ADOBE® PIXEL BENDER®

PIXEL BENDER 3D REFERENCE

Adobe Pixel Bender 3D Reference.

Contents

Preface

The Pixel Bender® technology delivers a common image and video processing infrastructure which provides automatic runtime optimization on heterogeneous hardware. Pixel Bender is a high-performance graphics programming tool intended for image processing. You can use the Pixel Bender kernel and graph languages to implement image processing algorithms in a hardware-independent manner.

Pixel Bender 3D is a version of the Pixel Bender kernel language which allows you to produce vertex and fragment shaders that run on 3D hardware to generate output images. These kernels operate on 3D objects and affect their appearance. Pixel Bender 3D is built on the Adobe® Flash Player® Molehill API, which handles the display of 3D objects in Flash Player. Pixel Bender 3D provides a quick and convenient way of supplying shaders to Molehill. Where appropriate hardware is available, Pixel Bender 3D is GPU accelerated.

This document, *Adobe Pixel Bender 3D Guide and Reference*, is a both a developer’s guide and reference manual for the 3D version of the Pixel Bender kernel language.

The reference is intended for programmers who wish to use Pixel Bender 3D to develop shaders for 3D objects in Flash Player. It assumes familiarity with the 2D version Pixel Bender kernel language, as well as familiarity with the intended target application. For more information about the structure and usage of Pixel Bender kernel language, see the *Adobe Pixel Bender Reference* and *Pixel Bender Developer’s Guide*.

1 Pixel Bender 3D Language Overview

Pixel Bender is a high-performance graphics programming technology intended for image processing. This document, *Adobe Pixel Bender 3D Guide and Reference*, is a both a developer's guide and reference manual for the 3D version of the Pixel Bender kernel language.

Because Pixel Bender 3D is an extension of the Pixel Bender kernel language, users are expected to have previous familiarity with the syntax and program structure of that language. For detailed descriptions, see the *Adobe Pixel Bender Reference* and *Pixel Bender Developer's Guide*. For differences between the 3D version and the 2D versions, see xx.

This chapter provides an overview of the capabilities of the language, an introduction to the expected usage, and some examples. The rest of this book provides a language reference and specification.

General usage guidelines

Pixel Bender 3D allows you to write shaders that operate on 3D objects and affect their appearance. The outputs of Pixel Bender 3D kernels are the positions of vertices, and the color of surfaces.

The primary operations are:

- Changing the positions of vertices in a mesh
- Setting the color of each point or fragment of a surface

You can use Pixel Bender 3D shaders to show the effect of lighting on an object, and achieve very fine control of the exact color of each point on the surface.

After writing the kernels that define these operations, you must process them with a command-line utility to produce code that you can load and run in Flash Player, using the Molehill API.

Differences between 2D and 3D Pixel Bender

- Pixel Bender 3D has more extensive array support than Pixel Bender kernel language.
- The following are not available in Pixel Bender 3D:
  - The `imageRef` and region data types
  - Region functions
  - Inverse trigonometric functions
  - The `evaluateDependents` function
- No graph language is availabe for Pixel Bender 3D.

Kernel types

Pixel Bender 3D divides functions that affects vertex *position* (warping, boning and skinning, morphing and so on) from functions that affect vertex and surface *appearance*. This allows you to build up libraries of useful kernels that you can "mix and match."

There are two basic types of Pixel Bender 3D kernel, *vertex* kernels and *material* kernels.

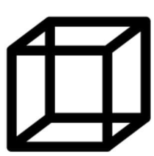
- A vertex kernel takes as input information about a vertex such as the position and normal. It must produce as output the new position of the vertex in clip space. Optionally, it can compute new information that is passed on to the next stage in the process.
- A material kernel is concerned with the appearance of vertices and of fragments. The first section of a material kernel deals with the appearance of each vertex, and the second deals with the appearance of *fragments* on the surfaces of the mesh. (A fragment is, roughly speaking, the area of a surface covering a single pixel).

Kernel compilation

Kernels are stored as plain text files, then compiled to create vertex and fragment shaders that can be loaded and run in Flash Player. During the compilation process:

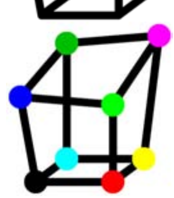
- A vertex kernel produces a *vertex program*.
- A material kernel produces a *material vertex program* and a *fragment program*.
- The vertex program and the material vertex program are combined (*welded*) to produce the vertex shader.
- The fragment program becomes the fragment shader.

This shows what happens to the object as each program runs:



Original object with 8 vertices

The *vertex program* changes the *position* of the vertices



The *material vertex program* changes the *appearance* of the vertices (such as color)

The *fragment program* changes the appearance of the surface fragments

The run-time sequence

The Pixel Bender 3D runtime, Molehill, runs each of the programs in sequence, using the output of each one as the input to the next one to calculate and render changes to all parts of the original object.

1. For every vertex in the mesh, Molehill runs the vertex program (produced by the vertex kernel). This reads values from the mesh (*input vertex* values), takes input from parameters, and produces a set of *output vertex* values. The *output vertex* values become available as *input vertex* values for the material vertex program (produced by the material kernel).

2. For every vertex in the mesh, Molehill runs the material vertex program (produced by the material kernel), with access to the *output vertex* values from the vertex program, as well as the same *input vertex* values the vertex kernel was using (that is, values that came directly from the vertices in the mesh) and input parameters.

   The material vertex program produces as output a set of appearance values for that vertex, such as color and UV coordinates.

3. For every surface in the mesh, Molehill collects all of the outputs produced by the vertex program and material vertex program. It divides the surface up into fragments, and for each fragment runs the fragment program (produced by the material kernel) to determine the color of that fragment. The outputs from the vertex programs are available to the fragment shader and are interpolated (in correct perspective) across the surface.

Compiling kernels and loading shaders

Vertex and material kernels written in the Pixel Bender 3D language as described in this document are stored as text files. We recommend that you name vertex kernels with the extension `.pbvk` and material kernels with the extension `.pbmk`.

