Getting Started** 19
Becoming familiar with the Pixel Bender Toolkit IDE 19
Creating a new Pixel Bender kernel program 20
Edit and run the kernel program 20
Split the image channels 21
Change the filter operation to gamma 21
Add a parameter 21
Constrain the parameter 22
Final Source 23

3 **Writing Pixel Bender Filters** 24
Parts of a kernel 24
Kernel metadata 24
Kernel members 25
Parameters and variables 26
The Pixel Bender coordinate system 27
Accessing pixel coordinates 28
Passing in coordinates 30
Input images and sampling 31
Non-square pixels 33
Multiple input images 36
Using dependent values 36
Hints and tips 38
Undefined results 38
Inexact floating-point arithmetic 38
Out-of-range output pixel values 38
Array sizes 39

Support functions 39
Example of region functions 39
Example of a support function 41

4 Working with Regions 43

How Pixel Bender runs a kernel 43
Region reasoning in graphs 44

Defining regions 44
Adjusting image size to allow for edge calculations 45
Conservative region bounds 46

Computing the needed region 47
Accessing kernel parameters in the needed() function 48
Using the DOD to calculate the needed region 48

Computing the changed region 49
A more complex example 50

Computing the generated region 51

5 Pixel Bender Graph Language 53

Graph elements 53
Simple graph example 54
Complete simple example 56

A complex graph 57
Defining the graph 58
Defining graph and kernel parameters 58
Defining multiple nodes 59
Defining complex connections 60
Complete complex example 61

Using the Graph Editor 65

6 Metadata Hinting 66

Defining parameter metadata for a filter 66

Value constraint elements 66

Display string elements 67
Examples 67
Localizing display strings 68

Parameter type specialization element 69
Examples 70
Enumerated values 71
Distinguishing among multiple input images 71
Naming vector values 72

7 Developing for After Effects 73

After Effects kernel metadata 73

Accessing 4-channel values 73

An example of convolution 74

Kernel parameters in After Effects 76
Expanded convolution kernel 77
Dependent functions 79

8 Developing for Flash 82

Using the Flash preview feature in the Pixel Bender Toolkit 82
Embedding a Pixel Bender filter in a SWF 83
Making Pixel Bender filters into ActionScript libraries 83
Encapsulate a filter in an ActionScript class 83
Create the SWC library 85
Using Pixel Bender kernels as blends 85

Preface

The Adobe® Pixel Bender® technology delivers a common image and video processing infrastructure which provides automatic runtime optimization on heterogeneous hardware. You can use the Pixel Bender kernel language to implement image processing algorithms (*filters* or *effects*) in a hardware-independent manner.

The Pixel Bender Toolkit includes the Pixel Bender kernel language and graph language, the Pixel Bender Toolkit IDE (an integrated development environment for Pixel Bender), sample filters, and documentation.

This document, *Pixel Bender Developer's Guide*, provides an introduction to the Pixel Bender Toolkit IDE, as well as tutorials and examples of how to use the Pixel Bender kernel and graph languages to develop filters. It is intended for programmers who wish to develop image filters for Adobe products such as After Effects®, Flash®, and Photoshop®.

A companion document, *Pixel Bender Reference*, is a complete reference manual and specification for the Pixel Bender kernel and graph languages.

Pixel Bender development offers many advantages:

- **Low learning curve** Pixel Bender offers a small number of tools that are sufficient to write complex image-processing algorithms. Learning Pixel Bender is easier than learning C/C++ and each application's plug-in SDK. You do not need to know any graphics shading language or multi-threading APIs.
- **Parallel processing** Pixel Bender allows the same filter to run efficiently on different GPU and CPU architectures, including multi-core and multiprocessor systems. It delivers excellent image processing performance in Adobe products,
- **Supports all bit-depths** The same kernel runs in 8-bit/16-bit/32-bit within the application.
- **Support by multiple Adobe applications** Pixel Bender is integrated with multiple Adobe applications. It allows you to develop filters that are portable among various Adobe products. There is an active Pixel Bender Exchange where developers share their filters.

Pixel Bender is best suited for the kind of algorithms in which processing of any pixel has minimum dependence on the values of other pixels. For example, you can efficiently write a kernel to manipulate brightness of the image because brightness of each pixel can be changed independently. You would not want to use Pixel Bender to compute a histogram, because a histogram requires the values of all the pixels in an image.

Pixel Bender has a familiar C-like syntax that is based on OpenGL Shading Language (GLSL). This guide assumes a basic knowledge of programming and image processing, as well as familiarity with the intended target application.

- For background information on image processing basics, see:
  - *Image Processing Fundamentals*: http://www.ph.tn.tudelft.nl/Courses/FIP/noframes/fip.html
- For more advanced material on image processing, see:
  - *Digital Image Processing (3rd Edition),* Rafael C. Gonzalez and Richard E. Woods, Prentice Hall, 2006. ISBN 013168728X

- ▷ *Digital Image Warping*, George Wolberg, IEEE Computer Society Press, 1990, ISBN 0818689447, 9780818689444
- ▷ *Handbook of Image and Video Processing*, Alan Conrad Bovik, Academic Press, 2005 ISBN 0121197921, 9780121197926

1 Overview of the Pixel Bender Toolkit

The Pixel Bender Toolkit includes:

- The Pixel Bender kernel language, a high-performance graphics programming language intended for image processing.
- The Pixel Bender graph language, an XML-based language for combining individual pixel-processing operations (*kernels*) into more complex filters.
- The Pixel Bender Toolkit IDE, an interactive development environment in which to create, compile and preview Pixel Bender kernels.
- A command-line utility for converting a Pixel Bender kernel file into a byte-code file that can be used in Flash Player 10 or later.
- Sample filters, sample images, and documentation.

The Pixel Bender Toolkit IDE allows you to edit and execute programs that use the Pixel Bender image-processing engine. The Pixel Bender run-time engine is integrated into a number of Adobe applications, allowing you to add your Pixel Bender filters and effects to those already offered in those applications.

For more resources to get you started with Pixel Bender development and the Pixel Bender Toolkit, go to the Pixel Bender Technology Center:

http://www.adobe.com/devnet/pixelbender/

Installing the Pixel Bender Toolkit

Download the Pixel Bender Toolkit from the Pixel Bender Technology Center.

To install the Pixel Bender Toolkit in Windows:

1. Download the Pixel Bender Toolkit ZIP file.
2. Unzip the package, navigate to the unzipped location, and run `Setup.exe`.

- In a 32-bit version of Windows, the Pixel Bender Toolkit files are installed under:

  `Program Files\Adobe\Adobe Utilities — CS5\Pixel Bender Toolkit 2`

- In a 64-bit version of Windows, the Pixel Bender Toolkit files are installed under:

  `Program Files (x86)\Adobe\Adobe Utilities — CS5\Pixel Bender Toolkit 2`

To install the Pixel Bender Toolkit in Mac OS:

1. Download the Pixel Bender Toolkit DMG file.
2. Double-click on the DMG item to mount the installer virtual disk.
3. In that virtual disk, double-click to launch the Setup application.

The Pixel Bender Toolkit files are installed into your `Applications` folder under:

```
/Applications/Utilities/Adobe Utilities — CS5/Pixel Bender Toolkit 2
```

Contents of the Pixel Bender Toolkit package

The Pixel Bender Toolkit contains these folders and files (locations are relative to the download location, *PBTKroot*):

At the root level, in addition to the shared libraries:	
`PixelBenderToolkitReadMe.pdf`	A brief introduction to the Pixel Bender Toolkit.
`Pixel Bender Toolkit.exe`	The executable for the Pixel Bender Toolkit IDE.
`pbutil.exe`	A command-line utility that converts a Pixel Bender kernel file into a byte-code file that can be used in Flash Player. See "File formats" on page 11.
`PixelBenderUtilityReadMe.pdf`	Instructions for using the command-line utility.
Additional folders:	
`docs/`	Documentation, including this document, *Pixel Bender Developer's Guide*, and the *Pixel Bender Reference*.
`legal/`	Licenses.
`pixel bender files/`	Sample programs; see "Samples" on page 10.
`sample images/`	A set of sample images for use in learning and testing.

Getting started

1. Launch the Pixel Bender Toolkit IDE.
   - In Windows, launch from the Start menu.
   - In Mac OS, launch from the Applications folder:

   ```
   /Applications/Utilites/Adobe Utilities — CS5/Pixel Bender Toolkit 2/Pixel Bender
   Toolkit
   ```

2. To load a Pixel Bender kernel file (`.pbk`) into the Pixel Bender Toolkit IDE, choose File > Open Filter, or press Ctrl-O.
3. To try the program, click Build and Run at the bottom right corner of the Pixel Bender Toolkit IDE, beneath the code editor window.
   - If the program requires an image for processing, you are prompted to open an image file.
   - If the program has parameters, editing controls (in the right-edge panel) allow you to enter them.

For an introduction to the development environment and a walkthrough of how to use it, see Chapter 2, "Getting Started."

Samples

The Pixel Bender Toolkit contains these sample programs that demonstrate various techniques and usage:

alphaFromMaxColor	Estimates alpha based on color channels.
BasicBoxBlur SimpleBoxBlur	Takes an average of the pixels (+1) surrounding the current coordinate to give a blurred output. Demonstrates how to allow for non-square pixels. BasicBoxBlur uses loops, which are not available in Flash Player. The SimpleBoxBlur version implements the same operation without loops, and does work with Flash.
Checkerboard	Combines two input images to create a checkerboard effect.
CombineTwoInputs	Merges two input images into the output image.
Crossfade	Performs a fade between two images.
inFocus	Creates a clear spot, which is followed by a fade-off region. The rest of the image is blurred.
InvertRGB	Inverts the red, green, and blue channels of an image.
Pixelate	Demonstrates sampling.
Sepia	Performs a simple color-space transformation that results in a sepia-toned image.
sepia-twirl	Changes image to sepia coloration, then creates a twirl effect in the image.
Twirl	Creates a twirl effect in the image that the user can adjust by assigning values to the radius and center point parameters.

Pixel Bender Concepts

The basic unit of image processing in Pixel Bender is the *kernel*. The job of a kernel is to produce a single output pixel. A typical Pixel Bender kernel:

- Samples one or more pixels from one or more input images
- Applies a calculation to the sampled pixels' colors
- Assigns the result to the output pixel

Each program in the Pixel Bender kernel language defines one named 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 different input images.

Each kernel must contain an `evaluatePixel()` function definition. This function is executed in parallel over all desired output pixels, to generate an output image. The kernel can take any number of parameters of any Pixel Bender data types.

A Pixel Bender kernel is run once for every pixel in the output image. No state information is saved between runs, so it is not possible to collect an average pixel value by accumulating the pixels sampled in each run. Each run of the kernel must compute all the information it needs—although you can pre-compute information and pass it in as an input or parameter.

You can use the Pixel Bender *graph language* to connect a sequence of kernels, in order to create more sophisticated image processing effects. Graphs are discussed in Chapter 5, "Pixel Bender Graph Language."

File formats

Pixel Bender defines three file formats: PBK, PBJ, and PBG.

- A kernel program is stored in a plain-text file with the extension `.pbk`.
- For use with Flash Player, a kernel must be exported to a *byte-code program*, stored in a binary file with the extension `.pbj`. The Pixel Bender Toolkit IDE provides an export command, or you can use the provided command-line conversion utility, `pbutil.exe`.
- A graph description is stored in a plain-text file with the extension `.pbg`.

Kernel program parts

The simplest Pixel Bender program consists of a kernel that returns a solid color everywhere:

```
<languageVersion : 1.0;>
kernel FillWithBlack
<
    namespace : "FillWithBlack";
    vendor : "Pixel Bender Guide";
    version : 1;
    description : "simplest kernel";
>
{
    output pixel4 dst;

    void evaluatePixel()
    {
        dst = pixel4(0,0,0,0);
    }
}
```

header

metadata section enclosed in < >

kernel definition enclosed in { }

- The `languageVersion` element, and the `namespace`, `vendor`, and `version` metadata values are required. When you create a new kernel in the Pixel Bender Toolkit IDE, it fills in the necessary infrastructure, which you can modify as needed.
- This kernel definition uses a Pixel Bender data type, `pixel4`, which holds a 4-channel pixel, typically describing red, green, blue, and alpha channels.

This slightly more interesting example takes in an input image `src` and darkens each pixel to create an output image.

```
<languageVersion : 1.0;>
kernel DarkenedOutput
<   namespace : "Tutorial";
    vendor : "Pixel Bender Guide";
    version : 1;
    description : "slightly more complex kernel";
>
{
    input image4 src;
    output pixel4 dst;
```

```
    void evaluatePixel() {
        dst = 0.5 * sampleNearest(src, outCoord());
    }
}
```

- In this case, the `evaluatePixel()` function reads the nearest current input pixel by calling one of the built-in image-sampling functions, `sampleNearest()`.
- The function multiplies the pixel value by 0.5 to darken the image. It does this using the `*` operation, which performs the multiplication on each channel of the `pixel4` value.

Notice that these programs define an output *pixel*, not an output *image*. The kernel has access to every pixel in the input image (although in this case it uses only one pixel), but can write only a single pixel of output. The Pixel Bender run-time engine runs the kernel once for every pixel in the output image.

Built-in functions

The Pixel Bender kernel language provides many built-in functions for common pixel-manipulation operations. Built-in functions fall into these categories:

Function type	Description
Mathematical	These functions convert between degree and radian measurements, get trigonometric values, and perform basic mathematical operations such as logs, powers, sign manipulations, comparisons, and so on. There are also functions to perform various kinds of interpolation.
Geometric	These functions operate on vectors, performing matrix arithmetic and comparisons. (Pixel Bender defines 2-, 3-, and 4-element vector types.)
Region	These functions query and transform rectangular regions. They operate on the `region` data type.
Sampling	Each sampling function takes an image of a particular number of channels and returns a pixel with the same number of channels. This is how a program reads data from an input image.
Intrinsics	These functions allow access to the system's compile-time or run-time properties. They include, for example, `dod()` (domain of definition), and `pixelSize()`.

The usage of many of these functions is demonstrated in the sample programs that are included with the Pixel Bender Toolkit.

Using Pixel Bender filters and effects

The Pixel Bender filters and effects are supported in these Adobe applications:

- Adobe Photoshop CS4 or later (through the Pixel Bender Plug-in)
- Adobe After Effects CS4 or later (the Pixel Bender Toolkit is installed automatically with After Effects)
- Adobe Flash Player and Adobe Flash CS4 or later (the Pixel Bender Toolkit is installed automatically with Flash)

Each application implements the Pixel Bender run-time engine differently. Each implementation offers some application-specific features, and has some limitations. In some cases, parts of your Pixel Bender programs (such as display names) are interpreted at run time in a way that allows each application to present Pixel Bender filters in a UI that is seamlessly integrated with built-in filters and effects. In fact, some built-in filters and effects are already implemented using Pixel Bender.

To find existing Pixel Bender filters and effects that have been developed by third parties, go to the Pixel Bender Exchange.

With a little forethought, you can create filters that are portable among these applications. This section provides an overview of the differences between the application platforms, and later chapters provide application-specific examples and recommendations.

Developing for Photoshop

The Pixel Bender Plug-in for Adobe Photoshop CS4 or later supports processing of Pixel Bender filters on images opened in Photoshop. You can download this plug-in from the Pixel Bender Technology Center.

The Pixel Bender Plug-in installer is provided as an Extension Manager installer; Extension Manager CS4 or later is required for proper installation. Pixel Bender filters can be executed on the graphics card (GPU) or CPU of a computer. The plug-in ReadMe file lists the supported graphics cards.

The plug-in creates the Pixel Bender Gallery. To use the Pixel Bender Plug-in for Photoshop:

1. Launch Adobe Photoshop.
2. Open an image.
3. Choose Filter > Pixel Bender > Pixel Bender Gallery.

Adding filters

The Pixel Bender Plug-in supports both kernel (PBK) and graph (PBG) programs.

The Pixel Bender Gallery is initially populated with Pixel Bender filters that are installed with the Pixel Bender Plug-in. Filters included in the download have been authored by Pixel Bender customers as well as members of the Adobe Pixel Bender team.

To install additional filters in the Pixel Bender Gallery:

1. Close the Gallery.
2. Copy the Pixel Bender program into the appropriate folder for your platform:
   - In Windows XP:
     `C:\Documents and Settings\`*`username`*`\My Documents\Adobe\Pixel Bender\`
   - In Windows Vista or Windows 7:
     `C:\Users\`*`username`*`\My Documents\Adobe\Pixel Bender\`
   - In Mac OS:
     `~/Documents/Adobe/Pixel Bender/`

The filter becomes available in the Pixel Bender Gallery the next time the plug-in is initialized.

Creating Smart Objects

The Pixel Bender Plug-in supports non-destructive rendering on Smart Objects. To create a Smart Object:

1. Open an image.
2. Select a layer.
3. From the Layer fly-out menu, choose "Convert to Smart Object."

The rendering of the filter is stored in a layer separate from the original image and may be subsequently modified if the filter is available.

Photoshop features and limitations

- The plug-in supports RGB-8 and 16-bit opaque and transparent images. If an unsupported image type is opened, the Pixel Bender Gallery is disabled in the menu, and Pixel Bender filters cannot be applied to the image. You must convert the image to a supported type in order to use a Pixel Bender filter on it.
- Each filter must contain a unique kernel name. If the plug-in encounters kernels with the same kernel name, the last kernel compiled is displayed in the Pixel Bender Gallery.

Developing for After Effects

Pixel Bender is seamlessly integrated with After Effects CS4 or later. You can use it to create effects without being familiar with the After Effects Plug-in SDK.

After Effects supports both the PBK and PBG file formats for Pixel Bender kernel and graph programs. All kernel and graph programs that are found in the Pixel Bender `Plug-ins` folder appear as effects in the Effects and Presets panel. They work just like effects written using the After Effects Plug-in SDK. Kernel or graph parameters are exposed in the Effect Controls panel, and can be animated like any other After Effects parameters.

Adding effects

To load a Pixel Bender kernel or graph program as an After Effects effect:

1. Make sure After Effects is not running.
2. Create a Pixel Bender folder at this location:
   - In Windows XP:
     `C:\Documents and Settings\`*`username`*`\My Documents\Adobe\Pixel Bender\`
   - In Windows Vista or Windows 7:
     `C:\Users\`*`username`*`\My Documents\Adobe\Pixel Bender\`
   - In Mac OS:
     `~/Documents/Adobe/Pixel Bender/`
3. Place the kernel or graph program file in the new folder.
4. Launch After Effects.

After Effects features and limitations

- 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.

- After Effects supports only 4-channel input and output. You must convert any kernel with less than 4-channel input or output in order to use it in After Effects.
- After Effects defines additional parameter metadata to emulate its own set of effect parameters. See Chapter 7, "Developing for After Effects."
- You must define the `needed()` and `changed()` functions for any non-pointwise filters. If you do not, the kernel performs inefficiently and generates incorrect images, even if it appears to be working. See "Defining regions" on page 44.
- After Effects processes video data, which commonly uses non-square pixels. Your kernels must take the pixel size and aspect ratio into account. See "Non-square pixels" on page 33.
- After Effects supports high dynamic range (HDR) images; you cannot assume that image data is always in the range 0.0 to 1.0.
- After Effects does not support the infinite region generator, `everywhere()`. 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.

More information

- For additional information on using Pixel Bender in Adobe After Effects, see:

  http://help.adobe.com/en_US/AfterEffects/9.0/

Developing for Flash

Pixel Bender effects in Flash Player 10 or later can be applied to any display object, including images, vector graphics, and even digital video. The execution speed is extremely fast; effects that would have taken seconds-per-frame to execute in ActionScript can be achieved in real time with Pixel Bender kernels.

When you save a kernel program, it is saved by default to a PBK-format file. For use with Flash, it must be exported to the PBJ format.

- In the Pixel Bender Toolkit IDE, choose File > Export Kernel Filter for Flash Player. This command compiles and exports the kernel for use in Flash Player, to a file with the extension `.pbj`.
- A command-line conversion utility, `pbutil.exe`, is provided with the Pixel Bender Toolkit, with instructions in the associated read-me file. The utility converts an input PBK file to the PBJ format.

Loading a kernel in Flash Player

The compiled byte code in a PBJ file can be loaded or embedded in a Shader object and used by your SWF content.