1. For every kernel, run the command line utility `pb3dutil`.

   ▷ For a vertex kernel, specify a single output file, the vertex program. For example:

   ```
   pb3dutil vertexKernel.pbvk vertexProgram.pbasm
   ```

   ▷ For a material kernel, specify two output files, the material vertex program and the fragment program. For example:

   ```
   pb3dutil materialKernel.pbmk matVertProg.pbasm fragProgram.pbasm
   ```

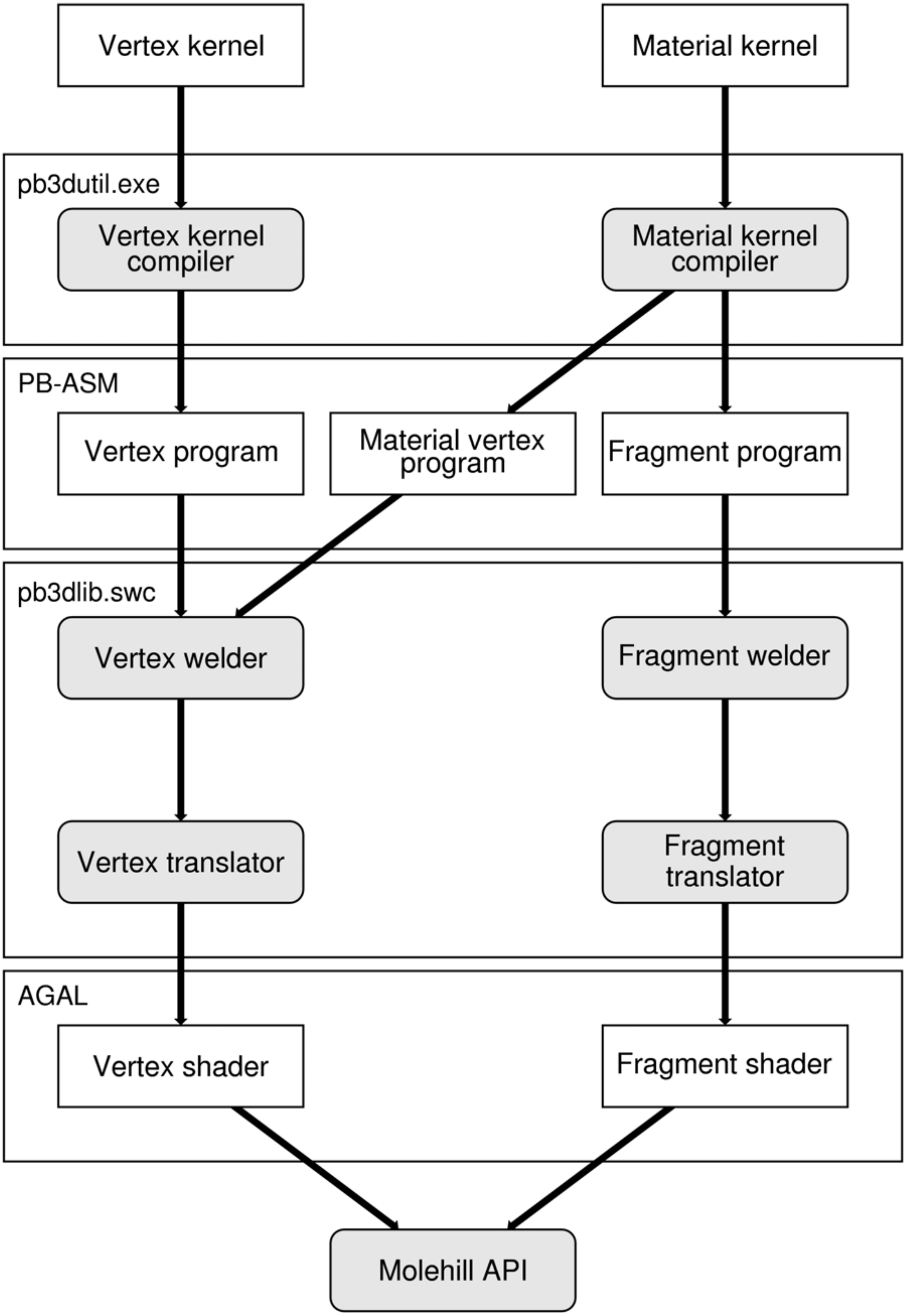
   The three resulting files contain a high-level assembler representation of your kernels, in a language known as PB-ASM.

2. In your ActionScript code, load the PB-ASM files and choose the combination of vertex and material kernels you want to apply to a given object.

3. Use the `welder` class to join the kernels together correctly, then use the `translator` class to turn them into AGAL (Adobe's low-level graphics assembler language).

4. Pass the AGAL through to the Molehill API , set up the vertex buffer and parameter inputs, and render the result.

The following figure shows the processing workflow.


Vertex kernel
Material kernel
pb3dutil.exe
Vertex kernel compiler
Material kernel compiler
PB-ASM
Vertex program
Material vertex program
Fragment program
pb3dlib.swc
Vertex welder
Fragment welder
Vertex translator
Fragment translator
AGAL
Vertex shader
Fragment shader
Molehill API

2 Kernel Specifications

Each Pixel Bender 3D program is specified by one string, which contains a set of metadata enclosed in angle brackets that describes the kernel, and a set of members enclosed in curly braces that define the filtering operation.

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

Every kernel must begin with the `languageVersion` statement, which identifies the version of the Pixel Bender 3D kernel language in which this kernel is written, followed by the kernel definition.

The kernel definition must begin with the kernel type, `vertex` or `material`.

- The first part of the kernel definition, a set of *metadata pairs* enclosed in angle brackets, has the same syntax for both types; see "Kernel metadata syntax" on page 10
- The second part of the kernel definition, a set of *kernel members* enclosed in curly braces, are variable declarations and function definitions. The member requirements differ for the two types; see:
  - "Vertex kernel member syntax" on page 14
  - "Material kernel member syntax" on page 20

Kernel metadata syntax

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

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

These metadata values are predefined:

`namespace`	Required. A string, the namespace within which this kernel is defined. The `namespace` value is used in combination with the other filter identifiers to determine the actual namespace, so it need not be globally unique. You can use it, for example, to distinguish categories of kernels.
`vendor`	Required. A string, the vendor supplying this kernel.

`version`	Required. A positive integer value, the version number of this implementation of this kernel. This is distinct from the kernel language version specified in the `languageVersion` element.
`description`	Optional. A string describing the purpose of this kernel. Applications that integrate with Pixel Bender 3D have access to this value, and can use it to create menu items, tooltips, or other UI elements.

For example:

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

You can define additional named metadata values. Kernel metadata values must be one of these data types:

```
int, int2, int3, int4
float, float2, float3, float4
float2x2, float3x3, float4x4
bool, bool2, bool3, bool4
string
```

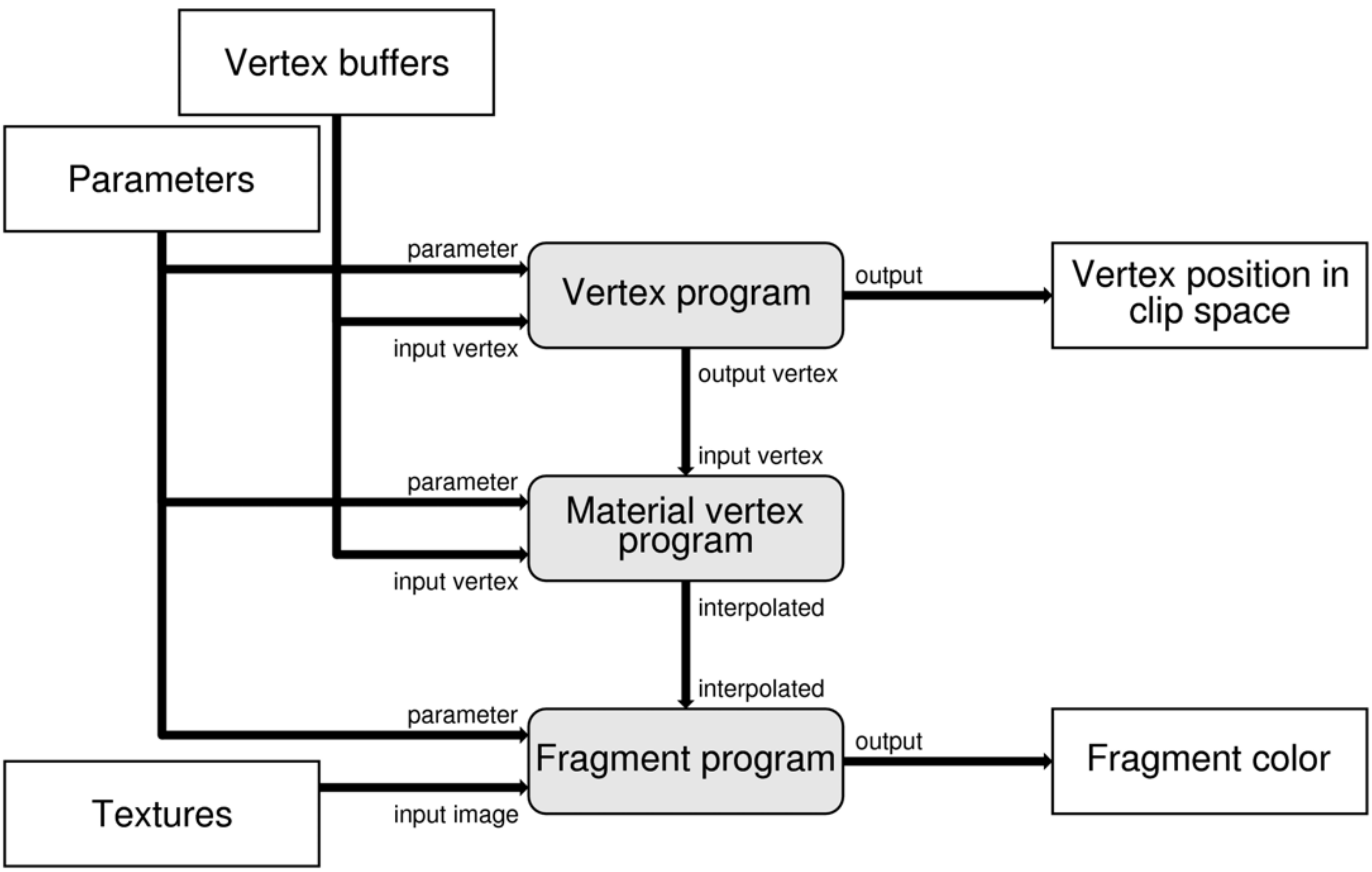
For `int`, `float`, and `bool`, the type is deduced automatically. For other types, specify a constant of the correct type (such as `float2( 1.0, -1.0 )`), or a string delimited by double quotes.

Kernel variable types

The input and output variables of both vertex and material kernels can be of these types:

Parameter	Set by the caller, with a developer-defined meaning. Values are constant across all vertices and across all fragments, and can be read by all parts of the system.
Input image	An input texture. Values are constant across all vertices and fragments. An input image can be sampled only by the `evaluateFragment()` function of a material kernel.
Output	▶ In a vertex kernel, the output of the vertex program; that is, the clip space position of the vertex. It is set once per vertex. It can only be accessed by the vertex program. ▶ In a material kernel, the output of the fragment program; that is, the color of the fragment. It is set once per fragment. It can only be accessed by the fragment program
Input vertex	The input vertex variables are set from the vertex buffers, which contain information about each vertex such as position, normal and color. Each input vertex variable must have metadata that indicates what attribute of the vertex it refers to; use the keyword strings such as `PB3D_POSITION`, `PB3D_NORMAL`. The type of the variable must be correct for that attribute. Values are updated for every vertex that is processed by the vertex program and the material vertex program. See "Connecting kernels to vertex buffers" on page 12.

Output vertex	Computed by the vertex program. The value is then available to the material vertex program and is constant for a given vertex. See "Connecting the vertex program to the material vertex program" on page 14.
Interpolated	Computed by the material vertex program for each vertex it processes. It is read by the fragment program and the value is automatically interpolated correctly. See "Connecting the material vertex program to the fragment program." on page 14.



Connecting kernels to vertex buffers

The vertex buffers that your kernels operate on are connected to the kernels themselves through the `input vertex` variables. Each `input vertex` variable has metadata that connects it to a given attribute of a vertex, and its type must be correct for that attribute. The metadata includes:

- An `id` whose value must be one of the supported keywords listed here.
- For indexed value types, an `index` value.