- You can load kernels at runtime. This example uses the `URLLoader` class to load a kernel:

```
var camellia_mc:MovieClip;

var urlRequest:URLRequest = new URLRequest( "channelscrambler.pbj" );
var urlLoader:URLLoader = new URLLoader();
urlLoader.dataFormat = URLLoaderDataFormat.BINARY;
urlLoader.addEventListener( Event.COMPLETE, applyFilter );
urlLoader.load( urlRequest );

function applyFilter( event:Event ):void {
    trace("apply");
    urlLoader.removeEventListener( Event.COMPLETE, applyFilter );
    var shader:Shader = new Shader( event.target.data );
    var shaderFilter:ShaderFilter = new ShaderFilter( shader );
    camellia_mc.filters = [ shaderFilter ];
}
```

- The `Embed` tag (supported in Flash) offers the easiest way to load a kernel. It instructs the ActionScript compiler to embed the Pixel Bender kernel when it creates the SWF file. The `Embed` tag is used with a variable definition of type `Class`:

```
[Embed(source="myFilter.pbj", mimeType="application/octet-stream")]
var MyFilterKernel:Class;
```

  To use the kernel, create an instance of the class (in this example, `MyFilterKernel`). For example, the following code uses an embedded kernel to create new `Shader` and `ShaderFilter` objects, which are applied to a `MovieClip` instance on the Stage:

```
var camellia_mc:MovieClip;
//Embed the PixelBender kernel in the output SWF
[Embed(source="myFilter.pbj", mimeType="application/octet-stream")]
var MyFilterKernel:Class;

var shader:Shader = new Shader(new MyFilterKernel() );
var shaderFilter:ShaderFilter = new ShaderFilter( shader );
camellia_mc.filters = [ shaderFilter ];
```

When you use the `Embed` tag, Flash uses the `Flex.swc` library from the Flex® SDK. This SDK is installed with Flash, typically in the folder `Common/Configuration/ActionScript 3.0/libs/flex_sdk_3`. The first time you test or publish a movie using the `Embed` tag, you are asked to confirm the location of the Flex SDK. You can click OK to use the Flex SDK installed with Flash, or you can change the path if you prefer to use a different version of Flex. The setting used for a project can be changed later on the Advanced ActionScript 3.0 Settings dialog box, under the Library path tab. (Access the Advanced ActionScript 3.0 Settings dialog box from the Flash tab of the Publish Settings by clicking the Settings button next to the Script drop-down menu.)

Accessing parameters in ActionScript

The following parameter statement declares a `float3` parameter with metadata:

```
parameter float3 weights
<
defaultValue : float3( 0.5, 0.5, 0.5 );
    minValue :   float3( 0.1, 0.1, 0.1 );
    maxValue :  float3( 0.9, 0.9, 0.9 );
    description : "A three element vector of weight values."
>;
```

To access parameter values in Flash Player, you use the `data` property of the ActionScript `Shader` object containing the kernel. In ActionScript, you can access the current, minimum and maximum values of this parameter like this (where `myShader` is a `Shader` object containing a kernel with this parameter):

```
var currentWeights:Array = myShader.data.weights.value;
var minWeights:Array = myShader.data.weights.minimumValue;
var maxWeights:Array = myShader.data.weights.maximumValue;
```

Because this parameter is a `float3` vector type, the returned ActionScript arrays contain three elements. For a scalar type, such as `float`, each returned array contains a single element.

Unlike ActionScript, Pixel Bender does not do any implicit data conversion. When you type a floating point number in a Pixel Bender program, you must include a decimal point; otherwise, the Pixel Bender compiler treats the number as an integer rather than a float value. If, for example, you type 1 instead of 1.0, the type conflict will eventually cause an error, but the error message may not be very useful in identifying the problem.

Flash features and limitations

- Pixel Bender in Flash Player does not support:
  - loops or control structures other than `if` and `else`.
  - custom support functions and libraries
  - region functions
  - arrays
  - dependent values
  - non-constant indices into vector values

  When developing Pixel Bender kernels for Flash Player using the Pixel Bender Toolkit IDE, always enable Flash warnings (Build > Turn on Flash Player Warnings and Errors). With this option enabled, the compiler informs you immediately when you are using Pixel Bender kernel language features that are not supported in Flash Player. Otherwise, these errors are not reported until you try to export the kernel for Flash Player.

- Pixel Bender images have 32 bits per channel, but graphics in Flash 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.

- Flash Player always uses 1x1 square pixels.

- Pixel Bender graphs are not supported by Flash Player.

More information

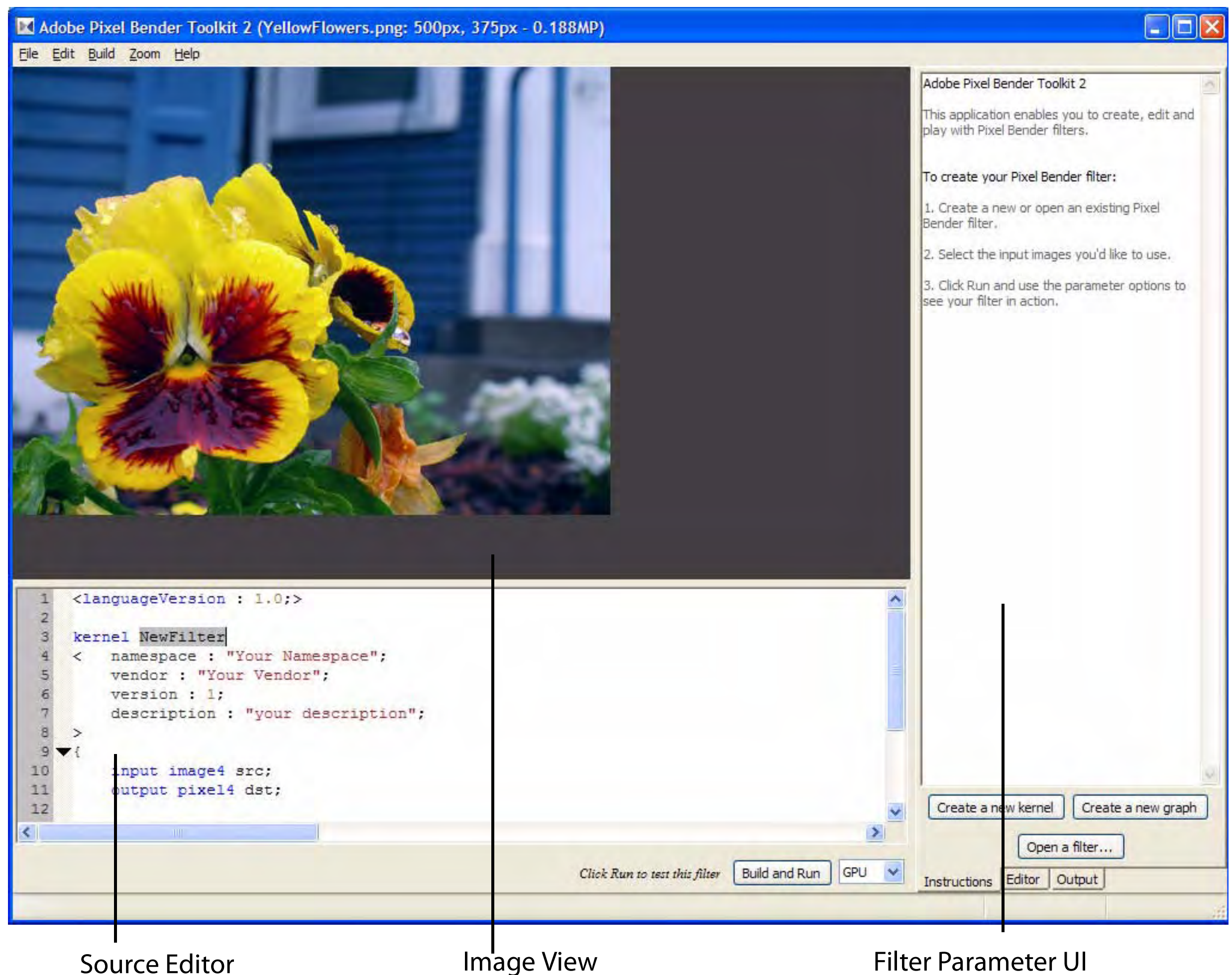
- For additional information on using Pixel Bender in Flash, see the Programming ActionScript 3.0 chapter Working with Pixel Bender shaders and the ActionScript Component and Language Reference Shader class section. This documentation contains a detailed description of the objects you can use with Pixel Bender in Flash.
- For some examples that focus on Flash and ActionScript, see Chapter 8, "Developing for Flash."
- You can find a public repository of kernels hosted by Adobe at the Pixel Bender Exchange. The authors of these kernel programs have kindly agreed to share them with the wider Pixel Bender developer community. There is also a forum for discussing Pixel Bender programming.

2 Getting Started

This chapter introduces the Pixel Bender Toolkit IDE and walks through the process of creating a new kernel filter, using Pixel Bender kernel language.

Becoming familiar with the Pixel Bender Toolkit IDE

The Pixel Bender Toolkit IDE is divided into three areas:



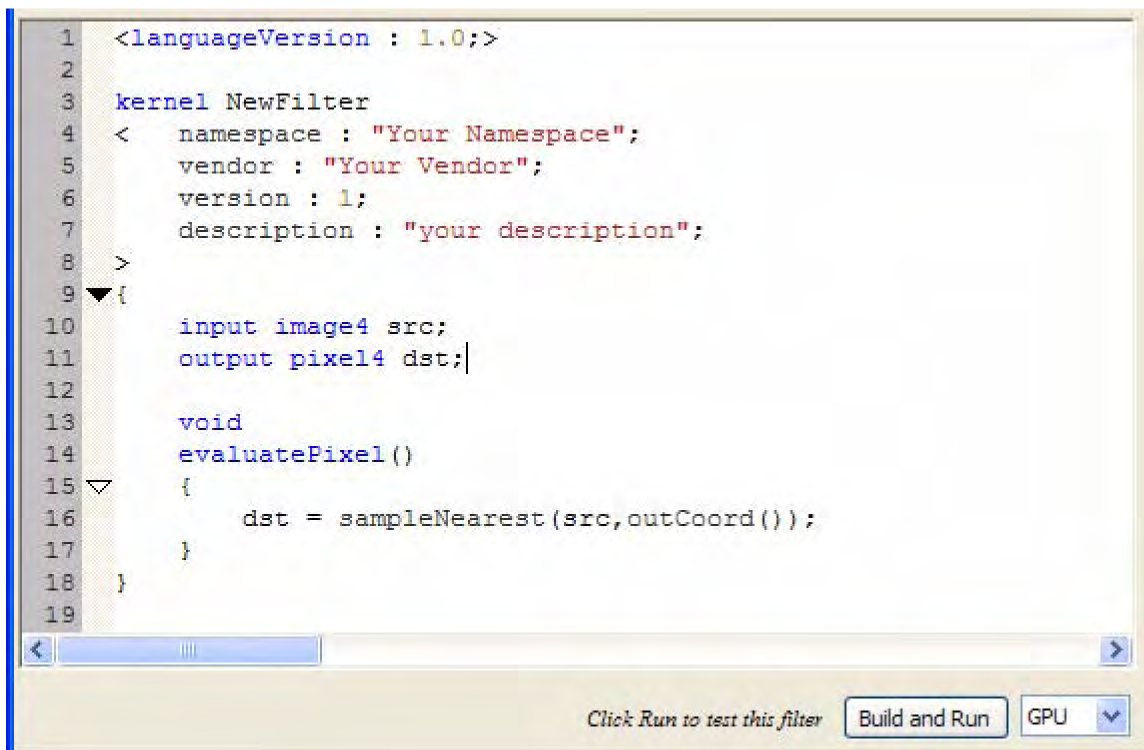
- The Source Editor area is where you enter and edit Pixel Bender source code. It provides basic code development capabilities, such as syntax highlighting and code completion.
- The Filter Parameter UI area is where the IDE displays the user interface that it automatically creates for any parameters that you expose.
- The Image View area shows the filter results on loaded images.

If a filter has not been loaded, the panel displays the image file that you load using File > Load Image 1. This command opens a file browser, in which you can navigate to an image of your choice. The Toolkit supports the JPEG and PNG file formats.

Creating a new Pixel Bender kernel program

To create a new kernel program, you can click "Create a new kernel", or choose File > New Kernel.

The Source Editor displays the basic structure of a Pixel Bender kernel. By default, the new filter contains the code necessary to perform a basic identity function on the image; that is, it performs no edits to the image, but includes all of the necessary infrastructure.



Edit and run the kernel program

Our first simple edit makes this generated program perform a basic filter operation.

▶ Edit the definition of the `evaluatePixel()` function:

```
evaluatePixel() {
    dst = 0.5 * sampleNearest(src, outCoord());
}
```

▶ Click Build and Run at the bottom right of the Source Editor to execute the program.

Running this filter reduces all of the channels (including the opacity) by one half. This has the effect of darkening the image. When you run the filter, you see the result on the image displayed above the program in the Image View area.

We will perform some additional modifications on this program to demonstrate some basic programming techniques.

Split the image channels

In this step, we will split up the input channels, so that we can darken only the color channels and leave the alpha channel (the image opacity) as it is.

▶ Edit the definition of the `evaluatePixel()` function:

```
evaluatePixel() {
    float4 inputColor = sampleNearest(src, outCoord());
    dst.rgb = 0.5 * inputColor.rgb;
    dst.a = inputColor.a;
}
```

This saves the input color in a temporary variable of type `float4`, which contains the red, green, and blue channels in the first three elements and the alpha channel in the last element. We can than access these individual values with the dot operator, using the letters `r`, `g`, `b`, and `a`.

Change the filter operation to gamma

In this step, we will change the algorithm from a simple scaling operation to a simple gamma change. To do this, we will use a built-in function, `pow()`, to do an exponential calculation on the color channels. (See the *Pixel Bender Reference* for a list of other built-in functions.)

▶ Edit the definition of the `evaluatePixel()` function:

```
evaluatePixel() {
    float4 inputColor = sampleNearest(src, outCoord());
    float3 exp = float3(0.5);
    dst.rgb = pow(inputColor.rgb, exp);
    dst.a = inputColor.a;
}
```

Notice that we are passing in all three color channels at once to a single function. Many built-in functions, including `pow()`, have the ability to operate on vector values in a component-wise fashion; that is, by working on each value in the vector individually. Both vector arguments to `pow()` must have the same number of elements for the operation to be applied correctly. The function `float3(0.5)` is a quick way of creating a three-element floating-point vector initialized to `(0.5, 0.5, 0.5)`.

NOTE: The `pow()` result is undefined for an `inputColor` value that is less than 0. This can happen in After Effects, or any program that supports high dynamic range (HDR) images.

Add a parameter

It makes sense for the amount of the gamma change, which is now a constant, to be a variable value provided by the user. In this step, we will turn the gamma value into a user-controlled parameter.

▶ Add the following code just before the `evaluatePixel ()` function:

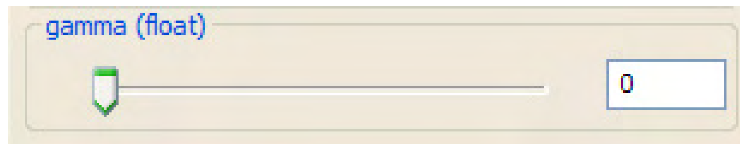
```
{
    input image4 src;
    output pixel4 dst;
    parameter float gamma<
        minValue: 0.0;
        maxValue: 1.0;
        defaultValue: 0.0;
    >;
```

```
    void
    evaluatePixel ()
    {...}
```

▶ Edit the definition of the `evaluatePixel()` function:

```
void
evaluatePixel() {
    float4 inputColor = sampleNearest(src, outCoord());
    dst.rgb = pow(inputColor.rgb, float3(1.0 - gamma));
    dst.a = inputColor.a;
}
```

▶ Click Build and Run. In the Filter Parameter UI section of the IDE, you now see a slider for the gamma parameter.



You can use this control to adjust the image. The entered value is subtracted from 1.0, so a positive value here brightens the image. As you drag the slider, you see the brightness change in the Image View.

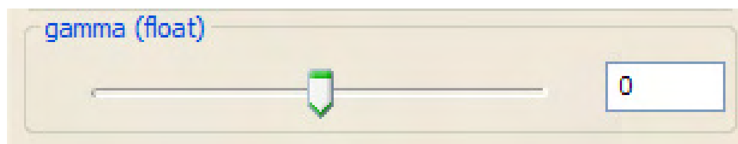
Constrain the parameter

So far, the range of gamma is 0 to 1, so we can only brighten the image, not make it darker. The next step is to broaden the gamma range by adding constraints around the parameter.

▶ Change the parameter specification to add the constraints:

```
parameter float gamma
<
    minValue:float(-0.5);
    maxValue:float(0.5);
    defaultValue:float(0.0);
>;
```

▶ Click Run. In the Filter Parameter UI section of the IDE, you now see that slider shows 0 in the middle, instead of the beginning of the value range.



Now you can drag the slider to the left to darken the image, or to the right to brighten it.

Final Source

This is the program that we have created:

```
<languageVersion : 1.0;>

kernel NewFilter
<   namespace : "your namespace";
    vendor : "your vendor";
    version : 1;
    description : "your description";
>

{
    input image4 src;
    output pixel4 dst;
    parameter float gamma
    <
        minValue:float(-0.5);
        maxValue:float(0.5);
        defaultValue:float(0.0);
    >;

    void
    evaluatePixel() {
        float4 inputColor = sampleNearest(src, outCoord());
        dst.rgb = pow(inputColor.rgb, float3(1.0 - gamma));
        dst.a = inputColor.a;
    }
}
```

3 Writing Pixel Bender Filters

This chapter discusses some of the unique features of Pixel Bender kernel-language programming.

Parts of a kernel

The basic unit of image processing in Pixel Bender is the kernel. Each Pixel Bender kernel language program defines one kernel, specified by a single string that contains the language-version element and the kernel statement.

```
<languageVersion : 1.0;>     ◄── required language-version element

kernel name
<
    kernel metadata pairs    ◄── metadata section in angle brackets
>
{
    kernel members           ◄── variables and functions section in curly braces
}
```

The kernel statement contains a name, 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.

Kernel metadata

The language-version element is required before each kernel definition. This statement and the metadata section (which you can modify) are supplied automatically when you create a new filter in the Pixel Bender Toolkit IDE.

The metadata section provides the namespace, a kernel version, and other identifying and descriptive information. This is particularly important when you are collecting several kernels into *graphs* to perform a more complex operation. For example:

```
<
    namespace : "Tutorial";
    vendor : "Adobe";
    version : 1;
    description: "My Filter";
>
```

The `namespace`, `vendor`, and `version` values are required; the `description` is optional.

- The `vendor` is the name of the company or individual who writes the filter.
- The `version` is an integer version number which should start at 1 and increase as new versions of the kernel are introduced. This allows you to produce new versions of a filter with bug fixes or improved performance, while still leaving the old versions available.

- The namespace is a way for a company or author to further segregate filters. For example, Adobe might have different versions of the gaussian blur filter for Photoshop and After Effects, and use the name of the product in the namespace field to distinguish them:

  ```
  kernel GaussianBlur
  <
     namespace : "Photoshop";
     vendor : "Adobe Systems";
     version : 1;
  >
  {
  // ... The Gaussian blur filter as used by Photoshop
  }
  kernel GaussianBlur
  <
     namespace : "After Effects";
     vendor : "Adobe Systems";
     version : 1;
  >
  {
  // ... The Gaussian blur filter as used by After Effects
  }
  ```

  The `namespace` value is used in combination with the other filter identifiers to determine the actual namespace, so it need not be globally unique.

For brevity in this chapter, only the kernel name and members section are shown; to try the examples, you can paste the kernel member section into a kernel definition that contains a language-version statement and metadata section.

Kernel members

A kernel is defined like a class in C++, with member variables and functions. The kernel members section contains a set of declarations, and a set of function definitions. Every kernel must provide at least the `evaluatePixel()` function and at least one output parameter of type `pixel`.

The simplest Pixel Bender program consists of a kernel that returns a solid color everywhere:

```
kernel FillWithBlack
< ... >
{
   output pixel4 dst;

   region generated()
   {
      return everywhere();
   }

   void evaluatePixel()
   {
      dst = pixel4(0,0,1,0);
   }
}
```

This kernel produces one output image with four channels (red, green, blue, alpha), as specified by the declaration `output pixel4 dst`. Because a kernel is executed for all pixels of the output image, each pixel output parameter defines an entire image.

Pixel Bender is a strongly typed language. In addition to the standard numeric (scalar) types, it defines a set of vector types for pixels and images with 1, 2, 3, or 4 members, or channels. For a complete listing and description of the Pixel Bender data types, see the *Pixel Bender Reference*.

Parameters and variables

In the declarations before the function definitions, you can specify *parameters*, which are passed in to the kernel program and have fixed values within the kernel, and *dependent variables*, which assigned values once using the `evaluateDependents()` function, and are then read only (see "Using dependent values" on page 36).

A kernel can take any number of parameters of arbitrary types. The parameters are passed to the Pixel Bender run-time system, and their values are held constant over all pixels, much like the "uniform" variables used in 3D shading languages.

The application in which a kernel is running provides a UI in which the user can set the parameters values. For example, it may show a dialog when the filter is invoked. During development, the Pixel Bender Toolkit IDE provides this UI.

This example adds a parameter to the `FillWithBlack` kernel, which is used to define a fill color other than black:

```
kernel FillWithColor
< ... >
{
    parameter pixel4 color;
    output pixel4 dst;

    region generated()
    {
        return everywhere();
    }

    void evaluatePixel()
    {
        dst = color;
    }
}
```

Remember that all Pixel Bender programs must specify the version of the Pixel Bender kernel language in which they are written, using the `languageVersion` statement. To try this code, paste it into a kernel program that contains the necessary infrastructure.

The data type of the parameter provides a clue to the host application about what kind of control is appropriate for setting the parameter value. You typically add constraints to the parameter value that help even more. For example, this parameter definition allows the UI to show the minimum and maximum allowed values, as well as the initial default value:

```
parameter pixel4 color
<
    minValue: float4(0.0,0.0,0.0,0.0);
    maxValue: float4(1.0,1.0,1.0,1.0);
    defaultValue: float4(0.0,0.0,0.0,1.0);
>;
```

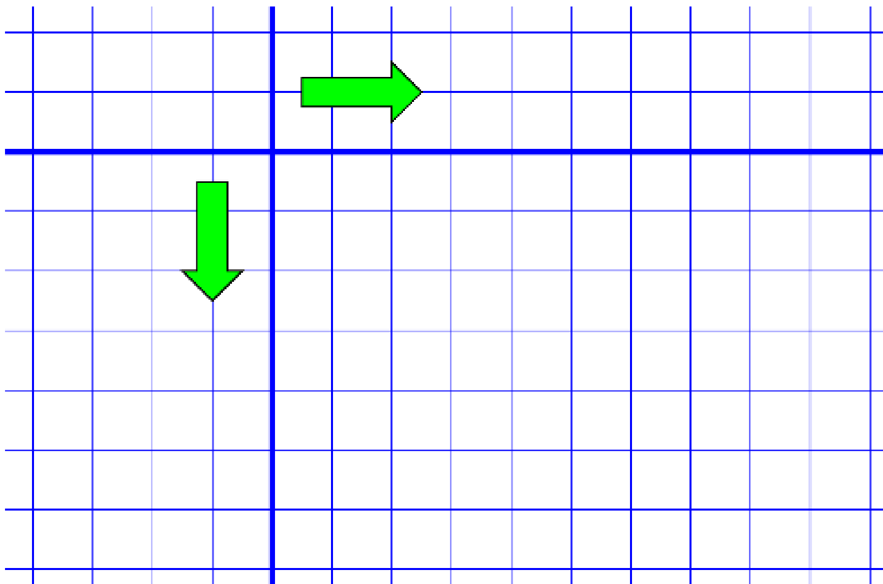
```
1  <languageVersion : 1.0;>
2
3  kernel NewFilter
4  <   namespace : "Your Namespace";
5      vendor : "Your Vendor";
6      version : 1;
7      description : "your description";
8  >
9  {
10     parameter pixel4 color;
11     output pixel4 dst;
12
13     void
14     evaluatePixel()
15     {
16         dst = color;
17     }
18 }
19
```

The filter is running | Build and Run | GPU

To try out these examples in the Pixel Bender Toolkit IDE, paste the body of code into the members section of a kernel definition, and fill in the kernel name.

The Pixel Bender coordinate system

Pixel Bender has a single, world coordinate system (sometimes called *world space*). The world coordinate system is square, uniform (isotropic, cartesian, and orthogonal), and infinite. The X axis increases to the right and the Y axis increases downwards:



The origins of input and output images are aligned to the origin of the world coordinate system.

In the Pixel Bender model, *images* do not have a size. Instead, each image is thought of as being defined over an infinite plane of discrete pixel coordinates. The Pixel Bender run-time engine that executes the kernel determines the size of the buffers needed to store and operate on the pixels. Only individual pixels have a size.

This Pixel Bender coordinate model has implications for filter design. For example, it is not possible to write a Pixel Bender kernel that reflects an image around its "center," because the center is undefined. Instead, the coordinates of the center must be denoted explicitly by passing them in as kernel parameters (see "Passing in coordinates" on page 30).

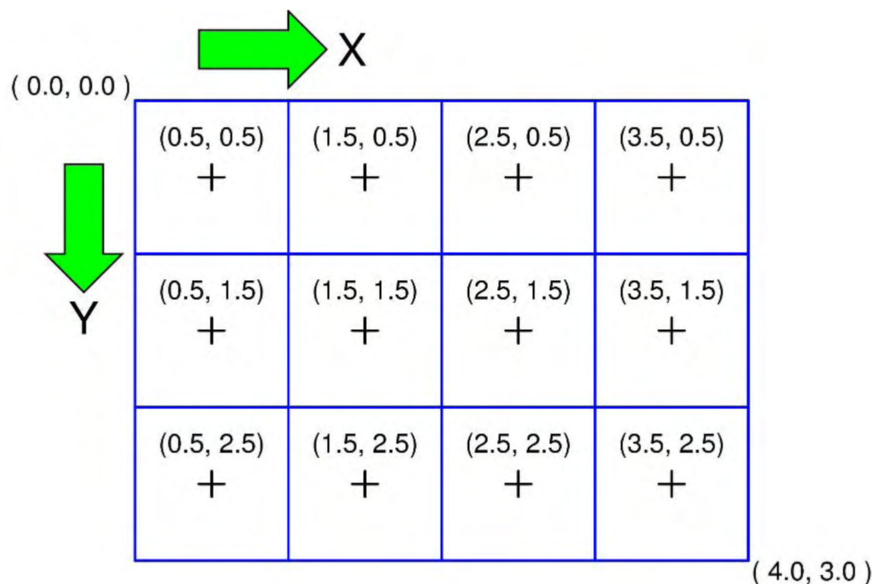
No coordinate transformation is possible; an image is transformed by being explicitly resampled to the output grid. This restriction may seem harsh, but it corresponds to how atomic image-processing operations must execute in practice. More sophisticated reasoning about coordinate systems—such as concatenation and reordering of sequential transforms—must be performed at a higher level, before kernel execution.

Accessing pixel coordinates

A kernel is executed in parallel over all pixels of the output image, with exactly the same parameter values for each pixel. The only thing that changes at each pixel is the coordinate of the current output pixel.

To access the current coordinate value, use the built-in function `outCoord()`. This function returns a value of type `float2` (a vector of two floats) that gives the (x, y) coordinates of the center of the output pixel being evaluated by the current invocation of the `evaluatePixel()` function. The current output coordinate varies across the pixels of the image, but is invariant during the lifetime of any one call to `evaluatePixel()`.

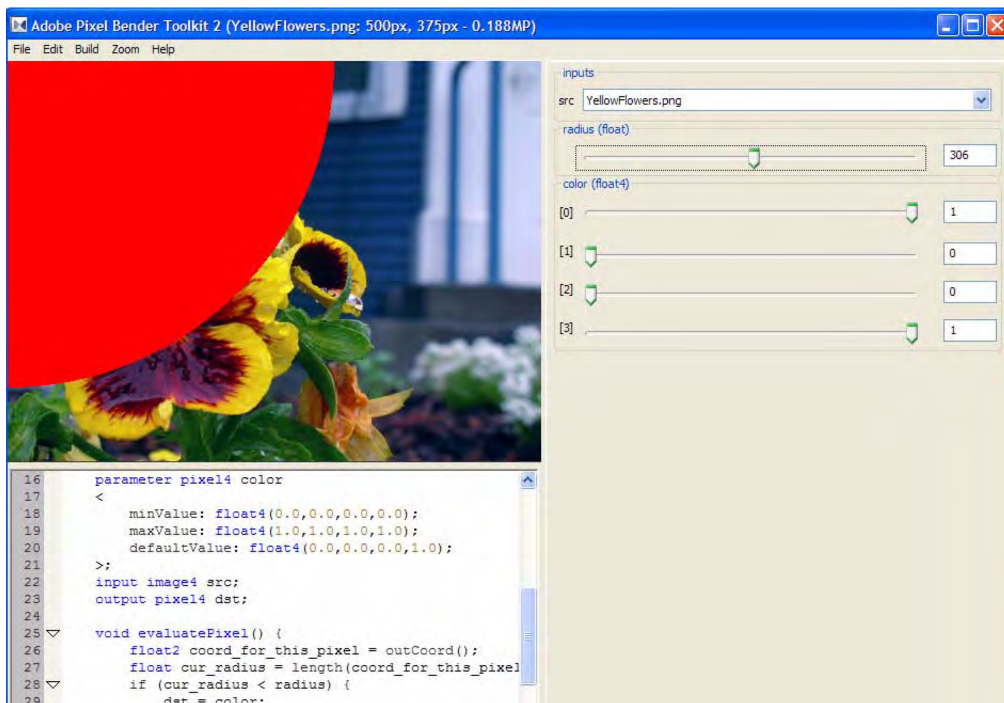
If we assume that the pixels are square (they are not necessarily; see "Non-square pixels" on page 33) the `outCoord()` function returns these values for a 4 x 3 pixel output image:



The following example demonstrates the `outCoord()` function, using it to produce a pattern in the output image. It also demonstrates the use of the vector type `float2` and the vector function `length()`. The Pixel Bender kernel language includes a rich set of vector types and operations; for details, see the *Pixel Bender Reference*.

This kernel renders a (non-anti-aliased) filled circle centered at the origin, with a color and radius specified by parameters:

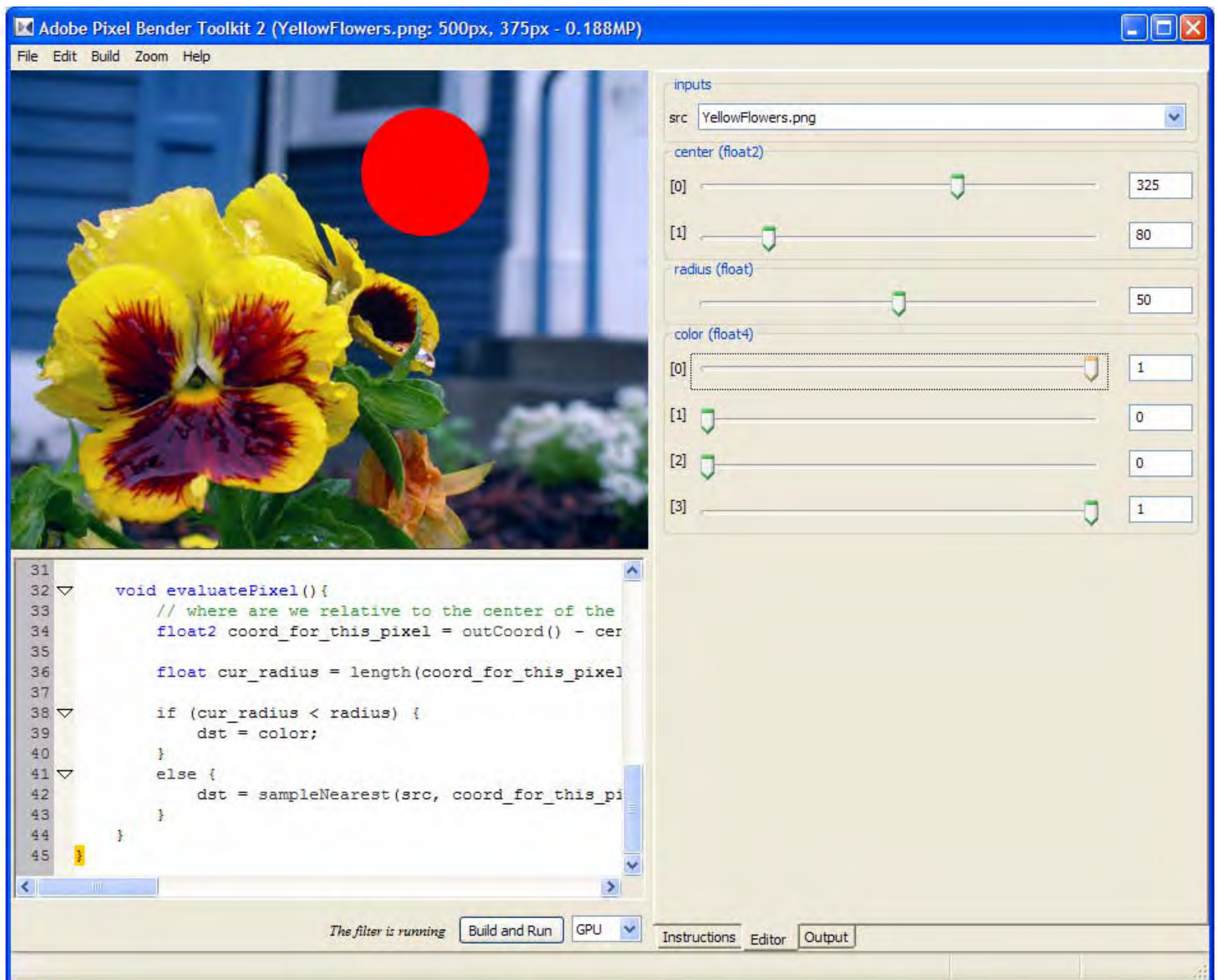
```
kernel RenderFilledCircle

< ... >
{
    parameter float radius
    <
        minValue: 0.0;
        maxValue: 600.0;
        defaultValue: 100.0;
    >;
    parameter pixel4 color
    <
        minValue: float4(0.0,0.0,0.0,0.0);
        maxValue: float4(1.0,1.0,1.0,1.0);
        defaultValue: float4(0.0,0.0,0.0,1.0);
    >;
    input image4 src;
    output pixel4 dst;

    void evaluatePixel() {
        float2 coord_for_this_pixel = outCoord();
        float cur_radius = length(coord_for_this_pixel);
        if (cur_radius < radius) {
            dst = color;
        }
        else {
            dst = sampleNearest(src, coord_for_this_pixel);
        }
    }
}
```



Passing in coordinates

This kernel renders a circle of a specified size and color, but instead of centering the circle at the origin, you pass a horizontal and vertical coordinate for the circle's center.

```
kernel PassCenterPoint
< ... >
{
    parameter float2 center
    <
        minValue: float2(0);
        maxValue: float2(500);
        defaultValue: float2(180);
    >;
    parameter float radius
    <
        minValue: 0.0;
        maxValue: 100.0;
        defaultValue: 30.0;
    >;
    parameter pixel4 color
    <
        minValue: float4(0.0,0.0,0.0,0.0);
        maxValue: float4(1.0,1.0,1.0,1.0);
        defaultValue: float4(0.0,0.0,0.0,1.0);
    >;

    input image4 src;
    output pixel4 dst;

    void evaluatePixel(){
        // where are we relative to the center of the circle
        float2 coord_for_this_pixel = outCoord() - center;

        float cur_radius = length(coord_for_this_pixel);

        if (cur_radius < radius) {
            dst = color;
        }
        else {
            dst = sampleNearest(src, coord_for_this_pixel + center);
        }
    }
}
```

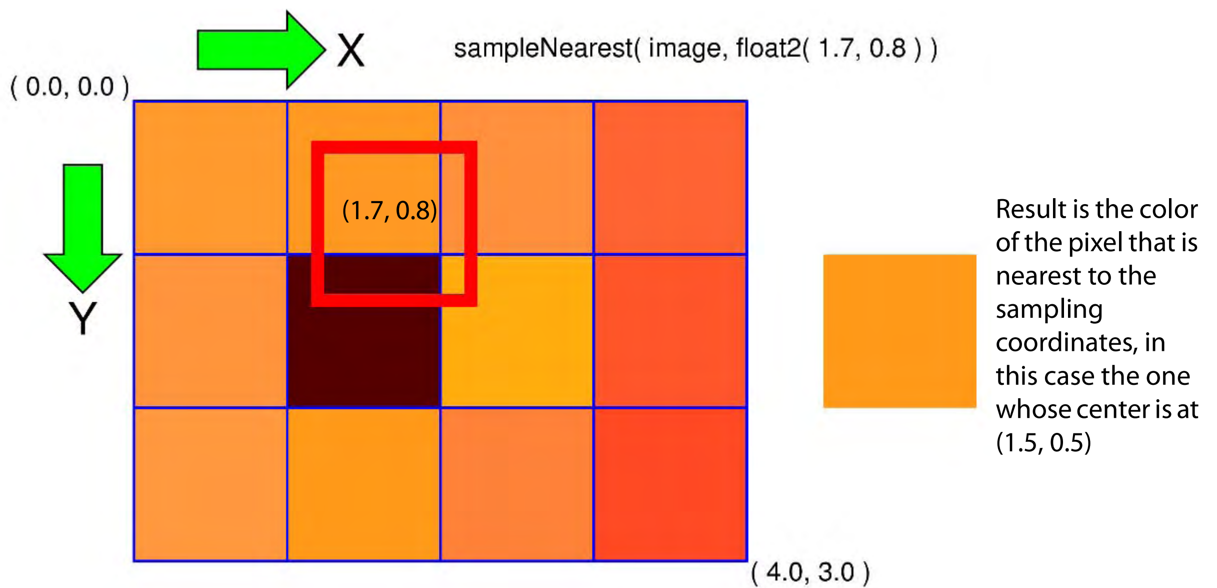


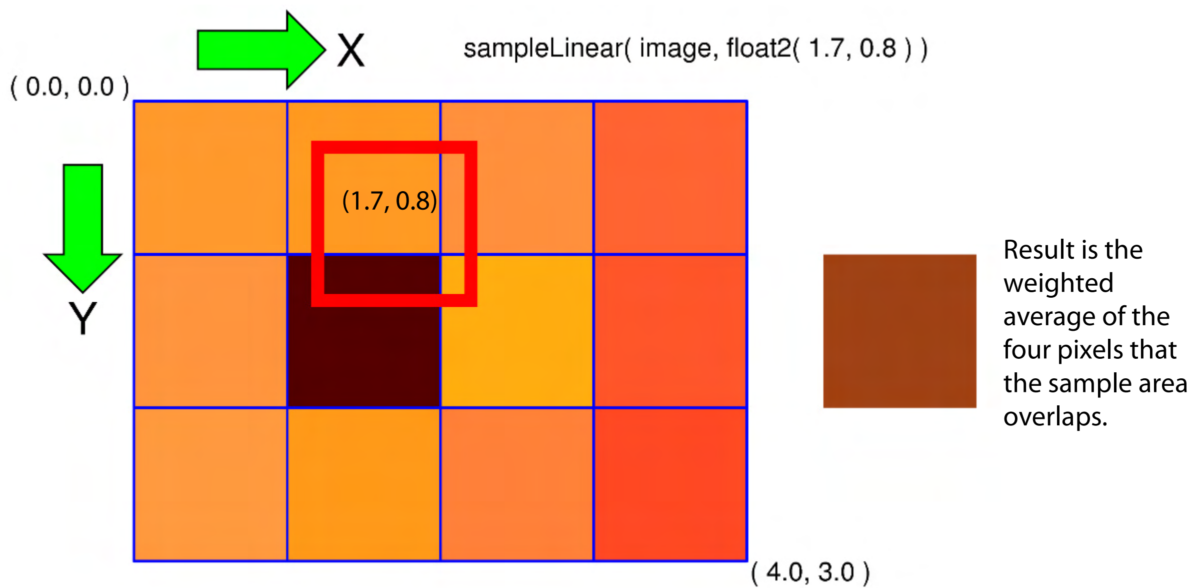
Input images and sampling

A kernel with no image inputs is termed a *source* or *generator*. A practical example of a generator is a procedural texture routine. However, most kernels take one or more image inputs.