For example, an `input vertex` variable that will be set to the vertex position would be declared like this:

```
input vertex float3 vertexPosition
<
    id : "PB3D_POSITION";
>;
```

Vertex buffer attributes that are indexed must also include the index value:

```
input vertex float weight0
<
    id : "PB3D_WEIGHT";
    index : "0";
>;

input vertex float weight1
<
    id : "PB3D_WEIGHT";
    index : "1";
>;
```

These value types are allowed for `input vertex` values:

Keyword	Data type	Description
Non-indexed types		
`PB3D_POSITION`	`float3`	Position coordinate vector
`PB3D_NORMAL`	`float3`	Normal vector
`PB3D_COLOR`	`float3`	Color coordinate vector. Color inputs are RGB
`PB3D_TANGENT`	`float3`	Tangent vector
`PB3D_BINORMAL`	`float3`	Binormal (bitangent) vector
`PB3D_TEXTURE_BINORMAL`	`float3`	Texture binormal (bitangent) vector
`PB3D_TEXTURE_COORDS`	`float3`	Texture coordinate vector
`PB3D_TEXTURE_TANGENT`	`float3`	Texture tangent vector
`PB3D_UV`	`float2`	Generic parameter vector
Indexed types		
`PB3D_WEIGHT`	`float`	Skin influence weighting value
`PB3D_BONE_INDEX`	`int`	Offset into the bone transform array
`PB3D_TARGET_BINORMAL`	`float3`	Morph targets for mesh morphing of the binormals
`PB3D_TARGET_NORMAL`	`float3`	Morph targets for mesh morphing of the normals
`PB3D_TARGET_POSITION`	`float3`	Morph targets for mesh morphing of the positions
`PB3D_TARGET_TANGENT`	`float3`	Morph targets for mesh morphing of the tangents
`PB3D_TARGET_TEXTURE_BINORMAL`	`float3`	Morph targets for mesh morphing of the texture binormals (vtangents)
`PB3D_TARGET_TEXTURE_TANGENT`	`float3`	Morph targets for mesh morphing of the texture tangents

Connecting the vertex program to the material vertex program

The `output vertex` values in the vertex program connect to `input vertex` values in the material vertex program.

The metadata on each output vertex value must match the metadata on the corresponding input vertex value. The most common use for this is to pass an updated vertex position to the material vertex program, as in "Example 2: Changing the vertex position" on page 19.

Connecting the material vertex program to the fragment program.

Interpolated values written by the material vertex program are available for reading in the fragment program. The interpolated values are written once per vertex in the material vertex program.

For example, three vertices define a triangle that is shaded by the fragment program. When the fragment program reads the interpolated values, they are correctly interpolated in perspective across the triangle.

Vertex kernel member syntax

A vertex kernel defines the position of a vertex. It takes input information about a vertex such as the position and normal, and must produce as output the new position of the vertex in clip space.

A vertex kernel must contain at least an `evaluateVertex()` function definition and a single `float4` *output* value; all other members are optional:

```
{
    [declarations]

    output float4 vertexClipPosition;

    [support functions]
    void evaluateVertex()
    {
        statements
    }
}
```

The kernel must set the `output` value to the position of the vertex in clip space. Declarations can also include parameters, constants, input vertex and output vertex values.

The main function, `evaluateVertex()`, is applied to each vertex of the input mesh.

Vertex kernel declarations

Before the function definitions, a vertex kernel definition can contain these declarational members:

Declaration	Syntax
`parameter` (kernel contains zero or more)	`parameter` *`type name`* `<` *`name1`* `:` *`value1`*`;` *`name2`* `:` *`value2`*`;` `...` `>;`
Parameters are set before a kernel is executed and are read-only within kernel functions. Parameters can be of any type except `image`. Float arrays used as parameters must have sizes determined at compile time. A parameter can have optional metadata attached to it, as one or more name-value pairs enclosed in angle brackets. See “Parameter metadata” on page 16.	
`const` (kernel contains zero or more)	`const` *`type name=compile-time_expression`*`;`
The value of the constant is determined at compile time. The use of `const` is recommended for constant definitions, rather than `#define`.	
`input vertex` (kernel contains zero or more)	`input vertex` *`type name`* `<` `id :` *`type_keyword_string`*`;` `[index :` *`int`*`];` `>;`
A variable connected to an attribute of a vertex, such as its position, normal, or color. Each variable must have metadata attached to it to indicate which attribute it represents; see “Connecting kernels to vertex buffers” on page 12.	
`output vertex` (kernel contains exactly one)	`output vertex` *`type name`* `<` `id :` *`type_keyword_string`*`;` `[index :` *`int`*`];` `>;`
An output vertex to be passed on to a material vertex program created by a material kernel. Each output vertex variable must have metadata attached to it to indicate which attribute it represents; see “Connecting kernels to vertex buffers” on page 12.	
`output` (kernel contains exactly one)	`output float4` *`name`*`;`
The output value is the position of the vertex in clip space `(`*`x,y,z,w`*`)`. It must be present, and must be of type `float4`.	

Parameter metadata

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

This ID keyword is for parameter metadata values:

`PB3D_CLIP_TRANSFORM`	`float4x4`	The transform from the position relative to the mesh into clip space. Not indexed.

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

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

The names are strings. Parameter metadata values must be one of these data types:

```
int, int2, int3, int4
float, float2, float3, float4
float2x2, float3x3, float4x4
bool, bool2, bool3, bool4
string
```

For `int`, `float`, and `bool`, the type is deduced automatically. For other types, specify a constant of the correct type (such as `float2( 1.0, -1.0 )`), or a string delimited by double quotes. For example:

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


Value constraint elements

These parameter metadata values specify constraints on the parameter value:

`minValue`	The minimum allowed value.
`maxValue`	The maximum allowed value.
`defaultValue`	The default value.

Descriptive elements

These parameter metadata values provide display values, to be used by the client application when presenting the parameter in the UI :

`description`	A descriptive string that a client application can use in any way.

Vertex kernel function definitions

The kernel definition must contain an `evaluateVertex()` function. Other support functions are optional.

`evaluateVertex()`

```
void evaluateVertex()
{
    statements
}
```

Defines the position processing to be performed at each vertex of the mesh. The function must set the `output` value to the position of the vertex in clip space. It can optionally set `output vertex` values. .

This function and all functions that it calls have:

- read-only access to all parameters
- read-only access to all input vertex values
- write access and read-after-write access to all output vertex values
- write access and read-after-write access to the output value

`other functions`

```
returnType name(args)
{
    statements
}
```

You can define zero or more additional kernel functions.

The argument syntax is:

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

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

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

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

Name functions according to the usual C conventions. All functions names that start with underscore (_) are reserved and cannot be used.

All functions are overloaded; that is, matched by argument types as well as names. Unlike C++, no implicit type conversion is performed when matching overloaded functions. All functions must be defined before calling; there are no forward declarations. Pixel Bender 3D does not support recursive function calls.

Statements in kernel functions

The following flow-control constructs are supported in Pixel Bender 3D, with the usual C syntax:

```
if (scalar_expression) true_statement
if (scalar_expression) true_statement else false_statement
for (initializer; condition; incremental) statement
return expression;
```

Pixel Bender 3D does not support `return` statements inside the body of a conditional statement or loop. `For` loops must have compile time constant lower and upper bounds. For example:

```
float3 sampleTotal = sample( src, coords, PB3D_NEAREST );

for( int i = 1; i < 10; ++i )
{
    sampleTotal += sample( src, coords + float2( i, 0.0 ), PB3D_NEAREST );
    sampleTotal += sample( src, coords - float2( i, 0.0 ), PB3D_NEAREST );
}
```

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

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

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

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

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

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

Vertex kernel examples

A vertex kernel must produce the new position of the input vertex in clip space.

Example 1: Simplest vertex kernel

In this example, notice how the use of metadata on `objectToClipSpaceTransform` and `vertexPosition` allows the ActionScript code to hook up the correct values from the mesh.

```
<languageVersion : 1.0;>

vertex kernel SimplestVertexKernel

<
    namespace : "AIF Test";
    vendor : "Adobe";
    version : 1;
>

{
    parameter float4x4 objectToClipSpaceTransform
    <
        id : "PB3D_CLIP_TRANSFORM";
    >;

    input vertex float3 vertexPosition
    <
        id : "PB3D_POSITION";
    >;

    output float4 vertexClipPosition;

   void evaluateVertex()
   {
   vertexClipPosition = float4(
   vertexPosition.x, vertexPosition.y,
   vertexPosition.z, 1.0 ) * objectToClipSpaceTransform;
    }
}
```

Example 2: Changing the vertex position

Suppose we have an animated Popeye image, and we want his muscles to bulge after he's eaten spinach. For simplicity, this kernel translates each vertex along its vertex normal. It calculates the vertex clip position using the new vertex position.

Notice that the metadata for the new vertex position (the `output vertex` value) is the same as for the original vertex position (the `input vertex` value). This tells us that the new vertex position is intended to substitute for the original vertex position; that is, if anything in the material kernel uses the vertex position, it will get the new position.

```
<languageVersion : 1.0;>

vertex kernel Bulge
```

```
<
    namespace : "AIF Test";
    vendor : "Adobe";
    version : 1;
>
{
    parameter float bulgeFactor;
    parameter float4x4 objectToClipSpaceTransform
    <
        id : "PB3D_CLIP_TRANSFORM";
    >;

    input vertex float3 vertexPosition
    <
        id : "PB3D_POSITION";
    >;

    input vertex float3 vertexNormal
    <
        id : "PB3D_NORMAL";
    >;

    output vertex float3 newVertexPosition
    <
        id : "PB3D_POSITION";
    >;

    output float4 vertexClipPosition;

   void evaluateVertex()
   {
      newVertexPosition = vertexPosition + vertexNormal * bulgeFactor;
      vertexClipPosition = float4(
         newVertexPosition.x, newVertexPosition.y,
         newVertexPosition.z, 1.0 )  * objectToClipSpaceTransform;
   }
}
```


Material kernel member syntax

The material kernel defines the appearance of vertices, and then defines the appearance of fragments on a surface. The material kernel must contain at least an `evaluateFragment()` function definition and a single `output` value; all other members are optional:

```
{
    [declarations]

   output float4 fragmentColor;

    [support functions]

   void evaluateFragment()
   {
      statements
   }
}
```

The kernel must set the single `output` value to the color of the fragment. Declarations can also include parameters, constants, input vertex, and interpolated values.

The `evaluateVertex()` function (if present) is applied to each vertex of the input mesh and computes the appearance of that vertex. The `evaluateFragment()` function is applied to each fragment of each surface of the mesh and computes the color of that fragment.

Material kernel declarations

Before the function definitions, a material kernel definition can contain these declarational members:

Declaration	Syntax
`parameter` (kernel contains zero or more)	`parameter` *`type name`* `<` *`name1 : value1;`* *`name2 : value2;`* `...` `>;`
Parameters are set before a kernel is executed and are read-only within kernel functions. Parameters can be of any type except `image`. Arrays used as parameters must have sizes determined at compile time. A parameter can have optional metadata attached to it, as one or more name-value pairs enclosed in angle brackets. See "Parameter metadata" on page 16.	
`const` (kernel contains zero or more)	`const` *`type name=compile-time_expression;`*
The value of the constant is determined at compile time. The use of `const` is recommended for constant definitions, rather than `#define`.	
`input vertex` (kernel contains zero or more)	`input vertex` *`type name`* `<` `id :` *`metadata string`*`;` `[index :` *`int`*`];` `>;`
A variable connected to an attribute of a vertex, such as its position, normal, or color. Each variable must have metadata attached to it to indicate which attribute it represents; see "Connecting kernels to vertex buffers" on page 12.	
`input image` (kernel contains zero or more)	`input` *`type name;`*
An image to use as input to the `evaluateFragment()` function. The type must be `image1`, `image2`, `image3`, or `image4`.	
`output` (kernel contains exactly one)	`output` *`type name;`*
The output value is the color of the fragment. It must be present, and must be of type `float3` or `float4`.	

Material kernel function definitions

The kernel definition must contain an `evaluateFragment()` function. Other support functions are optional. The *`statements`* syntax for all functions is the same as for vertex kernels.