You can declare one or more input images to a kernel, then access pixel values by passing an image and the pixel coordinates to one of the built-in sampling functions:

- The `sampleNearest()` function returns the value of the pixel whose midpoint is nearest the given coordinate.
- The `sampleLinear()` function performs bilinear interpolation between the four pixels adjacent to the given coordinate.





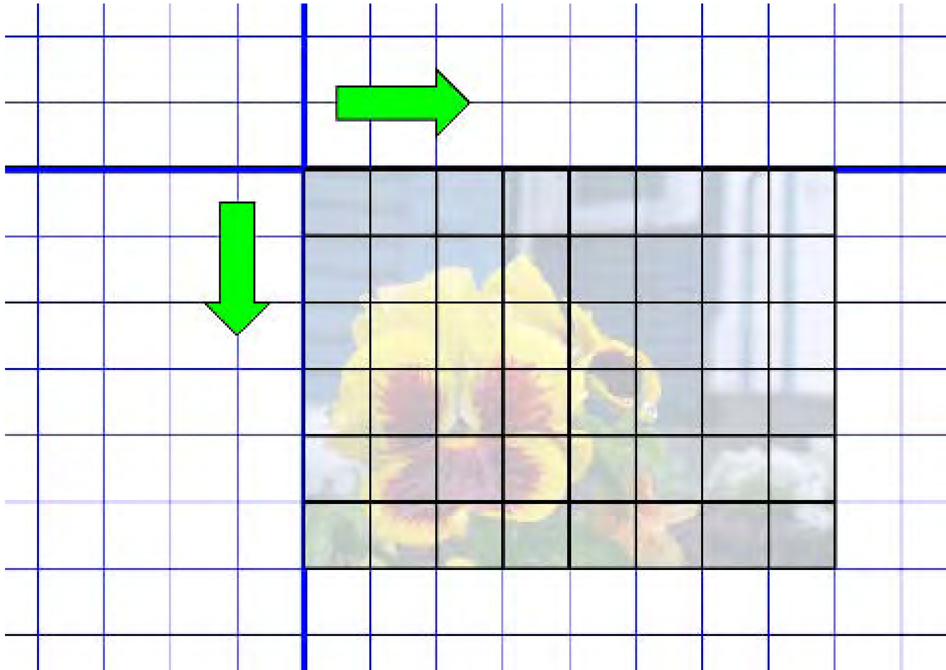
The simplest possible kernel with sampling leaves the image unchanged:

```
kernel Identity
< ... >
{
    input image4 source;
    output pixel4 dst;

    void evaluatePixel()
    {
        dst = sampleNearest( source, outCoord() );
    }
}
```

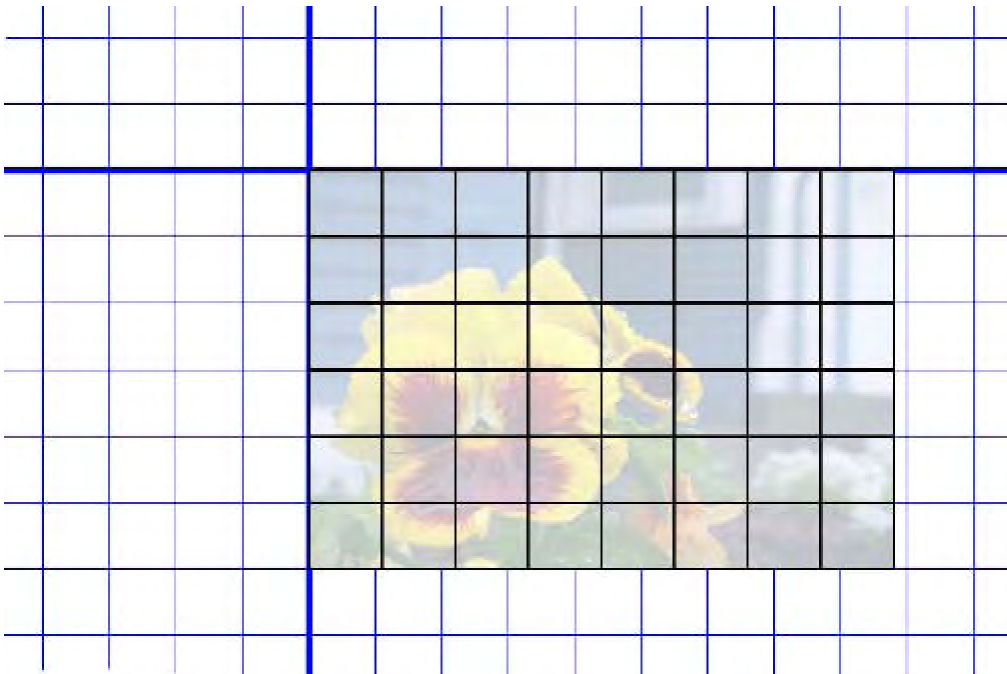
You can provide any coordinates to a sampling function; however, if you sample a point outside the defined area of an image, the function returns transparent black (0.0, 0.0, 0.0, 0.0). All of the sampling functions return a pixel value of exactly the same number of channels as the passed image.

Non-square pixels

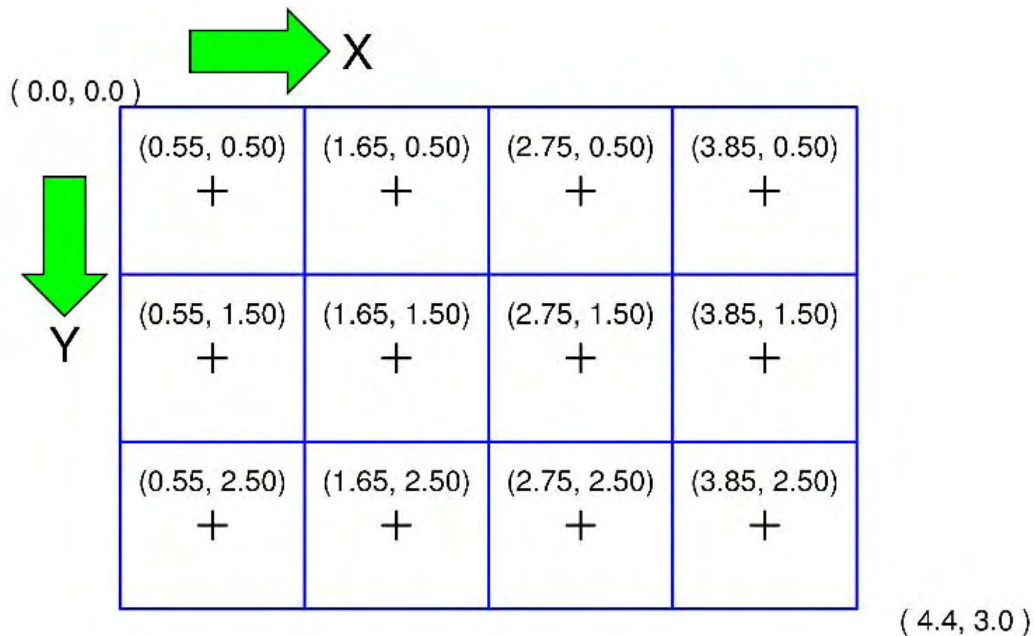
When you assume that pixels are square, the pixel grid of an image always lines up with the underlying world coordinate space:



In fact, however, pixels are not always square. For example, After Effects typically deals with video images that have non-square pixels. If pixels are 1.1 wide x 1.0 high, the pixel grid does not line up with world space:



Even though the size of the pixels has changed, all coordinates are still given in the world coordinate system. For example, the values returned by `outCoord()` for a 4 x 3 pixel image now look like this:



Within a kernel, all coordinates are given in the world coordinate space. This includes:

- The return value of the `outCoord()` function.
- The coordinates passed to the sampling functions.
- The regions passed to and returned from the region functions, including `needed()`, `changed()`, `generated()`, and `dod()`.

Any parameters that you pass to a kernel that measure distance or area (such as the radius of a blur) should also be specified in world coordinates.

A kernel can assume that the pixel size of the output image will be the same as the pixel size of the first input image. Beyond that, however, it cannot choose or make assumptions about the size of the pixels for any input or output images. The pixel sizes for all input images and the output image can be different. To discover the pixel size of any input or output image, use the `pixelSize()` and `pixelAspectRatio()` functions.

- The built-in function `pixelSize()` returns the pixel size for a given input or output image. The result is the size of the pixel measured in world coordinates.
- The built-in function `pixelAspectRatio()` gets the pixel width divided by the pixel height for a given image:

  ```
  pixelAspectRatio( i ) == pixelSize( i ).x / pixelSize( i ).y
  ```

  A square pixel has an aspect ratio of 1.

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

If there is a possibility that the pixels in your images are not square, you must take the pixel size into account to ensure accurate sampling results. To select adjacent pixels, for instance, get the pixel size of the input image and use it to modify the coordinates that you pass to the sampling function.

For example, this kernel averages each pixel with its immediate left and right neighbors, producing a slight horizontal blur:

```
kernel HorizontalAverage
< ... >
{
    input image4 source;
    output pixel4 result;

    void evaluatePixel()
    {
        float2 coord = outCoord();
        float2 hOffset = float2(pixelSize(source).x, 0.0);

        pixel4 left  = sampleNearest(source, coord - hOffset);
        pixel4 center= sampleNearest(source, coord);
        pixel4 right = sampleNearest(source, coord + hOffset);

        result = (left + center +  right)/3.0;
    }
}
```

This example uses `evaluateDependents()` to transform a radius that has been supplied in world coordinates into a radius measured in appropriately-sized pixels:

```
kernel HorizontalTentBlur3
< ... >
{
    input image4 src;
    output pixel4 dst;

    parameter float radius;

    dependent int intRadius;
    dependent float weights[ 100 ];

    void evaluateDependents() {
        float w = 1.0 / float( radius );
        float s = w / float( radius );
        intRadius = int( ceil( radius / pixelSize(src).x ) );
        weights[ 0 ] = w;
         for( int i = 1; i <= intRadius; ++i ) {
            w -= s;
            weights[ i ] = w;
        }
    }

    void evaluatePixel() {
        pixel4 total = sampleNearest( src, outCoord() ) * weights[ 0 ];
        for( int i = 1; i <= intRadius; ++i ) {
            float x = float( i ) * pixelSize( src ).x;
            total += sampleNearest( src, outCoord() + float2( x, 0 ) ) * weights[ i ];
            total += sampleNearest( src, outCoord() + float2( -x, 0 ) ) * weights[ i ];
    }
}
```

Multiple input images

A kernel can take any number of input images, each of which can have a different number of channels. The following kernel multiplies a four-channel image by a single-channel matte:

```
kernel MatteRGBA
< ... >
{
    input image4 source;
    input image1 matte;
    output pixel4 result;

    void evaluatePixel()
    {
        pixel4 in_pixel = sampleNearest( source, outCoord() );
        pixel1 matte_value = sampleNearest( matte, outCoord() );

        result = in_pixel * matte_value;
    }
}
```

Using dependent values

FLASH NOTE: Dependent values are not available when Pixel Bender is used in Flash Player.

Consider a convolution operation with a procedurally generated look-up table of pixel weights:

```
<languageVersion : 1.0;>
kernel GaussianBlur
<
    namespace:"Tutorial";
    vendor:"Adobe";
    version:1;
>
{
    input image4 source;
    output pixel4 result;

    void evaluatePixel()
    {
        const float sigma = 2.0;
        float c = 1.0 / ( sqrt(2.0 * 3.1415926535 ) * sigma );
        float ec = 2.0 * sigma * sigma;

        float weight0 = exp( -( 0.0 * 0.0 ) / ec ) * c;
        float weight1 = exp( -( 1.0 * 1.0 ) / ec ) * c;
        float weight2 = exp( -( 2.0 * 2.0 ) / ec ) * c;
        float weight3 = exp( -( 3.0 * 3.0 ) / ec ) * c;
        float weight4 = exp( -( 4.0 * 4.0 ) / ec ) * c;

        float4 acc = float4( 0.0 );
        acc += sampleNearest( source, outCoord() ) * weight0;
        acc += sampleNearest( source, outCoord() + float2( 1.0, 0.0 ) ) * weight1;
        acc += sampleNearest( source, outCoord() + float2( -1.0, 0.0 ) ) * weight1;
        acc += sampleNearest( source, outCoord() + float2( 2.0, 0.0 ) ) * weight2;
        acc += sampleNearest( source, outCoord() + float2( -2.0, 0.0 ) ) * weight2;
        acc += sampleNearest( source, outCoord() + float2( 3.0, 0.0 ) ) * weight3;
        acc += sampleNearest( source, outCoord() + float2( -3.0, 0.0 ) ) * weight3;
```

```
        acc += sampleNearest( source, outCoord() + float2( 4.0, 0.0 ) ) * weight4;
        acc += sampleNearest( source, outCoord() + float2( -4.0, 0.0 ) ) * weight4;

        result = acc;
    }
}
```

This code works but has a problem: the look-up table is regenerated on every pixel, which defeats the purpose of a look-up table. We want to precompute these values once, then apply them to all pixels. You could pass in the values using kernel parameters, but then you need external code to compute the values.

In order to keep the computation within the kernel, but perform it only once, you can declare the table as a set of dependent member variables. You then use the `evaluateDependents()` function to compute the value of the variables. Values declared as dependent are initialized within the body of `evaluateDependents()`, which is run once, before any pixels are processed. The values are then read-only, and are held constant over all pixels as `evaluatePixel()` is executed.

Here is the complete `GaussianBlur` kernel, modified to use dependent variables:

```
<languageVersion : 1.0;>
kernel GaussianBlur
<
    namespace : "Tutorial";
    vendor : "Adobe";
    version : 1;
>
{
    dependent float weight0;
    dependent float weight1;
    dependent float weight2;
    dependent float weight3;
    dependent float weight4;

    input image4 source;
    output pixel4 result;

    void evaluateDependents()
    {
        const float sigma = 2.0;
        float c = 1.0 / ( sqrt( 2.0 * 3.1415926535 ) * sigma );
        float ec = 2.0 * sigma * sigma;

        weight0 = exp( -( 0.0 * 0.0 ) / ec ) * c;
        weight1 = exp( -( 1.0 * 1.0 ) / ec ) * c;
        weight2 = exp( -( 2.0 * 2.0 ) / ec ) * c;
        weight3 = exp( -( 3.0 * 3.0 ) / ec ) * c;
        weight4 = exp( -( 4.0 * 4.0 ) / ec ) * c;
    }

    void evaluatePixel()
    {
        float4 acc = float4( 0.0 );
        acc += sampleNearest( source, outCoord() ) * weight0;
        acc += sampleNearest( source, outCoord() + float2( 1.0, 0.0 ) ) * weight1;
        acc += sampleNearest( source, outCoord() + float2( -1.0, 0.0 ) ) * weight1;
        acc += sampleNearest( source, outCoord() + float2( 2.0, 0.0 ) ) * weight2;
        acc += sampleNearest( source, outCoord() + float2( -2.0, 0.0 ) ) * weight2;
        acc += sampleNearest( source, outCoord() + float2( 3.0, 0.0 ) ) * weight3;
        acc += sampleNearest( source, outCoord() + float2( -3.0, 0.0 ) ) * weight3;
```

```
        acc += sampleNearest( source, outCoord() + float2( 4.0, 0.0 ) ) * weight4;
        acc += sampleNearest( source, outCoord() + float2( -4.0, 0.0 ) ) * weight4;

        result = acc;
    }
}
```

Hints and tips

Here are some things to watch out for in designing your kernel code.

Undefined results

Some of the built-in operators and functions produce undefined results under some conditions, such as division by zero, or square root of a negative number.

An undefined result causes unpredictable behavior. The kernel might stop running altogether, it might generate black pixels, or white pixels. It might produce different results on different runs, or it might produce different results when running on CPU, GPU or Flash Player. It might appear to work in one version, then fail in the next version.

You must not allow your kernel to use operators or functions in an undefined manner. For example, if you have a kernel that does division, you must take into account the possibility that the divisor will be zero. Make sure that your code detects that condition and takes the appropriate action.

Inexact floating-point arithmetic

Floating-point calculations performed on the CPU might give slightly different results to those performed on the GPU, or in Flash Player, due to differences in optimization and rounding. This can lead to problems when comparing floating-point values for equality or inequality. You can avoid these problems by replacing code like this:

```
if( f1 == f2 )
{ // Do something }
```

with code like this:

```
if( abs( f1 - f2 ) < epsilon )
{ // Do something }
```

Out-of-range output pixel values

Pixel values are normally assumed to be in the range 0.0 to 1.0. It is possible, however, to set the value of an output pixel to be outside of this range.

- When running on the GPU, all output pixel values are clamped to the 0.0 to 1.0 range.
- When running on the CPU, output pixel values are not clamped.

It is recommended that you write kernels that work the same in all execution environments. To ensure this, if your kernel might produce out-of-range values, you should clamp them to the expected range in your own code.

Array sizes

Some older GPUs have limits on the size of arrays they can handle. Be aware that if you write a kernel using arrays, it might not run on all GPUs.

Support functions

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

- A kernel can contain definitions for *region functions*, which have predefined names and signatures, and provide the Pixel Bender run-time engine with helpful information about how much space to allocate for building an output image from the pixels that a kernel computes. For details, see Chapter 4, "Working with Regions."
- A kernel can also define additional arbitrary support functions, which can be called only from `evaluatePixel()` or `evaluateDependents()`.

Example of region functions

`RotateAndComposite` is a kernel that transforms one input and composites it on top of another. It demonstrates the use of dependents to hold a pre-computed transformation matrix, multiple inputs that are treated differently, and more complex region functions. Matrices are stored in column-major order; for details, see the *Pixel Bender Reference*.

```
<languageVersion : 1.0;>
kernel RotateAndComposite
<
    namespace : "Tutorial";
    vendor : "Adobe";
    version : 1;
>
{
    parameter float theta;       // rotation angle
    parameter float2 center      // rotation center
    <
        minValue: float2(0);
        maxValue: float2(1000);
        defaultValue: float2(200);
    >;
    dependent float3x3 back_xform; // rotation matrix (from dest->src)
    dependent float3x3 fwd_xform; // and inverse

    input image4 foreground;
    input image4 background;
    output pixel4 result;

    // Compute the transform matrix and its inverse
    void evaluateDependents()
    {
        // translate center to origin
        float3x3 translate = float3x3(
            1, 0, 0,
            0, 1, 0,
            -center.x, -center.y, 1 );

        // rotate by theta
```

```
    float3x3 rotate = float3x3(
        cos(theta), sin(theta), 0,
        -sin(theta), cos(theta), 0, 0, 0, 1 );

    // compose to get complete fwd transform
    fwd_xform = -translate*rotate*translate;

    // get inverse rotation (by negating sin terms)
    rotate[0][1] = -rotate[0][1];
    rotate[1][0] = -rotate[1][0];

    // compose to get complete back transform
    back_xform = translate*rotate*-translate;
}

// Main function transforms outCoord to find foreground pixel
void evaluatePixel()
{
    // background coordinate is just destination coordinate
    float2 bg_coord = outCoord();

    // foreground coordinate is transformed outCoord
    // note up-conversion to float3, then we drop w with a swizzle
    float2 fg_coord =
        (back_xform*float3(bg_coord.x, bg_coord.y, 1)).xy;

    // alpha blend foreground over background
    pixel4 bg_pixel = sampleNearest(background, bg_coord);
    pixel4 fg_pixel = sampleLinear(foreground, fg_coord);

    result = mix(bg_pixel, fg_pixel, fg_pixel.a);
}

// needed function is different depending on which input
region needed(
    region output_region,
    imageRef input_index )
{
    region result;

    if( input_index == background )
    {
        // background is untransformed, so just pass through
        result = output_region;
    }
    else
    {
        // transform output region into foreground space
        result = transform( back_xform, output_region );
    }
        return result;
}

// changed function is like needed function but with different
// transform sense
region changed(
    region input_region,
    imageRef input_index )
{
    region result;
```

```
        if( input_index == background )
        {
            result = input_region;
        }
        else
        {
            result = transform( fwd_xform, input_region );
        }
        return result;
    }
}
```

Example of a support function

You can define additional support functions within a kernel to reduce repetition in your code. In this example the code to calculate the Gaussian weight in the Gaussian blur filter is moved into a support function:

```
<languageVersion : 1.0;>
kernel GaussianBlur
<
    namespace : "Tutorial";
    vendor : "Adobe";
    version : 1;
>
{
    dependent float weight0;
    dependent float weight1;
    dependent float weight2;
    dependent float weight3;
    dependent float weight4;

    input image4 source;
    output pixel4 result;

    float calculateWeight( float r, float sigma )
    {
        float c = 1.0 / ( sqrt( 2.0 * 3.1415926535 ) * sigma );
        float ec = 2.0 * sigma * sigma;

        return exp( -( r * r ) / ec ) * c;
    }

    void evaluateDependents()
    {
        const float sigma = 2.0;

        weight0 = calculateWeight( 0.0, sigma );
        weight1 = calculateWeight( 1.0, sigma );
        weight2 = calculateWeight( 2.0, sigma );
        weight3 = calculateWeight( 3.0, sigma );
        weight4 = calculateWeight( 4.0, sigma );
    }

    void evaluatePixel()
    {
        float4 acc = float4( 0.0 );
        acc += sampleNearest( source, outCoord() ) * weight0;
```

```
        acc += sampleNearest( source, outCoord() + float2( 1.0, 0.0 ) ) * weight1;
        acc += sampleNearest( source, outCoord() + float2( -1.0, 0.0 ) ) * weight1;
        acc += sampleNearest( source, outCoord() + float2( 2.0, 0.0 ) ) * weight2;
        acc += sampleNearest( source, outCoord() + float2( -2.0, 0.0 ) ) * weight2;
        acc += sampleNearest( source, outCoord() + float2( 3.0, 0.0 ) ) * weight3;
        acc += sampleNearest( source, outCoord() + float2( -3.0, 0.0 ) ) * weight3;
        acc += sampleNearest( source, outCoord() + float2( 4.0, 0.0 ) ) * weight4;
        acc += sampleNearest( source, outCoord() + float2( -4.0, 0.0 ) ) * weight4;

        result = acc;
    }
}
```

4 Working with Regions

The Pixel Bender run-time engine executes a kernel once for each output pixel in order to produce an output image. For a graph, where several kernels are connected together, each output pixel from one kernel becomes the input for the next. To make this process efficient, it helps if the Pixel Bender run-time engine can exclude a pixel from the calculation if that pixel is not going to affect the final display.