`evaluateVertex()`

```
void evaluateVertex()
{
    statements
}
```

Defines the appearance processing to be performed at each vertex of the mesh. This function must set *interpolated* values to have any effect. This function and all functions that it calls have:

- read-only access to all parameters
- read-only access to all input vertex values
- write access and read-after-write access to all interpolated values

---

`evaluateFragment()`

```
void evaluateFragment()
{
    statements
}
```

Defines the processing to be performed, in parallel, for each fragment of each surface of the mesh. The function must set all channels of the output value. This function and all functions that it calls have:

- read-only access to all parameters
- read-only access to all interpolated values
- read-only access to all input images
- write access and read-after-write access to the output value

---

`other functions`

```
returnType name(args)
{
    statements
}
```

You can define zero or more additional kernel functions, using the same syntax as for vertex kernels.

Material kernel example

The ultimate output from a material kernel is the color of a single fragment; the `evaluateFragment()` function must write to an `output` value.

In this simple material kernel, an `interpolatedValue` is set for each vertex. The `evaluateFragment()` function reads this value and interpolates it correctly for the position of the fragment.

This material kernel can be combined with any of the vertex kernels examples.

```
<languageVersion : 1.0;>

material kernel SimpleMaterialKernel
```

```
    <
        namespace : "AIF Test";
        vendor : "Adobe";
         version : 1;
    >
{
    input vertex float3 vertexPosition
    <
        id : "PB3D_POSITION";
    >;

    input vertex float3 vertexColor
    <
        id : "PB3D_COLOR";
    >;

    interpolated float3 interpolatedColor;

    output float4 fragmentColor;

    void evaluateVertex()
    {
        interpolatedColor = vertexColor;

        // Tweak the red channel depending on the Z coordinate
        if( vertexPosition.z > 4.0 )
            interpolatedColor.r /= 2.0;
    }

    void evaluateFragment()
    {
        fragmentColor.rgb = interpolatedColor;
        fragmentColor.a = 1.0;
    }
}
```

3 Pixel Bender 3D Data Types

Pixel Bender 3D is strongly typed. There are no automatic conversions between types, with the single exception of integral promotion during construction of floating-point vector and matrix types. There are several classes of types, each defined over a particular set of operators and intrinsic functions.

Scalar types

Pixel Bender 3D supports these basic numeric types:

`bool`	A Boolean value
`int`	An integer value
`float`	A floating-point value

All of these numeric types can participate in arithmetic operations. All the usual arithmetic operators are defined over the scalar types; see “Operators” on page 28.

Conversions between scalar types

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

```
type(expression)
```

For example:

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

The `pixel1` type can be used interchangeably with `float`.

Conversions to and from `bool` have these results:

`bool` -> `int`	`false` -> 0 `true` -> 1
`bool` -> `float`	`false` -> 0.0 `true` -> 1.0
`int` -> `bool`	0 -> `false` everything else `->` `true`
`float` -> `bool`	0.0 -> false everything else `->` `true`

Vector types

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

```
float2    bool2  int2
float3    bool3  int3
float4    bool4  int4
```

Initialize any of the vector types using this constructor syntax:

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

For example:

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

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

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

Most scalar arithmetic operators are defined over vectors as operating component-wise; see "Operators" on page 28.

You can access a vector element by index or names.

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

  ```
  vectorValue[index]
  ```

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

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

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

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

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

Selecting and reordering elements

Pixel Bender 3D allows "swizzling" to select and re-order vector elements. For a vector value with `n` elements, up to `n` named indices can be specified following the dot operator. The corresponding elements of the vector value are concatenated to form a new vector result with as many elements as index specifiers. This syntax can be used to re-order, remove, or repeat elements; for example:

```
float4 vec4;
```

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

Indices from separate sequences cannot be combined:

```
float4 vec4;

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

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

```
float3 vec3;
float2 vec2;

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

Interactions

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

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

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

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

A naive expansion of this would look like this:

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

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

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

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

Conversions between vector types

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

```
type(expression)
```

For example:

```
float3 fvec3;
int3 ivec3;

fvec3=float3(ivec3);
```

Matrix types

These matrix types are available:

```
float2x2
float3x3
float4x4
```

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

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

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

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

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

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

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

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

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

```
matrix[ column ]
```

A small set of operators is defined for matrices, which perform component-wise, matrix/matrix, and matrix/vector operations. See “Operators” on page 28.

Other types

Image types

Pixel Bender 3D supports images of up to four channels.

```
image1
image2
image3
image4
```

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

Array types

Fixed length, one dimensional arrays of any type are available as parameters only.

```
parameter float4x4 boneTransforms[ 10 ];
```

There is no built-in limit on array length; hardware,however, has its own limits.

Operators

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

`.`	Member selection
`++ --`	Postfix increment or decrement
`++ --`	Prefix increment or decrement
`- !`	Unary negation, logical not
`* /`	Multiply, divide
`+ -`	Add, subtract
`< > <= >=`	Relational
`== !=`	Equality
`&&`	Logical and
`^^`	Logical exclusive or
`||`	Logical inclusive or
`= += -= *= /=`	Assignment
`?:`	Selection

Short-circuit evaluation for logical AND, and logical inclusive OR is undefined. If you require short-circuit evaluation to be present (or absent), you must explicitly code it.

Operations on multiple-value types

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

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

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

This operation is equivalent to:

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

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

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

This operation is equivalent to:

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

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

`float2x2 * float2x2` `float3x3 * float3x3` `float4x4 * float4x4`	Linear-algebraic matrix multiplication
`float2x2 * float2` `float3x3 * float3` `float4x4 * float4`	Column-vector multiplication
`float2 * float2x2` `float3 * float3x3` `float4 * float4x4`	Row-vector multiplication

Operand and result types

These tables show all of the combinations of types that can be operated on by each of the operators, and the resulting type of each operation.