Previous examples (such as the `GaussianBlur` kernel) have shown how a kernel can access other pixels besides the current pixel. The value of one single pixel in the output image can depend on multiple pixels in the input image—that is, a single pixel in the output image depends on a *region* of pixels in the input image. To calculate the output image as efficiently as possible, the Pixel Bender run-time engine needs to know exactly which pixels are required from the input image.

Similarly, a single pixel in the input image can affect multiple pixels in the output image. If a pixel changes in the input image, multiple output pixels might have to be recalculated.

If your kernel accesses any pixels other than the current pixel, you must implement the *region functions* within the kernel, in order to supply the Pixel Bender run-time engine with specific information about what regions are involved in the calculations.

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

How Pixel Bender runs a kernel

When the user invokes a filter defined in a Pixel Bender kernel program, the Pixel Bender run-time engine follows this procedure:

1. Evaluate dependent variables;
2. Determine input and output regions, using the results of region functions if they are provided;
3. Execute the `evaluatePixel()` function for each pixel of the output region to produce the output image.

By determining the regions first, the engine eliminates all unnecessary pixel calculations, and greatly improves the efficiency of its image processing. The engine uses region information that is both implicit and explicit in the kernel definition:

- The *domain of definition* (DOD): The region of the input that contains all non-trivial pixels. You can think of this as “What have I got to start with?”.

  Every image has some region over which it is defined, the DOD. For instance, if you read a PNG file from the hard drive which is 200 pixels wide by 100 pixels high, the domain of definition is (0, 0, 200, 100).

- The *region of interest* (ROI): The region of pixels on the output that need to be calculated. You can think of this as “What do I want to end up with?”.

  A kernel can provide information about the ROI by supplying region function definitions (see “Defining regions” on page 44) to answer these questions:

  - I want this region of the output, what region of the input is needed to calculate it?

▷ I have changed this region of the input, what region of the output will change because of this?

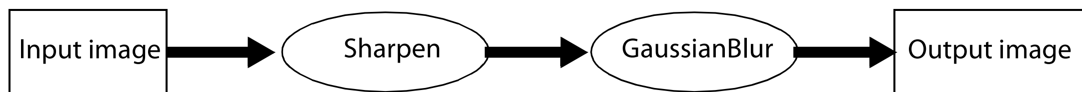
▷ What output region do I need to generate, even if all of the input images are of zero size?

Region reasoning in graphs

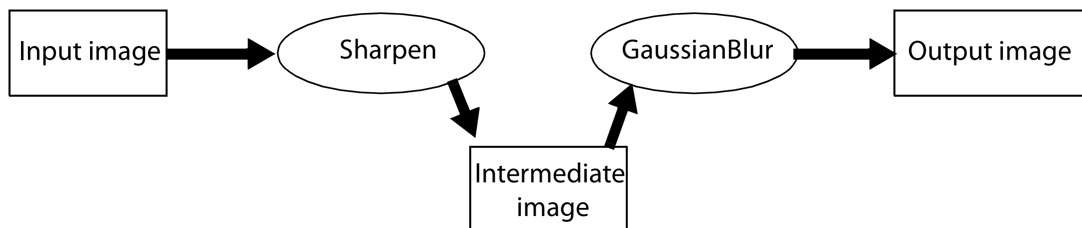
Region definition is particularly important in graphs, where the input image to one kernel is the output image of another kernel.

Graph definitions do not have region functions that apply to the graph as a whole, but the embedded kernels generally should include region functions. Before running any of the kernel operations, the Pixel Bender run-time engine determines the required output region from the final node, then works backward through the chain of nodes to determine each needed region, making that the output region of the upstream node. This ensures that the engine does not have to calculate any pixels that are not used in the final output.

For example, suppose that the `GaussianBlur` kernel is the second in a short chain of two kernels:



As images flow through the graph, each intermediate image has a DOD, as does the final output image.



The final output of the graph is the initial ROI, which the engine then tracks backward through the graph to determine the ROI for each intermediate node. In this case, the Pixel Bender run-time engine calculates the region that the output from the `Sharpen` kernel must cover, in order to correctly calculate the final output from the `GaussianBlur` kernel.

Graphs are discussed in detail in Chapter 5, "Pixel Bender Graph Language."

FURTHER READING: For a more rigorous discussion of region computation, see Michael A. Shantzis, *A model for efficient and flexible image computing* (http://portal.acm.org/citation.cfm?id=192191).

Defining regions

A kernel can contain these function definitions, each of which returns an opaque `region` data type. The functions tell the Pixel Bender run-time engine how much space to allocate for the input and output images of the kernel operation:

- The `needed()` function specifies the region of the input image within which pixels need to participate in the computation, in order to produce the desired output region. This function uses the ROI of the output image to determine the ROI of the input image or images.

- The `changed()` function specifies the region of the output image within which pixels must be recomputed when input pixels change. This is typically either the same as or the inverse of the `needed()` function. This function uses the DOD of the input image or images to determine the DOD of the output image.
- The `generated()` function specifies the region of the output where non-zero pixels will be produced even if all image inputs are completely empty. This determines the DOD of an output image that is entirely generated by the kernel.

When running a kernel, the Pixel Bender run-time engine calls the `needed()` and `changed()` functions once for each input image, before making any calls to `evaluatePixel()`.

Region functions are optional. If none are supplied, the Pixel Bender run-time engine assumes the kernel is *pointwise*, meaning it accesses only one input pixel from each input, directly under the current output pixel. For a pointwise kernel, the inputs need only be exactly as large as the desired output. (The desired output size is a subset of the infinite image plane, determined by the application, not the kernel writer. It may correspond to the entire domain of definition—the "whole" image—or a single tile or other sub-region.)

Even for a pointwise operation, however, region functions can increase the efficiency by excluding pixels that would, for example, be masked out during the operation. An example of this usage is the `MatteRGBA` kernel (page 49).

Many useful image-processing operations are not pointwise, such as the `HorizontalAverage` kernel (page 47), which accesses one pixel to the left and one pixel to the right of the current output pixel. Non-pointwise kernels are those which perform convolution or gathering operations, transforming data from multiple pixels in each input image to derive each output pixel. For these kernels, correct results and efficient operation depend on correctly identifying the regions that are required for and affected by the operation.

After Effects note: After Effects requires that you provide the `needed()` and `changed()` functions for all non-pointwise operations. If you do not, the operation is inefficient, and may produce in incorrect images.

Adjusting image size to allow for edge calculations

For non-pointwise operations, surrounding pixel values affect the calculation of each output pixel. For example, this simple blur kernel averages each pixel with its left and right neighbors:

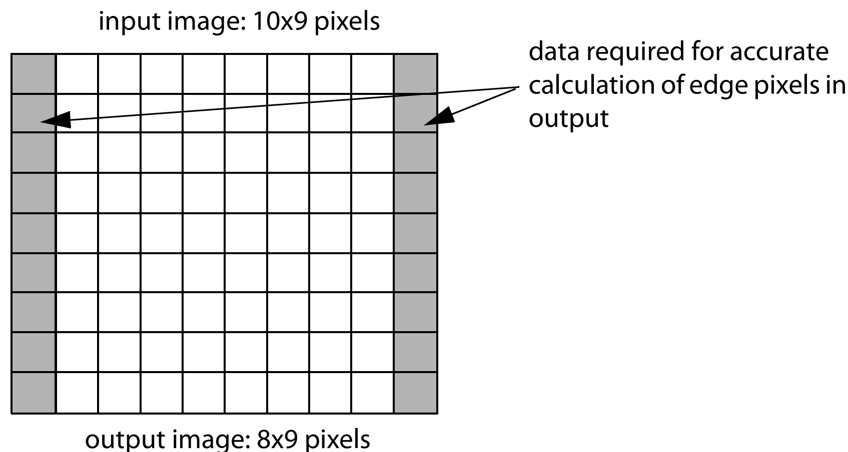
```
kernel HorizontalAverage
<...>
{
    input image4 source;
    output pixel4 result;

    void evaluatePixel()
    {
        float2 coord = outCoord();
        float2 hOffset = float2(pixelSize(source).x, 0.0);
        pixel4 left = sampleNearest(source, coord - hOffset);
        pixel4 center= sampleNearest(source, coord);
        pixel4 right = sampleNearest(source, coord + hOffset);
        result = (left + center + right) / 3.0;
    }
}
```

In order for this kernel to produce accurate results, it must obtain the left and right neighbors of each pixel. At the right edge of the image, however, no pixel data is defined for the right-hand neighbor, and at the left edge, there is no data for left neighbor. That is, the DOD of the input image does not supply all of the data needed for the ROI.

Pixel Bender defines the sampling functions to use a transparent black color for any pixels that are not in the defined area of an image, as though the image has an infinitely wide transparent black border. For this averaging filter, attempting to calculate the new value of a pixel on the left or right edge of the image makes that pixel more transparent, because it is being averaged with transparent black on one side.

If you maintain the size of the image, you must either accept increased transparency in the output on the edges, or introduce special processing for the edge cases. If you do not want the increased transparency, but also do not want to do special processing at the edge of the image, you must reduce the size of the output image with respect to that of the input image.

For example, if the input image is 10 pixels wide by 9 pixels high, the maximum size of an output image with correct transparency would be 8 pixels wide by 9 pixels high. This allows for accurate sampling of data on the left and right edges.



Alternatively, if you know you need an output image that is 8x9 pixels, you know that you will need at least a 10x9 input region in order to get accurate sampling. This is the *needed* region, which you can specify using the `needed()` function (as shown in the example on page 47).

If the input image happens to be smaller than the needed region, you do not actually get any more data; the sampling function uses the default of transparent black, and you still get the increased transparency effect. However, if the input image is coming from an upstream kernel in a graph, specifying the needed region in the downstream kernel allows the Pixel Bender run-time engine to ensure that the upstream kernel produces a sufficiently large output image.

Conservative region bounds

The region functions are not expected to return exact, pixel-perfect bounds for all cases. Often, exact bounds are difficult or impossible to compute. Rather, the returned regions must be conservative bounds: they must completely contain the ideal minimal set of pixels. The effect of making the bounds too large is potentially wasted storage and computation. The operation can be very inefficient if the bounds are grossly exaggerated, but the effect of making the bounds too small is worse: incorrect computation.

- If the `needed()` region is too small, needed pixels are not supplied, and portions of the input image or images are incorrectly represented to the kernel as transparent black.

- If the `changed ()` region is too small, the DOD may not include all image data (resulting in clipped images), and bad caches may be incorrectly held as valid.
- If the `generated()` region is too small, the DOD may not include all image data, resulting in clipped images.

There are cases where the true bounds of a result depend on image data; even here, bounds must be conservative. The classic example of this is a displacement-map effect, which transforms an image based on the pixel values of an input map. To express this effect in Pixel Bender, a maximum possible displacement limit must be set—say, according to a user-controllable parameter—and this limit used as a conservative radius in bounds computation.

Computing the needed region

You can supply a `needed()` function to determine the ROI for an input image. This tells the Pixel Bender run-time engine how many input pixels are required to compute a given output region. Specifically, the `needed()` function determines the set of pixels from each input that must be accessed to compute all pixels of the output region. The Pixel Bender run-time engine then ensures that all these pixels are evaluated and available for access through the corresponding image object.

The `needed()` function always returns a `region` and takes two arguments:

- `outputRegion` — The output region (also known as the *desired* or *requested* region) is the portion of the output image's infinite plane that is to be computed; that is, the portion of the plane for which pixel values will be evaluated by calling the kernel function. You can think of the output region as the size of the output image that will be computed.
- `inputIndex` — A kernel can have an arbitrary number of inputs, each of which can access a different needed region for computation of the same output region. For example, a kernel can transform each input differently. For this reason, the `needed()` function receives a reference to the specific input for which the ROI is to be computed. For a kernel that has only one input, the reference always refers to that image, and can be ignored.

Regions cannot take part in arithmetic expressions in the usual way, but the Pixel Bender kernel language provides a suite of custom functions for dealing with them. For example, the following function definition takes the output region and expands it by one pixel on all sides to find an input region that contains adequate data for sampling the edges. It uses one of the built-in region-manipulation functions, `outset()`, to perform the region calculation. It also takes into account the pixel size of the image (in case it has non-square pixels).

This example ignores the input argument, because the kernel has only one input image. For an example that uses the `inputIndex` value to distinguish among multiple input images, see "A more complex example" on page 50.

```
kernel HorizontalAverage
<...>
{
    input image4 source;
    output pixel4 result;

    region needed(region outputRegion, imageRef inputIndex) {
        region result = outputRegion;
        result = outset( result, float2(pixelSize(source).x, 0) );
        return result;
    }
```

```
    void evaluatePixel()
    {
        float2 coord = outCoord();
        float2 hOffset = float2(pixelSize(source).x, 0.0);

        pixel4 left  = sampleNearest(source, coord - hOffset);
        pixel4 center= sampleNearest(source, coord);
        pixel4 right = sampleNearest(source, coord + hOffset);

        result = (left + center + right) / 3.0;
    }
}
```

Accessing kernel parameters in the needed() function

The region functions can access kernel parameters and dependents simply by referring to them, just as `evaluateDependents()` and `evaluatePixels()` can do. This is useful where the ROI varies based on a kernel parameter, such as a variable-size convolution:

```
kernel VariableHorizontalAverage
<...>
{
    parameter int radius; // Measured in pixels

    input image4 source;
    output pixel4 result;

    region needed(region outputRegion, imageRef inputIndex)
    {
        region result = outputRegion;
        result = outset( result, float2( float( radius ) * pixelSize( source ).x, 0 ) );
        return result;
    }

    void evaluatePixel()
    {
        result = pixel4(0,0,0,0);
        float2 hOffset = float2(pixelSize(source).x, 0.0);

        for (int i=-radius; i<=radius; i++)
            result += sampleNearest( source, outCoord()+ float(i) * hOffset );

        result /= float( radius*2 + 1 );
    }
}
```

Using the DOD to calculate the needed region

The domains of definition of each input also are available during region computation. A `needed()` function can return any region, but only pixels inside the DOD are actually computed; that is, the ROI is always clipped to the DOD.

Knowledge of the DOD can lead to certain optimizations. For example, if a kernel mattes one source against a mask, only that part of the source inside the mask DOD is needed, and the reverse. You can use this information in a `needed()` function to optimize the `MatteRGBA` kernel. The `needed()` function is not required for correct operation of this kernel, as `MatteRGBA` is a pointwise operation (that is, it accesses only

the current input pixel for each input). Adding the `needed()` function, however, increases the efficiency by allowing the run-time system to eliminate extra pixels from each input that will simply be masked away.

```
kernel MatteRGBA
<...>
{
    input image4 source;
    input image1 matte;
    output pixel4 result;

    region needed(region outputRegion, imageRef inputIndex) {
        region result = outputRegion;
        // clip to source
        result = intersect( result, dod( source ) );
        // and to mask
        result = intersect( result, dod( matte ) );
        return result;
    }

    void evaluatePixel() {
        pixel4 in_pixel = sampleNearest( source, outCoord() );
        pixel1 matte_value = sampleNearest( matte, outCoord() );

        result = in_pixel * matte_value;
    }
}
```

The built-in function `dod()` takes an `image` or an `imageRef`, and returns the domain of definition of that image in the global coordinate system.

Computing the changed region

The `changed()` function can be considered the inverse of the `needed()` function. This function computes the set of pixels that may be affected if any pixel in the specified input region changes. This is of crucial importance for cache invalidation, for example. This function also is used to compute an image's DOD.

The body of the `changed()` function is often similar to or even identical to that of the `needed()` function, as in this definition for `VariableHorizontalAverage`:

```
region changed( region inputRegion, imageRef inputIndex)
{
    region result = inputRegion;
    result = outset( result, float2( float( radius ) * pixelSize( source ).x, 0 ) );
    return result;
}
```

This is not always the case, though. Generally, asymmetric kernels require a reflection between `needed()` and `changed()`. Kernels that perform a geometric transformation of the image need to invert the sense of the transform between these two functions.

If a `changed()` function is not supplied, changes are assumed to be propagated in pointwise fashion. This is almost always an error if a `needed()` function must be supplied for correct operation. There are exceptions, such as the `MatteRGBA` example, but generally `needed()` and `changed()` should be supplied as a pair.

A more complex example

`RotateAndComposite` is a kernel that transforms one input and composites it on top of another. It demonstrates multiple inputs that are treated differently, and more complex region functions, including a `changed()` function. It also shows the use of dependents to hold a pre-computed transformation matrix. Matrices are stored in column-major order; for details, see the *Pixel Bender Reference*.

```
<languageVersion : 1.0;>
kernel RotateAndComposite
<
    namespace : "Tutorial";
    vendor : "Adobe";
    version : 1;
>
{
    parameter float theta; // rotation angle
    parameter float2 center // rotation center
    <
        minValue: float2(0);
        maxValue: float2(1000);
        defaultValue: float2(200);
    >;
    dependent float3x3 back_xform; // rotation matrix (from dest->src)
    dependent float3x3 fwd_xform; // and inverse

    input image4 foreground;
    input image4 background;
    output pixel4 result;

    // Compute the transform matrix and its inverse
    void evaluateDependents()
    {
        // translate center to origin
        float3x3 translate = float3x3(
            1, 0, 0,
            0, 1, 0,
            -center.x, -center.y, 1 );

        // rotate by theta
        float3x3 rotate = float3x3(
            cos(theta), sin(theta), 0,
            -sin(theta), cos(theta), 0, 0, 0, 1 );

        // compose to get complete fwd transform
        fwd_xform = -translate*rotate*translate;

        // get inverse rotation (by negating sin terms)
        rotate[0][1] = -rotate[0][1];
        rotate[1][0] = -rotate[1][0];

        // compose to get complete back transform
        back_xform = translate*rotate*-translate;
    }

    // needed function is different depending on which input
    region needed(region outputRegion, imageRef inputIndex )
    {
        region result;
```

```
        if( inputIndex == background ) {
            // background is untransformed, so just pass through
            result = outputRegion;
        }
        else {
            // transform output region into foreground space
            result = transform( back_xform, outputRegion );
        }
            return result;
    }

    // changed function is like needed function but with different
    // transform sense
    region changed(region inputRegion, imageRef inputIndex )
    {
        region result;

        if( inputIndex == background ) {
            result = inputRegion;
        }
        else {
            result = transform( fwd_xform, inputRegion );
        }
        return result;
    }

    // Main function transforms outCoord to find foreground pixel
    void evaluatePixel()
    {
        // background coordinate is just destination coordinate
        float2 bg_coord = outCoord();

        // foreground coordinate is transformed outCoord
        // note up-conversion to float3, then we drop w with a swizzle
        float2 fg_coord =
            (back_xform*float3(bg_coord.x, bg_coord.y, 1)).xy;

        // alpha blend foreground over background
        pixel4 bg_pixel = sampleNearest(background, bg_coord);
        pixel4 fg_pixel = sampleLinear(foreground, fg_coord);

        result = mix(bg_pixel, fg_pixel, fg_pixel.a);
    }
}
```

Computing the generated region

The `generated()` function describes the region of the output where non-zero pixels will be produced even if all image inputs are completely empty. It is required for correct operation whenever the kernel renders something to the output, as in this `RenderFilledCircle` generator kernel:

```
kernel RenderFilledCircle
<...>
{
    parameter float radius;
    parameter pixel4 color;

    output pixel4 result;
```

```
    region generated()
    {
        float r = ceil(radius);
        return region(float4(-r, -r, r, r));
    }

    void evaluatePixel()
    {
        float2 coord_for_this_pixel = outCoord();
        float cur_radius = length(coord_for_this_pixel);
        if (cur_radius < radius)
            result = color;
        else
            result = pixel4(0,0,0,0);
    }
}
```

A `generated()` function has no arguments, but usually needs to access some kernel parameters to compute its result. In this case, it uses the `radius` parameter to determine a bounding rectangle for the output, which is then used to initialize a `region` object in `(`*`leftX`*`, `*`topY`*`, `*`rightX`*`, `*`bottomY`*`)` form.

Although this example takes no input images, a `generated()` function may also be required for kernels that process one or more inputs. Usually, this occurs when the kernel composites a procedurally rendered object over an input; for example, a paint stroke or lens flare.

The `generated()` function must return a region that contains the rendered object; otherwise, the Pixel Bender run-time engine does not know to expand the output buffer to include the entire object where it extends outside the borders of the input image.

A kernel that theoretically produces an output over the entire infinite plane, like some types of procedural texture generation, should return the infinite region, using the built-in function `everywhere()`.

AFTER EFFECTS NOTE: After Effects does not support the infinite region generator, `everywhere()`. 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.

5 Pixel Bender Graph Language

This chapter introduces the Pixel Bender graph language, which allows you to create more sophisticated image processing effects by connecting multiple Pixel Bender kernels into a *processing graph*, that can be treated as a single filter.

When executing a graph, the Pixel Bender run-time engine passes an input image to the first kernel in a chain. It runs that kernel to completion, computing all of the pixels for a complete output image, which it passes as the input image to the next kernel. It continues through the chain of kernels until it produces the desired output image. For efficiency, the Pixel Bender run-time engine first determines all necessary input and output regions, including those for the intermediate images, so that it does not have to process any pixels that are not actually required for the final output image.

Pixel Bender supplies a Graph Editor, which you can use to create and edit graphs; see "Using the Graph Editor" on page 65.

FLASH NOTE: Graphs are not available in Flash Player.

Graph elements

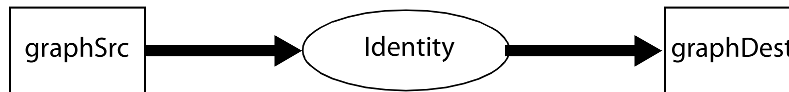
The Pixel Bender graph language is an XML-based language that describes the structure of a graph. It allows you to declare a set of *nodes*, specify the *connections* between those nodes, and supply parameters.

The top-level container, the `graph` element, contains these sections:

Graph header	The `graph` element attributes supply basic header information, including the graph's `name`. An application typically uses this value as the title of the resulting filter.
Graph metadata	A set of `metadata` elements supplies the namespace and graph version information.
Graph parameters	Optionally, a set of `parameter` elements supplies named values to be entered by the user, with optional constraints.
Image inputs and outputs	The `input` and `output` elements specify the input and output images.
Embedded kernels	One or more complete `kernel` definitions, written in Pixel Bender.
Nodes	One or more `node` elements, which specify a unique *instance*, or application, of one of the embedded kernels.
Connections	A set of `connect` elements, which specify the sequence of nodes between input and output.

Simple graph example

The simplest possible graph contains a single identity node and connects an input image to an output image like this:



Graph header

The `graph` element is the top-level container. The attributes supply the name of the graph, the version of the Pixel Bender graph language that is in use, and the XML namespace.

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

Graph metadata

```
    <metadata name = "namespace" value = "GraphTest" />
    <metadata name = "vendor" value = "Adobe" />
    <metadata name = "version" type = "int" value = "1" />
```

All Pixel Bender graphs and kernels are identified by a unique combination of name, namespace, vendor, and version. The name of the graph has already been supplied in the `graph` element; the `metadata` elements supply the namespace, vendor and version. All of these identifying items must be supplied; the graph does not compile unless they are present. Note that the `version` metadata must be identified as type `int`.

Each kernel within the graph has its own namespace, separate from the graph namespace; see "Embedded kernels" on page 55.

Graph parameters

This simple graph contains no parameters so this section is empty. See "Defining graph and kernel parameters" on page 58 for an example of graph parameters.

Image inputs and outputs

```
    <inputImage type = "image4" name = "graphSrc" />
    <outputImage type = "image4" name = "graphDst" />
```

This graph accepts a single 4-channel input image called "graphSrc", and produces a single 4-channel output image called "graphDst".

A graph produces an image, unlike a kernel, which produces a single pixel.

- A kernel defines its output as a single pixel. The Pixel Bender run-time engine produces a complete image by running the kernel for every pixel in the desired output region.

- The graph defines its output as an image, which is produced by running the embedded kernels. A graph must have an output image.

Embedded kernels

An embedded kernel is a fully-specified kernel, created in Pixel Bender kernel language syntax, embedded within the XML code for a Pixel Bender graph.

```
<kernel>
    <![CDATA[
        <languageVersion : 1.0;>
        kernel Identity
        <   namespace: "Tutorial";
            vendor: "Adobe";
            version: 1;
        >
        {
            input image4 src;
            output pixel4 dst;
            void evaluatePixel() {
                dst = sampleNearest(src, outCoord());
            }
        }
    ]]>
</kernel>
```

This graph contains a single embedded kernel, written in the Pixel Bender kernel language.

- The kernel, like the graph, has its own name, namespace, vendor and version metadata, which is used to uniquely identify the kernel.
- The kernel parameters, dependent variables, input image, and output pixel are all defined within the kernel namespace. Here, the `namespace` value is the same as the one used in the graph metadata, but the namespace is still unique to this kernel because it is combined with the other identifying features of the kernel.

Nodes

A node defines an *instance* of a particular kernel, which is an application of that kernel's processing in the sequence defined by the graph connections.

```
<node id = "identity"
    name="Identity" vendor="Adobe"
    namespace="Tutorial" version="1" />
```

This graph has a single node, with the unique node identifier `identity`. The kernel that it invokes is identified by its name and metadata attributes. In this case, it is the `Identity` kernel that was defined as an embedded kernel.

Typically, a node ID is assigned in a way that associates it with the kernel it invokes, while remaining unique. For instance, you can use case to distinguish the node ID from the kernel name, as in this example: the `identity` node calls the `Identity` kernel. For examples, see "Defining multiple nodes" on page 59.

Connections

Finally, the inputs, outputs and nodes of the graph are connected to complete the graph.

```
<connect fromImage = "graphSrc" toNode = "identity" toInput = "src" />
<connect fromNode ="identity" fromOutput = "dst" toImage = "graphDst" />
```

This graph has only two connections:

- The input image of the graph (`graphSrc`) is connected to the input image of the `myNode` node (`src`).
- The output pixel of the `identity` node (`dst`) is connected to the output image of the graph (`graphDst`).

The names assigned to these image variables are in different namespaces (the graph namespace and the kernel namespace), so they could be the same. They are given different names here to help distinguish the context:

- The graph's input and output images are defined by the graph's `inputImage` and `outputImage` statements:

```
<!-- Image inputs and outputs -->
<inputImage type = "image4" name = "graphSrc" />
<outputImage type = "image4" name = "graphDst" />
```

- The node's input image and output pixel are part of the kernel definition:

```
<kernel>
    <![CDATA[
        <languageVersion : 1.0;>
        kernel Identity
        < ... >
        {
            input image4 src;
            output pixel4 dst;
            void evaluatePixel()
            { ... }
        }
    ]]>
</kernel>
```

Complete simple example

Here is the complete program that defines the `Simplest` graph:

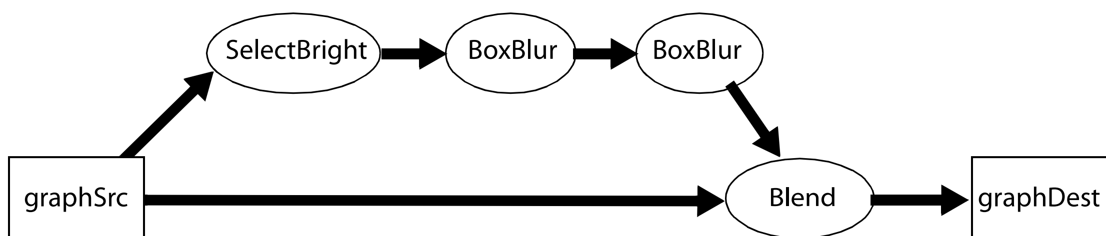
```
<?xml version="1.0" encoding="utf-8"?>
<graph name = "Simplest"
    languageVersion = "1.0"
    xmlns="http://ns.adobe.com/PixelBenderGraph/1.0">

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

    <!-- Graph parameters (none in this example)-->

    <!-- Image inputs and outputs -->
```

```
    <inputImage type = "image4" name = "graphSrc" />
    <outputImage type = "image4" name = "graphDst" />

    <!-- Embedded kernels -->
    <kernel>
        <![CDATA[
            <languageVersion : 1.0;>
            kernel Identity
            <   namespace: "Tutorial";
                vendor: "Adobe";
                version: 1;
            >
            {
                input image4 src;
                output pixel4 dst;
                void evaluatePixel()
                {
                    dst = sampleNearest(src, outCoord());
                }
            }
        ]]>
    </kernel>

    <!-- Nodes -->
    <node id = "identity" name="Identity" vendor="Adobe"
        namespace="Tutorial" version="1" />

    <!-- Connections -->
    <connect fromImage = "graphSrc" toNode = "identity" toInput = "src" />
    <connect fromNode ="identity" fromOutput = "dst" toImage = "graphDst" />

</graph>
```

A complex graph

The power of the graph language derives from the ability to connect several kernels together. A particular kernel operation can be executed multiple times; each call gets different parameter settings, and is represented by a unique node in the graph.

We can put together a simple glow filter by combining several kernels like this:



This graph demonstrates several interesting points:

- The `graphSrc` input image (declared as an input to the graph) is processed by a chain of filters. This illustrates how you pass the result of one operation into another operation.
- The result of the chain of filters is blended with the unchanged version of the `graphSrc` input image. This illustrates how you pass multiple input images into a node.

- The chain of processing applies the same filter (`BoxBlur`) twice, using different input parameters (one is a horizontal blur, the other is a vertical blur). This illustrates how a node is a specific instance of a filter operation.
- The graph itself takes parameters, and several of the internal nodes take parameters, which illustrates how you declare, pass, set, and access parameters in different contexts.

Defining the graph

The complete XML description of this graph is given in "Complete complex example" on page 61. It contains all of the parts that were in the simple example:

- The graph header and metadata, which name this graph "SimpleGlow".
- A single input image named `graphSrc` and output image named `graphDst`.

In addition, this graph defines:

- A graph parameter, which the user sets when the graph is invoked.
- Embedded kernel definitions for the three kernels used by the graph, `SelectBright`, `BoxBlur`, and `Blend`. These kernels take parameters.
- Multiple nodes that set the parameters for the kernels they invoke, using the graph parameter.
- Connections between nodes, and connections that define multiple inputs to a node.

Let's take a closer look at the sections of the graph that are different from those in the simple example.

Defining graph and kernel parameters

This graph has a single floating-point parameter called `amount`:

```
<parameter type = "float" name = "amount" >
    <metadata name = "defaultValue" type = "float" value = "5.0"/>
    <metadata name = "maxValue" type = "float" value = "10.0" />
    <metadata name = "minValue" type = "float" value = "0.0" />
</parameter>
```

As with parameters to a Pixel Bender kernel, the graph parameter can have metadata that describes the constraints. In this case, there is a minimum, maximum and default value.

The kernels also take parameters.

- `SelectBright` takes one parameter, the brightness `threshold`:

```
kernel SelectBright
<  ... >
{
    ...
    parameter float threshold;
    void evaluatePixel() {
    ... code uses threshold...
        }
}
```

- `BoxBlur` takes two parameters, `direction` and `radius`:

```
kernel BoxBlur
<  ... >
{
    ...
    parameter float2 direction;
    parameter int radius;
    void evaluatePixel() {
    ... code uses direction and radius...
        }
}
```

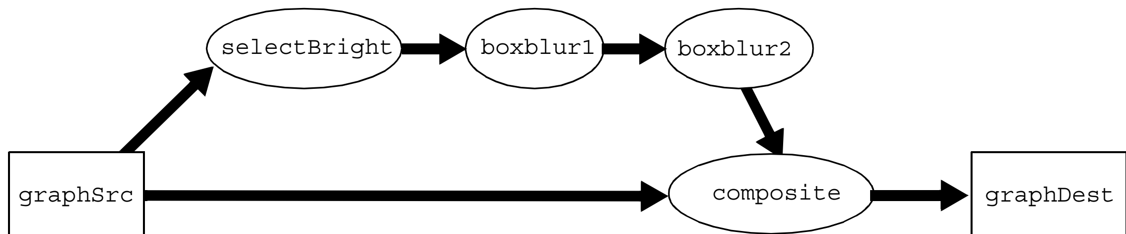
The graph parameter is available to any node in the graph. The nodes use the graph parameter to set the kernel parameters, as shown in "Setting kernel parameters" on page 59.

Defining multiple nodes

This graph has four nodes; two of the nodes are instances of the same kernel. Each node has a unique node ID, which is different from the name of the kernel it invokes. This example illustrates a useful naming convention for nodes and kernels, to help distinguish them.

- The `selectBright` node invokes the `SelectBright` kernel
- The `boxblur1` and `boxblur2` nodes both invoke the `BoxBlur` kernel; each instance sets the kernel parameter values differently.

The node that invokes the `Blend` kernel is named using a different convention. The node name `composite` reflects the particular use of blending in this context, to compose a single image from two input images. You might use this convention if a kernel operation could be applied for different purposes in a graph.

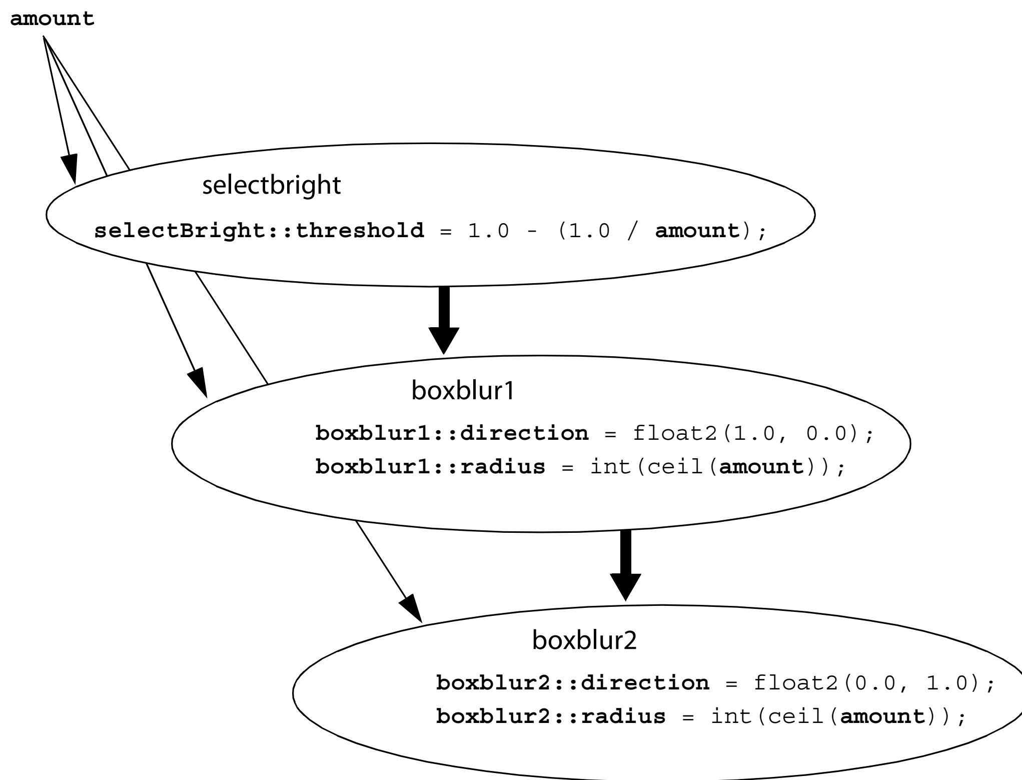


Setting kernel parameters

For those kernels that require parameters, the node must set the values of those parameters. A node *must* set *all* of the parameters for the kernel it invokes, even if they have default values.

The node definition does this by including an `evaluateParameters` element, which defines an `evaluateParameters()` function. (If the kernel has no parameters, simply omit the `evaluateParameters` element.)

The `evaluateParameters()` function has direct access to the graph parameters, and can reference the kernel parameters using the syntax *`nodeName::parameterName`*.



For example, the `selectBright` node definition looks like this:

```
<node
    id = "selectBright"
    name="SelectBright" vendor="Adobe"
    namespace="Tutorial" version="1"  >
    <evaluateParameters>
        <![CDATA[
            void evaluateParameters() {
                selectBright::threshold = 1.0 - (1.0 / amount);
            }
        ]]>
    </evaluateParameters>
</node>
```

The two `BoxBlur` nodes specify different direction parameters. The first one performs a horizontal blur, the second is vertical.

Defining complex connections

This graph has six connections:

- The first connection is from the graph's input image to the single input image of the selectBright node:

```
<connect fromImage="graphSrc" toNode="selectBright" toInput="src"/>
```

- The next two connections pass each node's output image to the input image of the next node:

```
<connect fromNode="selectBright" fromOutput="dst"
   toNode="boxblur1" toInput="src"/>
<connect fromNode="boxblur1" fromOutput="dst"
   toNode = "boxblur2" toInput="src"/>
```

- The `composite` node has two input images, `orig` and `glow`. One is supplied by the graph's input image, and the other is the result of the final blur operation:

```
<connect fromImage="graphSrc"
   toNode="composite" toInput="orig" />
<connect fromNode="boxblur2" fromOutput="dst"
   toNode="composite" toInput="glow"/>
```

- Finally, the result of the final compositing operation is passed to the graph's output image:

```
<connect fromNode="composite" fromOutput="dst" toImage="graphDst"/>
```

This shows how you can connect a single value to multiple inputs; the graph's input image is connected to both the `selectBright` node and the `composite` node.