Operator: +

operand types	result type	operand types	result type
float + float	float	float + float4	float4
float2 + float	float2	float4 + float4	float4
float3 + float	float3	int + int2	int2
float4 + float	float4	int2 + int2	int2
float2x2 + float	float2x2	int + int3	int3
float3x3 + float	float3x3	int3 + int3	int3
float4x4 + float	float4x4	int + int4	int4
int + int	int	int4 + int4	int4

operand types	result type	operand types	result type
int2 + int	int2	float + float2x2	float2x2
int3 + int	int3	float2x2 + float2x2	float2x2
int4 + int	int4	float + float3x3	float3x3
float + float2	float2	float3x3 + float3x3	float3x3
float2 + float2	float2	float + float4x4	float4x4
float + float3	float3	float4x4 + float4x4	float4x4
float3 + float3	float3		

Operator: -

operand types	result type	operand types	result type
float - float	float	float - float4	float4
float2 - float	float2	float4 - float4	float4
float3 - float	float3	int - int2	int2
float4 - float	float4	int2 - int2	int2
float2x2 - float	float2x2	int - int3	int3
float3x3 - float	float3x3	int3 - int3	int3
float4x4 - float	float4x4	int - int4	int4
int - int	int	int4 - int4	int4
int2 - int	int2	float - float2x2	float2x2
int3 - int	int3	float2x2 - float2x2	float2x2
int4 - int	int4	float - float3x3	float3x3
float - float2	float2	float3x3 - float3x3	float3x3
float2 - float2	float2	float - float4x4	float4x4
float - float3	float3	float4x4 - float4x4	float4x4
float3 - float3	float3		

Operator: *

operand types	result type	operand types	result type
float * float	float	float * float3x3	float3x3
float2 * float	float2	float3 * float3x3	float3
float3 * float	float3	float3x3 * float3x3	float3x3
float4 * float	float4	float * float4x4	float4x4

operand types	result type	operand types	result type
float2x2 * float	float2x2	float4 * float4x4	float4
float3x3 * float	float3x3	float4x4 * float4x4	float4x4
float4x4 * float	float4x4	int * int	int
float * float2	float2	int2 * int	int2
float2 * float2	float2	int3 * int	int3
float2x2 * float2	float2	int4 * int	int4
float * float3	float3	int * int2	int2
float3 * float3	float3	int2 * int2	int2
float3x3 * float3	float3	int * int3	int3
float * float4	float4	int3 * int3	int3
float4 * float4	float4	int * int4	int4
float4x4 * float4	float4	int4 * int4	int4
float * float2x2	float2x2		
float2 * float2x2	float2		
float2x2 * float2x2	float2x2		

Operator: /

Division by 0 is undefined for `int` and `float` types.

operand types	result type	operand types	result type
float / float	float	float / float4	float4
float2 / float	float2	float4 / float4	float4
float3 / float	float3	int / int2	int2
float4 / float	float4	int2 / int2	int2
float2x2 / float	float2x2	int / int3	int3
float3x3 / float	float3x3	int3 / int3	int3
float4x4 / float	float4x4	int / int4	int4
int / int	int	int4 / int4	int4
int2 / int	int2	float / float2x2	float2x2
int3 / int	int3	float2x2 / float2x2	float2x2
int4 / int	int4	float / float3x3	float3x3
float / float2	float2	float3x3 / float3x3	float3x3
float2 / float2	float2	float / float4x4	float4x4

operand types	result type	operand types	result type
`float / float3`	`float3`	`float4x4 / float4x4`	`float4x4`
`float3 / float3`	`float3`		

Unary operators: +, -

operand types	result type	operand types	result type
`+ float`	`float`	`- float`	`float`
`+ int`	`int`	`- int`	`int`
`+ float2`	`float2`	`- float2`	`float2`
`+ float3`	`float3`	`- float3`	`float3`
`+ float4`	`float4`	`- float4`	`float4`
`+ int2`	`int2`	`- int2`	`int2`
`+ int3`	`int3`	`- int3`	`int3`
`+ int4`	`int4`	`- int4`	`int4`
`+ float2x2`	`float2x2`	`- float2x2`	`float2x2`
`+ float3x3`	`float3x3`	`- float3x3`	`float3x3`
`+ float4x4`	`float4x4`	`- float4x4`	`float4x4`

Unary operators: ++, --

operand types	result type	operand types	result type
`++ float`	`float`	`-- float`	`float`
`++ int`	`int`	`-- int`	`int`
`++ float2`	`float2`	`-- float2`	`float2`
`++ float3`	`float3`	`-- float3`	`float3`
`++ float4`	`float4`	`-- float4`	`float4`
`++ int2`	`int2`	`-- int2`	`int2`
`++ int3`	`int3`	`-- int3`	`int3`
`++ int4`	`int4`	`-- int4`	`int4`
`++ float2x2`	`float2x2`	`-- float2x2`	`float2x2`
`++ float3x3`	`float3x3`	`-- float3x3`	`float3x3`
`++ float4x4`	`float4x4`	`-- float4x4`	`float4x4`
`float ++`	`float`	`float --`	`float`
`int ++`	`int`	`int --`	`int`

operand types	result type	operand types	result type
float2 ++	float2	float2 --	float2
float3 ++	float3	float3 --	float3
float4 ++	float4	float4 --	float4
int2 ++	int2	int2 --	int2
int3 ++	int3	int3 --	int3
int4 ++	int4	int4 --	int4
float2x2 ++	float2x2	float2x2 --	float2x2
float3x3 ++	float3x3	float3x3 --	float3x3
float4x4 ++	float4x4	float4x4 --	float4x4

Assignment operators: +=, -=

operand types	result type	operand types	result type
float += float	float	float -= float	float
float2 += float	float2	float2 -= float	float2
float3 += float	float3	float3 -= float	float3
float4 += float	float4	float4 -= float	float4
float2x2 += float	float2x2	float2x2 -= float	float2x2
float3x3 += float	float3x3	float3x3 -= float	float3x3
float4x4 += float	float4x4	float4x4 -= float	float4x4
int += int	int	int -= int	int
int2 += int	int2	int2 -= int	int2
int3 += int	int3	int3 -= int	int3
int4 += int	int4	int4 -= int	int4
float2 += float2	float2	float2 -= float2	float2
float3 += float3	float3	float3 -= float3	float3
float4 += float4	float4	float4 -= float4	float4
int2 += int2	int2	int2 -= int2	int2
int3 += int3	int3	int3 -= int3	int3
int4 += int4	int4	int4 -= int4	int4
float2x2 += float2x2	float2x2	float2x2 -= float2x2	float2x2
float3x3 += float3x3	float3x3	float3x3 -= float3x3	float3x3
float4x4 += float4x4	float4x4	float4x4 -= float4x4	float4x4