Complete complex example

Here is the complete program that defines the `SimpleGlow` graph:

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

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

   <!-- Graph parameters -->
   <parameter type="float" name="amount" >
      <metadata name="defaultValue" type="float" value="5.0"/>
      <metadata name="maxValue" type="float" value="10.0" />
      <metadata name="minValue" type="float" value="0.0" />
   </parameter>

   <!-- Image inputs and outputs -->
   <inputImage type="image4" name="graphSrc" />
   <outputImage type="image4" name="graphDst" />

   <!-- Embedded kernels -->
   <kernel>
      <![CDATA[
         <languageVersion : 1.0;>
         kernel SelectBright
         <
            namespace: "Tutorial";
            vendor: "Adobe";
            version: 1;
         >
         {
            input image4 src;
```

```
            output float4 dst;
            parameter float threshold;
             void evaluatePixel() {
                float4 sampledColor=sampleNearest(src, outCoord());
                sampledColor.a *= step(threshold,
                    dot(sampledColor.rgb, float3(0.3, 0.59, 0.11)));
                dst=sampledColor;
            }
        }
    ]]>
</kernel>

<kernel>
    <![CDATA[
        <languageVersion : 1.0;>
        kernel BoxBlur
        <
            namespace: "Tutorial";
            vendor: "Adobe";
            version: 1;
        >
        {
            input image4 src;
            output float4 dst;

            parameter float2 direction;
            parameter int radius;

            void evaluatePixel() {
                float denominator=0.0;
                float4 colorAccumulator=float4(0.0, 0.0, 0.0, 0.0);

                float2 singlePixel=pixelSize(src);

                colorAccumulator +=sampleNearest(src, outCoord());

                for(int i=0; i <=radius; ++i) {
                    colorAccumulator +=sampleNearest(src,
                        outCoord()-(direction * singlePixel * float(i)));

                    colorAccumulator +=sampleNearest(src,
                        outCoord()+(direction * singlePixel * float(i)));
                }
                dst=colorAccumulator / (2.0 * float(radius) + 1.0);
            }
            region needed( region output_region, imageRef input_index ) {
                region result = output_region;
                result = outset( result,
                        ( pixelSize(src) * direction ) * float( radius ) );
                return result;
            }

            region changed( region input_region, imageRef input_index ) {
                region result = input_region;
                result = outset( result,
                        ( pixelSize(src) * direction ) * float( radius ) );
                return result;
            }
        }
    ]]>
```

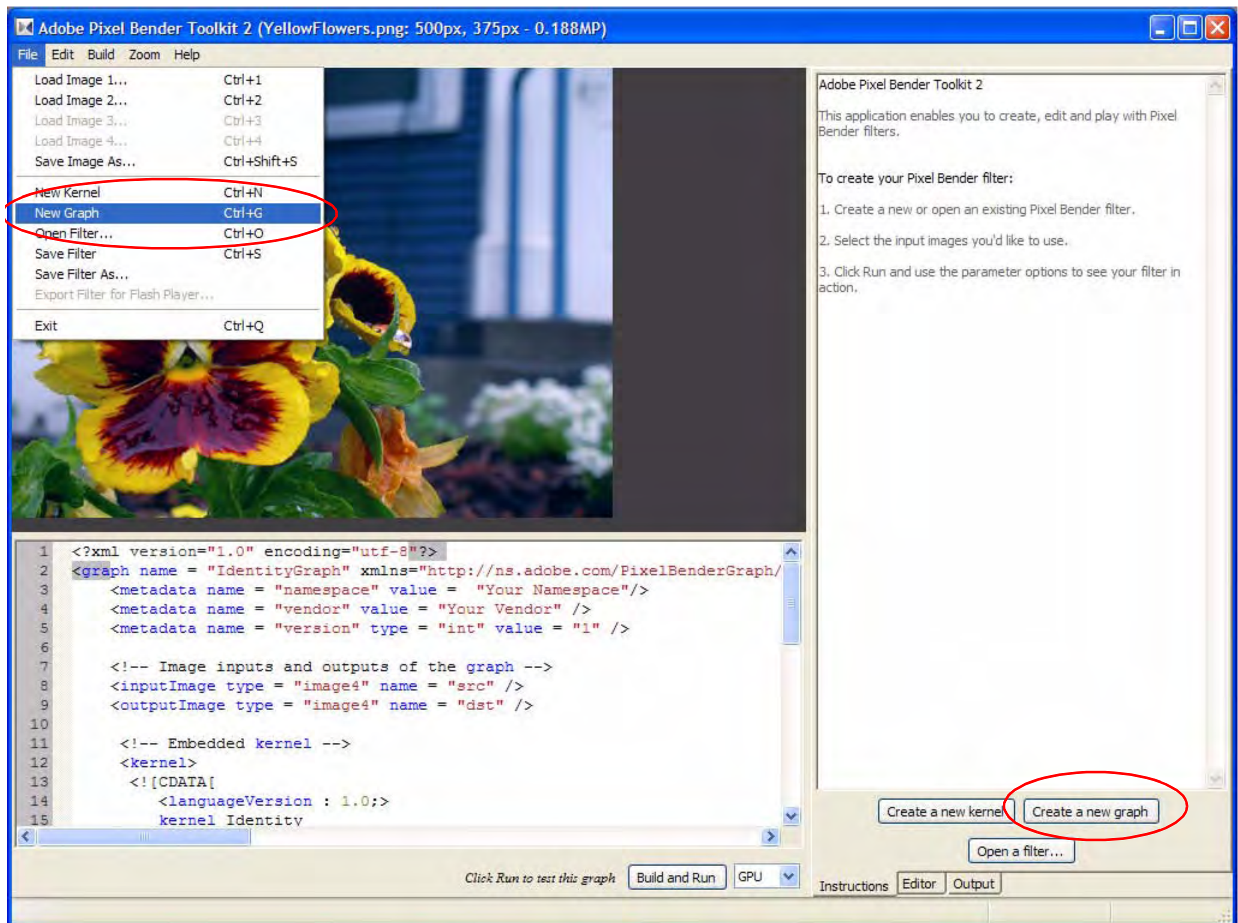
```
    </kernel>

    <kernel>
        <![CDATA[
            <languageVersion : 1.0;>
            kernel Blend
            <
                namespace: "Tutorial";
                vendor: "Adobe";
                version: 1;
            >
            {
                input image4 orig;
                input image4 glow;
                output float4 dst;

                void evaluatePixel() {
                    float4 sourcePixel=sampleNearest(orig, outCoord());
                    float4 blurredGlowPixel=sampleNearest(glow,outCoord());

                    sourcePixel.rgb=mix(
                        blurredGlowPixel.rgb, sourcePixel.rgb,
                        blurredGlowPixel.a);
                    dst=sourcePixel;
                }
            }
        ]]>
    </kernel>

    <!-- Nodes -->
    <node
        id="selectBright"
        name="SelectBright"
        vendor="Adobe"
        namespace="Tutorial"
        version="1"  >
            <evaluateParameters>
            <![CDATA[
                void evaluateParameters() {
                    selectBright::threshold=1.0 - (1.0 / amount);
                }
            ]]>
        </evaluateParameters>
    </node>

    <node
        id="boxblur1"
        name="BoxBlur" vendor="Adobe"
        namespace="Tutorial" version="1"  >
        <evaluateParameters>
            <![CDATA[
                void evaluateParameters() {
                    boxblur1::direction=float2(1.0, 0.0);
                    boxblur1::radius=int(ceil(amount));
                }
            ]]>
        </evaluateParameters>
    </node>

    <node
```

```
        id="boxblur2"
        name="BoxBlur" vendor="Adobe"
        namespace="Tutorial" version="1"  >
        <evaluateParameters>
            <![CDATA[
                void evaluateParameters() {
                    boxblur2::direction=float2(0.0, 1.0);
                    boxblur2::radius=int(ceil(amount));
                }
            ]]>
        </evaluateParameters>
    </node>

    <node
        id="composite"
        name="Blend"
        vendor="Adobe"
        namespace="Tutorial"
        version="1"  >
    </node>

    <!-- Connections -->
    <connect fromImage="graphSrc"
        toNode="selectBright" toInput="src"/>
    <connect fromImage="graphSrc"
        toNode="composite" toInput="orig" />
    <connect fromNode="selectBright" fromOutput="dst"
        toNode="boxblur1" toInput="src"/>
    <connect fromNode="boxblur1" fromOutput="dst"
        toNode="boxblur2" toInput="src"/>
    <connect fromNode="boxblur2" fromOutput="dst"
        toNode="composite" toInput="glow"/>
    <connect fromNode ="composite" fromOutput="dst"
        toImage="graphDst"/>
</graph>
```

Using the Graph Editor

The Pixel Bender Toolkit includes a Graph Editor, which works in the same way as the Kernel Editor.

- To open an existing graph program, choose File > Open Filter, and choose from among existing PBG files.
- To get a basic code framework for a new graph program, choose File > New Graph or click "Create a new graph".



Editing a graph program is just like editing a kernel program. When you run the program, the Filter Parameter UI section allows you to set the graph parameters, in the same way you set kernel parameters.

6 Metadata Hinting

Using Pixel Bender allows applications to load filters at run time; adding a new filter can be as simple as dropping a text file into a directory. However, this can make it difficult to create a sensible user interface for the filter. *Metadata hinting* allows an application to get extra information about a filter, in order to create a more effective UI.

Defining parameter metadata for a filter

The Pixel Bender language allows you to define metadata for each kernel parameter. There are predefined metadata values you can use to set constraints on the allowed values for a parameter, and to describe it so that the UI can present a reasonable way of setting it.

You can also create arbitrary, application-specific elements; these additional elements do not interfere with any system or application functionality.

Each metadata element associates a key name with a value. Define the metadata in a clause within the parameter specification. For example:

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

It is a good idea to provide these metadata elements to make your filters more usable. The metadata is made available to the client application after compilation, and helps the client determine how to present an appropriate UI that allows users to set the parameter value.

For details of the allowed types and usage, see "Kernel Specifications" in the *Pixel Bender Reference*.

Value constraint elements

A set of predefined metadata elements places constraints on the value that can be assigned to the parameter, or that is assigned to it by default. The data type of the element value must match the data type of the parameter. These value-constraint elements are predefined:

`minValue` `maxValue`	Minimum and maximum value of the parameter. It is recommended that you always set maximum and minimum values; if they are not set, the application defaults might be inappropriate.
`stepInterval`	Suggested step interval value within a range.

`defaultValue`	Default value of the parameter.
`previewValue`	A value for this parameter to be used when creating a preview of the filter. For example, while the default value of a blur radius might be 0 (no blur), a preview value might be 10, to provide a better preview of what the filter does.

The Pixel Bender compiler validates that the type-specific values are of the correct type, but it does not validate that the ranges are correct or that the default value is between the minimum and maximum values.

- Photoshop and After Effects enforce the minimum and maximum limits, and set the parameter values to their specified defaults on first run.
- Flash Player does not enforce the minimum and maximum limits, nor does it automatically set the default value. It is up to the developer using the filter to enforce them.

Display string elements

A set of predefined metadata elements allows the application to display user-friendly strings in the UI, in order to make the meaning and use of the parameter clear. These include:

`displayName`	The default name to display for the parameter. You can include language-specific elements for localization; see "Localizing display strings" on page 68.
`minValueName` `maxValueName`	Display strings for the minimum and maximum values.
`description`	A description of the use of a parameter.

Examples

```
parameter int angle
<
    minValue :          0;
    maxValue :          360;
    defaultValue :      30;
    previewValue :      15;
    description :       "measured in degrees";
    displayName :       "vector angle";
    minValueName :      "no angle";
    maxValueName :      "full rotation";
    stepInterval :      5;
>;
```

```
parameter float2 center
<
    minValue :          float2(0.0, 0.0);
    maxValue :          float2(2048.0, 2048.0);
    defaultValue :      float2(256.0, 256.0);
    previewValue :      float2(512.0, 512.0);
    stepInterval :      float2(10.0, 20.0);
    displayName :       "twirl center";
    description :       "the center of the twirl vortex";
    minValueName :      "upper left";
    maxValueName :      "lower right";
>;
```

Localizing display strings

To define a localized display name for a parameter, include a language-specific metadata element with a name in the form:

`displayName_`*`code`*

Append an IETF 2-character language code after an underscore to specify the language for which the value should be used.

An application can look for an element that matches the current locale. If the appropriate localized element is not available, the application uses the `displayName` element that has no language code as the default,.

For example:

```
parameter pixel4 color
<
    minValue:        float4(0.0);
    maxValue:        float4(1.0);
    defaultValue:    float4(1.0, 0.0, 0.0, 1.0);
    displayName:     "color";
    displayName_en:  "color";
    displayName_fr:  "couleur';
    displayName_de:  "farbe";
    minValueName:    "black";
    minValueName_fr: "noir";
    minValueName_de: "schwarz";
    maxValueName:    "white";
    maxValueName_fr: "blanc";
    maxValueName_de: "weiß";
>;
```

Parameter type specialization element

The `parameterType` metadata element allows you to further specify what a parameter is meant to represent, beyond its data type. This allows applications to create special controls for editing the value, such as an appropriate color picker for the various color types.

The value of the `parameterType` element is a string whose possible values depend on the data type of the parameter.

Parameter value type	`parameterType` element values	Description
`int`	`enum`	The parameter is a set of choices rather than a numeric value. The parameter must have a separate `enumValues` metadata element that specifies the choice list; see “Enumerated values” on page 71. An application can choose to display this to the user as a drop-down list or radio button rather than a numeric input.
	`frame`	The parameter is a frame number. A video application can choose to automatically set this to the current video frame number, or a Flash-based application can set this to be the current frame number of the Flash animation.
	`time`	The parameter is a time, expressed in milliseconds. This can be used to pass in the current time of a video file or animation to a time-varying filter.
`float`	`angleDegrees` `angleRadians`	The parameter is an angle. An angle of zero is along the positive X axis, and increases clockwise (from the positive X-axis down to the positive Y-axis). An application can choose to show a compass-style UI control to allow the user to set this parameter value.
	`percentage`	The parameter is a percentage. This does not imply a particular range, but an application can choose to default the range from 0.0 to 100.0 if the author does not specify `minValue` or `maxValue` information.
	`time`	The parameter is a time, expressed in seconds. This can be used to pass in the current time of a video file or animation to a time-varying filter.

Parameter value type	`parameterType` element values	Description
`float2`	`inputSize`	The parameter is the size (width and height) of one of the input images. An application can choose to set this value automatically without exposing it to the user applying this filter. The parameter value cannot be set automatically when the kernel is part of a larger graph, because the size of the input image may not be available. A parameter of this type can also have an `inputSizeName` metadata element; see "Distinguishing among multiple input images" on page 71.
	`position`	The parameter is a pixel position in an image. The position must be in the world coordinate system. An application can choose to allow a user to click within an image to set this position.

`float3`
`pixel3`
 	
`float4`
`pixel4`
	`colorLAB`
	`colorRGB`
	
	`colorCMYK`
	`colorRGBA`
		The parameter represents a color in LAB or RGB space, or in CMYK or RGBA space.
		
		An application can show a color wheel or an eyedropper to allow a user to set this parameter.
		
		Alternatively, you can use the `componentName` element to specify component names within the vector value of the parameter. See "Naming vector values" on page 72.

Examples

```
parameter float angle
<
    parameterType:       "angleDegrees";
    minValue :           0.0;
    maxValue :           359.9;
    defaultValue :       30.0;
    previewValue :       15.0;
    description :        "measured in degrees";
    displayName :        "vector angle";
    minValueName :       "no angle";
    maxValueName :       "full rotation";
    stepInterval :       5.0;
>;

parameter float2 mousePos
<
    parameterType :      "position";
    minValue :           float2(0.0);
    maxValue :           float2(1024.0);
    defaultValue :       float2(512.0);
>;
```

```
parameter pixel4 color
<
    parameterType: "colorCMYK";
>;
```

Enumerated values

Pixel Bender developers often use integer parameters as enumerated types, allowing the user a set number of choices, with each choice being distinct. When you do this, use the `enumValues` metadata element to associate each possible value with a descriptive string. This enables the application to display a meaningful UI, such as a drop-down list rather than a slider.

The `enumValues` element has a string value containing the text for each choice delimited by the vertical bar (|) character. If the string does not contain the correct number of choices, or the `parameterType` is not set as `enum`, this metadata element is ignored.

By default the first choice string maps to `minValue`, the second choice string maps to `minValue+1`, and so on. To change this mapping, following each choice string with an "=" sign and assign it a specific integer.

For example:

```
parameter int fractalTypes
<
    minValue :       0;
    maxValue :       2;
    defaultValue :   0;
    parameterType :  "enum";
    enumValues :     "Mandelbrot|Julia|Strange Attractors";
>;

parameter int fractalTypes
<
    minValue :       0;
    maxValue :       2;
    defaultValue :   0;
    parameterType :  "enum";
    enumValues :     "Mandelbrot=1|Julia=0|Strange Attractors=2";
>;
```

Distinguishing among multiple input images

If the `parameterType` for a `float2` parameter is set to `inputSize` and the kernel has more than one image input, use the `inputSizeName` metadata element to specify which of the input images this parameter describes.

Examples

```
<languageVersion: 1.0;>
kernel BlendCentered
<
    namespace : "AIF";
    vendor : "Adobe Systems";
     version : 2;
     description : "blend two images, position them so they are centered";
>
```

```
{
   parameter float blend
   <
      minValue :       0.0;
      maxValue :       1.0;
      defaultValue :   0.0;
   >;

   parameter float2 inputSize0
   <
      parameterType : "inputSize";
      inputSizeName : "inputImage0";
   >;

   parameter float2 inputSize1
   <
      parameterType : "inputSize";
      inputSizeName : "inputImage1";
   >;

   input image4 inputImage0;
   input image4 inputImage1;

   output pixel4 outputPixel;

   void evaluatePixel()
   {
      pixel4 pix0 = sampleNearest(inputImage0, outCoord());

      float2 center0 = inputSize0 / 2.0;
      float2 center1 = inputSize1 / 2.0;

      float2 sampleCoords = center1 + outCoord() - center0;

      pixel4 pix1 = sampleNearest(inputImage1, sampleCoords);

      outputPixel = mix( pix0, pix1, blend );
   }
}
```

Naming vector values

Vector parameters are common in Pixel Bender filters. The `parameterType` metadata element has some entries for common color values; however it is also useful to have a more general naming scheme. Use the `componentName` metadata element to associate each vector component with a descriptive string.

The string value contains a set of choices, one per vector component, separated by the vertical bar (|) character. If the string does not contain the correct number of choices, this metadata element is ignored.

For example:

```
parameter float3 colorXYZ
<
   componentName : "x|y|z";
>;
```

7 Developing for After Effects

This chapter introduces more techniques, with a focus on the use of Pixel Bender programs as effects in After Effects. All of the examples here use 4-channel input and output, as supported by After Effects.

After Effects kernel metadata

- 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.

For example, here is the simple Identity kernel, adapted to After Effects by including the extra metadata values, and using only 4-channel input and output images:

```
<languageVersion : 1.0;>
kernel Identity
<
    namespace: "After Effects";
    vendor: "Adobe Systems Inc.";
    version: 1;
    description: "This is an identity kernel";
    displayname: "Identity Kernel";
    category: "PB Effects";
>
{
    input image4 source;
    output pixel4 result;
    void evaluatePixel() {
        result = sampleNearest(source, outCoord());
    }
}
```

Accessing 4-channel values

- After Effects allows only 4-channel input and output images.

As our first exercise, we will modify the `evaluatePixel()` function to modulate the values of the different channels of the input image.

```
void evaluatePixel() {
    float4 temp = sampleNearest(source, outCoord());
    result.r = 0.5 * temp.r;
    result.g = 0.6 * temp.g;
    result.b = 0.7 * temp.b;
    result.a = temp.a;
}
```

This shows how you access different channels using the first letter of the channel name after the dot operator. This example operates on the color channels, and does not change the opacity (the alpha channel).

Here is another way to access the different channels, modulating all of the colors in a single operation:

```
void evaluatePixel() {
    float4 temp = sampleNearest(source, outCoord());
    float3 factor = float3(0.5, 0.6, 0.7);
    result.rgb = factor * temp.rgb;
    result.a = temp.a;
}
```

An even more compact version of this code is:

```
void evaluatePixel() {
    float4 temp = sampleNearest(source, outCoord());
    result = float4( 0.5, 0.6, 0.7, 1.0) * temp;
}
```

In this case, the channels are not specified explicitly, so all channels of `temp` and `result` are used.

An example of convolution

Convolution is a common image-processing operation that filters an image by computing the sum of products between the source image and a smaller image, called the convolution filter or convolution mask. Depending on the choice of values in the convolution mask, a convolution operation can perform smoothing, sharpening, edge detection, and other useful imaging operations.

In this chapter, we will build on this example to demonstrate some of the Pixel Bender features that are specific to After Effects.

This example begins by using convolution to smooth an image. An image can be smoothed by simply averaging pixels contained in the mask. The filter mask for smoothing is a 3x3 mask with a radius of 1:

```
[1, 1, 1]
[1, 1, 1]
[1, 1, 1]
```

The kernel defines this mask as a 3 by 3 constant, `smooth_mask`:

```
const float3x3 smooth_mask = float3x3( 1.0, 1.0, 1.0,
                                       1.0, 1.0, 1.0,
                                       1.0, 1.0, 1.0);
```

A user-defined function, `convolve()`, implements the core algorithm. It convolves the `smooth_mask` array over the entire input image. It then normalizes the result by dividing by 9 (the sum of the coefficients of the mask).

```
float4 convolve(float3x3 in_kernel, float divisor) {
    float4 conv_result = float4(0.0, 0.0, 0.0, 0.0);
    float2 out_coord = outCoord();
    for(int i = -1; i <= 1; ++i) {
        for(int j = -1; j <= 1; ++j) {
            conv_result += sampleNearest(source,
                out_coord + float2(i, j)) * pixelSize(src) * in_kernel[i + 1][j + 1];
        }
    }
    conv_result /= divisor;
```

```
    return conv_result;
}
```

Finally, the `evaluatePixel()` function calls the `convolve()` function, and returns the result in the kernel's output image:

```
void evaluatePixel() {
    float4 conv_result = convolve(smooth_mask, smooth_divisor);
    result = conv_result;
}
```

Here is the complete kernel that convolves this filter mask with an input image:

```
<languageVersion : 1.0;>
kernel ConvKernel
<
    namespace: "After Effects";
    vendor : "Adobe Systems Inc.";
    version : 1;
    description : "Convolves an image using a smoothing mask";
>
{
    input image4 source;
    output pixel4 result;
    const float3x3 smooth_mask = float3x3( 1.0, 1.0, 1.0,
                                           1.0, 1.0, 1.0,
                                           1.0, 1.0, 1.0);
    const float smooth_divisor = 9.0;

    float4 convolve(float3x3 in_kernel, float divisor) {
        float4 conv_result = float4(0.0, 0.0, 0.0, 0.0);
        float2 out_coord = outCoord();
        for(int i = -1; i <= 1; ++i) {
            for(int j = -1; j <= 1; ++j) {
                conv_result += sampleNearest(source,
                    out_coord + float2(i, j)) * in_kernel[i + 1][j + 1];
            }
        }
        conv_result /= divisor;
        return conv_result;
    }

    void evaluatePixel() {
        float4 conv_result = convolve(smooth_mask, smooth_divisor);
        result = conv_result;
    }

    region needed( region output_region, imageRef input_index ) {
        region result = output_region;
        result = outset( result, float2( 1.0, 1.0 ) );
        return result;
    }

    region changed( region input_region, imageRef input_index ) {
        region result = input_region;
        result = outset( result, float2( 1.0, 1.0 ) );
        return result;
    }
} //kernel ends
```

Kernel parameters in After Effects

After Effects defines optional parameter metadata to emulate its own set of parameters:

- `aeDisplayName`

  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. For example:

```
parameter float myKernelParam
<
   minValue: 0.0;
   maxValue: 100.0;
   defaultValue: 0.0;
   aeDisplayName: "Pretty Display Name";
>;
```

- `aeUIControl`

  This metadata value tells After Effects what kind of UI control to use to allow user input for this parameter. The allowed values depend on the parameter's data type:

Parameter data type	Allowed control values

`int`	`aeUIControl:"aePopup"`
	Requires an additional metadata value to specify the popup menu values, as a string with individual items separated by the pipe character:
	`aePopupString:"Item 1|Item 2|Item 3"`

`float`	`aeUIControl:"aeAngle"`
	`aeUIControl:"aePercentSlider"`
	`aeUIControl:"aeOneChannelColor"`

`float2`	`aeUIControl: "aePoint"`
	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:
	`aePointRelativeDefaultValue: float2(x, y)`
	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.

`float3` `float4`	`aeUIControl: "aeColor"` In `float4` color values, the alpha channel is always 1, because in After Effects, color controls only support opaque colors.

Here is an example of a parameter that uses the After Effects metadata. We can add this parameter to the convolution example, in order to blend the convolution results with the original image:

```
parameter float blend_with_original
<
    minValue: 0.0;
    maxValue: 100.0;
    defaultValue: 0.0;
    description: "Amount to blend with original input"; //comment only, ignored in AE
    aeDisplayName: "Blend Factor"; //AE-specific metadata
    aeUIControl: "aePercentSlider"; //AE-specific metadata
>;
```

- If you do not supply minimum, maximum, or default values for a parameter, After Effects chooses values that are appropriate to the parameter's data type.
    - Integer parameters get a minimum value of 0, a maximum value of 100 and a default value of 0.
    - Float parameters get a minimum value of 0.0, a maximum value of 1.0 and a default value of 0.0.
- The `description` value is not used in After Effects. You can still use it as a comment in your code.
- After Effects uses the `aeDisplayName`, "Blend Factor" to identify the parameter in the UI. If it was not supplied, the application would use the parameter name, "blend_with_original."
- The `aeUIControl` value tells After Effects to represent this parameter with a percent slider.

We can use this parameter in the `evaluatePixel()` function:

```
void evaluatePixel() {
    float4 conv_result = convolve(smooth_mask, smooth_divisor);
    result = mix(conv_result, sampleNearest(source, outCoord()), blend_with_original);
}
```

The built-in function `mix()` performs a linear interpolation between the original input and the convolution result.

Expanded convolution kernel

This version of the convolution kernel adds a new parameter to choose between a smoothing and sharpening operation. It makes these changes to the original convolution kernel:

- Adds a sharpen convolution mask and its normalizing constant (1.0).
- Adds a new integer parameter (`kernel_type`) to choose between smoothing and sharpening filter.
- The core function, `convolve()`, is not changed. The `evaluatePixel()` function is modified to pass the chosen filter mask to the `convolve()` function.

```
<languageVersion : 1.0;>
kernel ConvKernel
<
    namespace: "After Effects";
    vendor : "Adobe Systems Inc.";
    version : 1;
    description : "Convolves an image using a smoothing or sharpening mask";
>
{
    input image4 source;
```

```
    output pixel4 result;
    parameter int kernel_type
    <
        minValue: 0;
        maxValue: 1;
        defaultValue: 0;
        aeDisplayName: “Kernel Type”;
        aeUIControl: "aePopup";
        aePopupString: "Smooth|Sharpen";
    >;
    parameter float blend_with_original
    <
        minValue: 0.0;
        maxValue: 100.0;
        defaultValue: 0.0;
        aeDisplayName: "Blend Factor"; //AE-specific metadata
        aeUIControl: "aePercentSlider"; //AE-specific metadata
    >;
    const float3x3 smooth_mask = float3x3( 1.0, 1.0, 1.0,
                                            1.0, 1.0, 1.0,
                                            1.0, 1.0, 1.0);
    const float smooth_divisor = 9.0;
    const float3x3 sharpen_mask = float3x3(-1.0, -1.0, -1.0,
                                            -1.0, 9.0, -1.0,
                                            -1.0, -1.0, -1.0);
    const float sharpen_divisor = 1.0;

    float4 convolve(float3x3 in_kernel, float divisor) {
        float4 conv_result = float4(0.0, 0.0, 0.0, 0.0);
        float2 out_coord = outCoord();
        for(int i = -1; i <= 1; ++i) {
            for(int j = -1; j <= 1; ++j) {
                conv_result += sampleNearest(source,
                    out_coord + float2(i, j)) * in_kernel[i + 1][j + 1];
            }
        }
        conv_result /= divisor;
        return conv_result;
    }

    void evaluatePixel() {
        float4 conv_result;
        if (kernel_type == 0) {
            conv_result = convolve(smooth_mask, smooth_divisor);
        }
        else {
            conv_result = convolve(sharpen_mask, sharpen_divisor);
        }

        result = mix(conv_result, sampleNearest(source, outCoord()),
                    blend_with_original);
    }
}
```

Dependent functions

This section demonstrates how you can define dependent functions to optimize the convolution kernel.

The `evaluatePixel()` function is called for each output pixel, so in its current form, the check for which type of convolution to perform is also made for each pixel. Since this value doesn't change over entire frame, the per-pixel check can be eliminated if we can compute the mask first. We can do this using dependent member variables and the `evaluateDependents()` function.

The divisor for any convolution mask can also be calculated by adding the coefficients of that mask, so we can compute that ahead of time too. This makes the kernel more easily extensible as well, as we can add any other convolution mask without worrying about the divisor.

Values declared as dependent are initialized within the body of `evaluateDependents()`, which is run just once for each frame. After that, these values are considered read-only and are held constant over all pixels as `evaluatePixel()` is applied.

We will make these changes to the previous version of the convolution kernel:

- Add an `emboss` convolution mask, along with a new item or the popup list for the `kernel_type` parameter.
- Add two dependent variables, `fiter_mask` and `mask_divisor`, to store the selected convolution mask and associated divisor.
- Define `evaluateDependents()` to copy the selected convolution mask into the dependent variable `filter_mask`.
- Remove the mask-specific divisors, and add the divisor calculation to the new `evaluateDependents()` definition, where it depends on the selected convolution mask. Store the computed value in the dependent variable `mask_divisor`.
- Redefine `evaluatePixel()` so that it passes these two dependent variables to the core function, `convolve()`.

Everything else is the same. Now we are not comparing `kernel_type` for each pixel, but for each frame.

Here is the modification of the kernel code:

```
<languageVersion : 1.0;>
kernel ConvKernel
<
    namespace: "After Effects";
    vendor : "Adobe Systems Inc.";
    version : 1;
    description : "Convolves an image using a smooth, sharpen, or emboss mask";
>
{ //kernel starts
    input image4 source;
    output pixel4 result;
    parameter int kernel_type
    <
        minValue: 0;
        maxValue: 1;
        defaultValue: 0;
        aeDisplayName: “Kernel Type”;
        aeUIControl: "aePopup";
        aePopupString: "Smooth|Sharpen|Emboss";
```

```
>;
parameter float blend_with_original
<
    minValue: 0.0;
    maxValue: 100.0;
    defaultValue: 0.0;
    aeDisplayName: "Blend Factor"; //AE-specific metadata
    aeUIControl: "aePercentSlider"; //AE-specific metadata
>;
const float3x3 smooth_mask = float3x3( 1.0, 1.0, 1.0,
                                      1.0, 1.0, 1.0,
                                      1.0, 1.0, 1.0);
const float3x3 sharpen_mask = float3x3(-1.0, -1.0, -1.0,
                                      -1.0, 9.0, -1.0,
                                      -1.0, -1.0, -1.0);
const float3x3 emboss_mask = float3x3( -2.0, -1.0, 0.0,
                                      -1.0, 1.0, 1.0,
                                      0.0, 1.0, 2.0);
dependent float3x3 filter_mask;
dependent float mask_divisor;

float findDivisor(float3x3 mask) {
    float divisor = 0.0;
    for(int i = 0; i < 3; ++i) {
        for(int j = 0; j < 3; ++j) {
            divisor += mask[i][j];
        }
    }
return divisor;
}

void evaluateDependents() {
    if (kernel_type == 0) {
        filter_mask = smooth_mask;
    }
    else if (kernel_type == 1) {
        filter_mask = sharpen_mask;
    }
    else if (kernel_type == 2) {
        filter_mask = emboss_mask;
    }
    mask_divisor = findDivisor(filter_mask);
}

float4 convolve(float3x3 in_kernel, float divisor) {
    float4 conv_result = float4(0.0, 0.0, 0.0, 0.0);
    float2 out_coord = outCoord();
    for(int i = -1; i <= 1; ++i) {
        for(int j = -1; j <= 1; ++j) {
            conv_result += sampleNearest(source,
                out_coord + float2(i, j)) * in_kernel[i + 1][j + 1];
        }
    }
    conv_result /= divisor;
    return conv_result;
}

void evaluatePixel() {
    float4 conv_result = convolve(filter_mask, mask_divisor);
    result = mix(conv_result, sampleNearest(source, outCoord()),
                blend_with_original);
```

```
    }

    region needed( region output_region, imageRef input_index ) {
        region result = output_region;
        result = outset( result, float2( 1.0, 1.0 ) );
        return result;
    }

    region changed( region input_region, imageRef input_index ) {
        region result = input_region;
        result = outset( result, float2( 1.0, 1.0 ) );
        return result;
    }
}//kernel ends
```

8 Developing for Flash

This chapter discusses how to work with Pixel Bender in Flash, Flex, and ActionScript.

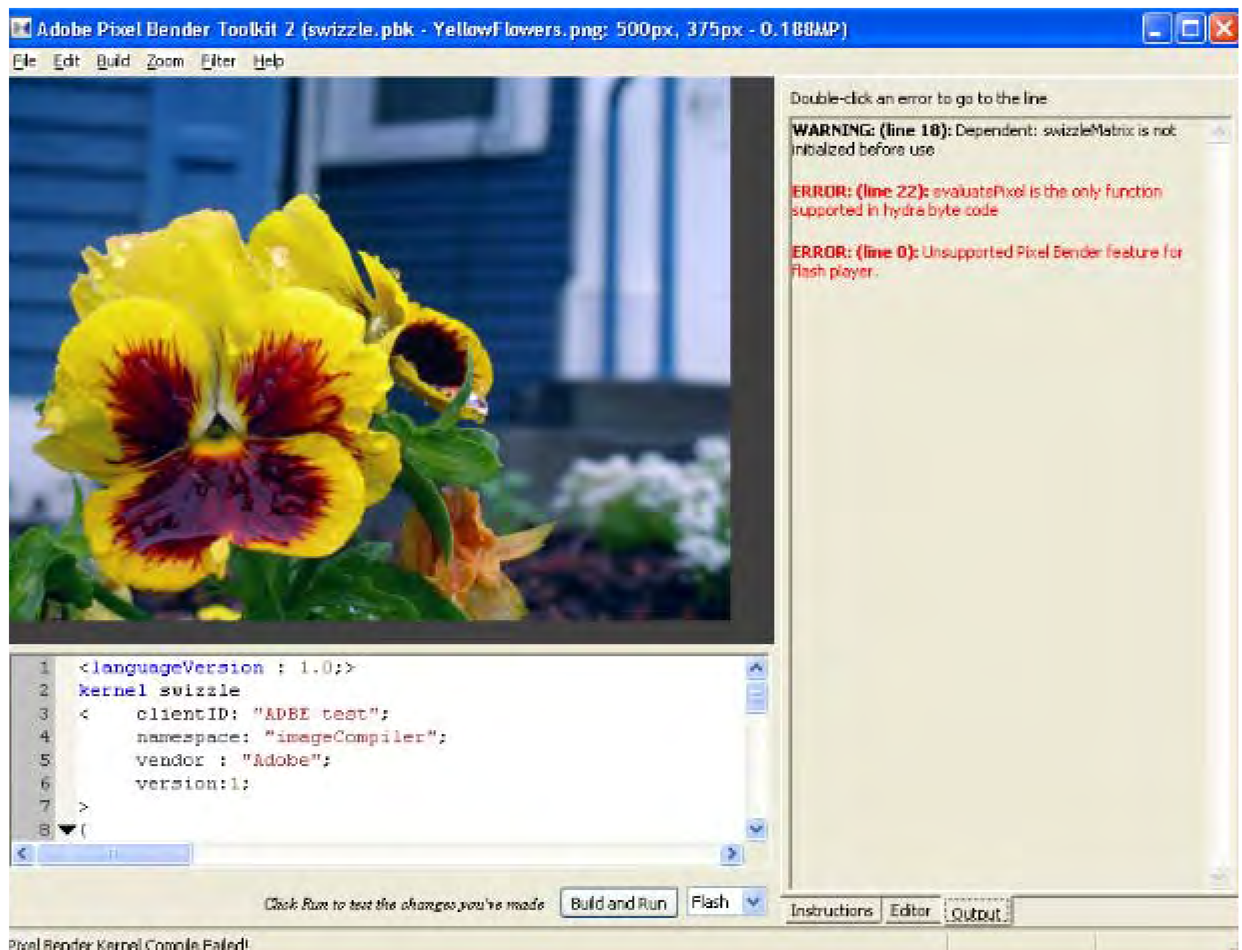
Using the Flash preview feature in the Pixel Bender Toolkit

Pixel Bender Toolkit 2.0 and later can directly show the result of a Pixel Bender filter running in Flash Player.

To use the Flash preview option:

1. Choose **File > Load Image 1**.
2. Click **Open a filter** to load the filter you wish to preview.
3. In the drop-down menu next to the **Build and Run** button, select **Flash**.
4. Click **Build and Run**.

If the loaded filter is not Flash compatible, the Pixel Bender Toolkit displays error messages in the output panel.



Embedding a Pixel Bender filter in a SWF

This modifies the example to allow the filter to be embedded within the SWF, so that it does not need to be dynamically loaded at runtime.

```
package
{
    import flash.display.*;
    import flash.events.*;
    import flash.filters.*;
    import flash.net.*;
    import flash.utils.ByteArray;
    // SWF Metadata
    [SWF(width="600", height="400", backgroundColor="#000000", framerate="30")]

    public class PB extends Sprite {
        //the file that contains the binary bytes of the PixelBender filter
        [Embed("testfilter.pbj", mimeType="application/octet-stream")]
        private var TestFilter:Class;

        //The image to display the filter on
        [Embed(source="image.jpg")]
        private var image:Class;

        private var im:Bitmap;

        public function PB():void {
            im = new image() as Bitmap;
            addChild(im);
            //Pass the loaded filter to the Shader as a ByteArray
            var shader:Shader = new Shader(new TestFilter() as ByteArray);
            shader.data.amount.value = [100];
            var filter:ShaderFilter = new ShaderFilter(shader);
            //add the filter to the image
            im.filters = [filter];
        }
    }
}
```

Making Pixel Bender filters into ActionScript libraries

You can create a SWC (a redistributable library of ActionScript classes) that contains a Pixel Bender filter encapsulated within an ActionScript class. You can use this technique to create SWC libraries of custom Pixel Bender filters that can be used in Flash Player projects developed with Flex Builder, MXMLC, and Flash.

Encapsulate a filter in an ActionScript class

This simple example shows how to encapsulate a custom Pixel Bender filter inside an ActionScript 3 class, which you can then use and re-use as you would any other built-in filter.

Document Class: PBFilter.as

```
package
{
    import flash.display.Bitmap;
```

```
    import flash.display.Sprite;

    // SWF Metadata
    [SWF(width="600", height="400", backgroundColor="#000000", framerate="30")]

    public class PBFilter extends Sprite
    {
        //The image to display the filter on
        [Embed(source="image.jpg")]
        private var image:Class;

        private var im:Bitmap;

        public function PBFilter():void
        {
            im = new image() as Bitmap;
            addChild(im);

            var filter:TestFilter = new TestFilter();
                filter.value = 30;

            //add the filter to the image
            im.filters = [filter];
        }

    }
}
```

Filter Class: TestFilter.as

```
package
{
    import flash.display.Shader;
    import flash.filters.ShaderFilter;
    import flash.utils.ByteArray;

    public class TestFilter extends ShaderFilter
    {
        //the file that contains the binary bytes of the PixelBender filter
        [Embed("testfilter.pbj", mimeType="application/octet-stream")]
        private var Filter:Class;

        private var _shader:Shader;

        public function TestFilter(value:Number = 50)
        {
            //initialize the ShaderFilter with the PixelBender filter we
            //embedded
            _shader = new Shader(new Filter() as ByteArray);

            //set the default value
            this.value = value;
            super(_shader);
        }

        //This filter only has one value, named value
        public function get value():Number
        {
            return _shader.data.amount.value[0];
        }
```

```
        public function set value(value:Number):void
        {
            //not that pixel bender filters take an array of values, even
            //though we only have one in our example
            _shader.data.amount.value = [value];
        }

    }
}
```

Create the SWC library

You can create the SWC in Flex Builder, or using the `compc` command line tool included in the Flex SDK.

To use Flex Builder:

1. Add the Flex SDK to Flex Builder, using Window > Preferences > Flex > Installed Flex SDKs > Add. The Flex SDK contains classes that we need to create and compile our filter class.
2. Create a new “Flex Library Project” using File > New > Flex Library Project.
3. Specify a project name and where the project will be stored.
4. Under Flex SDK version, select "Use a specific SDK" and choose the Flex SDK you just added (which should be named Flex 3.2).

Using Pixel Bender kernels as blends

When you use a kernel as a blend within Flash Player, calling the sample function gives the correct result (blends are always assumed to be pointwise), but the `outCoord()` function always returns `(0.0, 0.0)`.