Assignment operators: *=, /=

operand types	result type	operand types	result type
float *= float	float	float /= float	float
float2 *= float	float2	float2 /= float	float2
float3 *= float	float3	float3 /= float	float3
float4 *= float	float4	float4 /= float	float4
float2x2 *= float	float2x2	float2x2 /= float	float2x2
float3x3 *= float	float3x3	float3x3 /= float	float3x3
float4x4 *= float	float4x4	float4x4 /= float	float4x4
int *= int	int	int /= int	int
int2 *= int	int2	int2 /= int	int2
int3 *= int	int3	int3 /= int	int3
int4 *= int	int4	int4 /= int	int4
float2 *= float2	float2	float2 /= float2	float2
float3 *= float3	float3	float3 /= float3	float3
float4 *= float4	float4	float4 /= float4	float4
int2 *= int2	int2	int2 /= int2	int2
int3 *= int3	int3	int3 /= int3	int3
int4 *= int4	int4	int4 /= int4	int4
float2 *= float2x2	float2	float2x2 /= float2x2	float2x2
float2x2 *= float2x2	float2x2	float3x3 /= float3x3	float3x3
float3 *= float3x3	float3	float4x4 /= float4x4	float4x4
float3x3 *= float3x3	float3x3		
float4 *= float4x4	float4		
float4x4 *= float4x4	float4x4		

Assignment operator: =

operand types	result type	operand types	result type
float = float	float	bool = bool	bool
float2 = float2	float2	bool2 = bool2	bool2
float3 = float3	float3	bool3 = bool3	bool3
float4 = float4	float4	bool4 = bool4	bool4

operand types	result type	operand types	result type
float2x2 = float2x2	float2x2	region = region	region
float3x3 = float3x3	float3x3	image1 = image1	image1
float4x4 = float4x4	float4x4	image2 = image2	image2
int = int	int	image3 = image3	image3
int2 = int2	int2	image4 = image4	image4
int3 = int3	int3		
int4 = int4	int4		

Logical operators: &&, ||, ^^, !

operand types	result type
bool && bool	bool
bool || bool	bool
bool ^^ bool	bool
! bool	bool

Relational operators: <, >, <=, >=

operand types	result type	operand types	result type
float < float	bool	float <= float	bool
int < int	bool	int <= int	bool
float > float	bool	float >= float	bool
int > int	bool	int >= int	bool

Equality operators: ==, !=

operand types	result type	operand types	result type
float == float	bool	float != float	bool
int == int	bool	int != int	bool
bool == bool	bool	bool != bool	bool
float2 == float2	bool	float2 != float2	bool
float3 == float3	bool	float3 != float3	bool
float4 == float4	bool	float4 != float4	bool
int2 == int2	bool	int2 != int2	bool

operand types	result type	operand types	result type
int3 == int3	bool	int3 != int3	bool
int4 == int4	bool	int4 != int4	bool
bool2 == bool2	bool	bool2 != bool2	bool
bool3 == bool3	bool	bool3 != bool3	bool
bool4 == bool4	bool	bool4 != bool4	bool
float2x2 == float2x2	bool	float2x2 != float2x2	bool
float3x3 == float3x3	bool	float3x3 != float3x3	bool
float4x4 == float4x4	bool	float4x4 != float4x4	bool
image1 == image1	bool	image1 != image1	bool
image2 == image2	bool	image2 != image2	bool
image3 == image3	bool	image3 != image3	bool
image4 == image4	bool	image4 != image4	bool

Selection operator: ? :

operand types	result type	operand types	result type
bool ? float : float	float	bool ? bool : bool	bool
bool ? float2 : float2	float2	bool ? bool2 : bool2	bool2
bool ? float3 : float3	float3	bool ? bool3 : bool3	bool3
bool ? float4 : float4	float4	bool ? bool4 : bool4	bool4
bool ? float2x2 : float2x2	float2x2	bool ? image1 : image1	image1
bool ? float3x3 : float3x3	float3x3	bool ? image2 : image2	image2
bool ? float4x4 : float4x4	float4x4	bool ? image3 : image3	image3
bool ? int : int	int	bool ? image4 : image4	image4
bool ? int2 : int2	int2		
bool ? int3 : int3	int3		
bool ? int4 : int4	int4		

Array, vector, and matrix access

When the index value is out of range, the result of `[i]` is undefined.

operand types	result type	operand types	result type
`float2 [ int ]`	`float`	`bool2 [ int ]`	`bool`
`float3 [ int ]`	`float`	`bool3 [ int ]`	`bool`
`float4 [ int ]`	`float`	`bool4 [ int ]`	`bool`
`int2 [ int ]`	`int`	`float2x2 [ int ]`	`float2`
`int3 [ int ]`	`int`	`float3x3 [ int ]`	`float3`
`int4 [ int ]`	`int`	`float4x4 [ int ]`	`float4`
`float_array [ int ]`	`float`		

4 Pixel Bender 3D Built-in Functions

Pixel Bender 3D supports a variety of built-in functions over different data types.

Mathematical functions

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

`float radians( float degrees )` `float2 radians( float2 degrees )` `float3 radians( float3 degrees )` `float4 radians( float4 degrees )`	Converts degrees to radians.
`float degrees( float radians )` `float2 degrees ( float2 radians )` `float3 degrees ( float3 radians )` `float4 degrees ( float4 radians )`	Converts radians to degrees.
`float sin( float radians )` `float2 sin( float2 radians )` `float3 sin( float3 radians )` `float4 sin( float4 radians )`	Returns the sine of the input.
`float cos( float radians )` `float2 cos( float2 radians )` `float3 cos( float3 radians )` `float4 cos( float4 radians )`	Returns the cosine of the input.
`float tan( float radians )` `float2 tan( float2 radians )` `float3 tan( float3 radians )` `float4 tan( float4 radians )`	Returns the tangent of the input. Undefined if `cos(radians)==0`.
`float asin( float x )` `float2 asin( float2 x )` `float3 asin( float3 x )` `float4 asin( float4 x )`	Returns the arc sine of the input. The result is in the range `[-pi/2..pi/2]`. Undefined if `x<-1` or `1<x`.
`float acos( float x )` `float2 acos( float2 x )` `float3 acos( float3 x )` `float4 acos( float4 x )`	Returns the arc cosine of the input. The result is in the range `[0..pi]`. Undefined if `x<-1` or `1<x`.
`float atan( float y_over_x )` `float2 atan( float2 y_over_x)` `float3 atan( float3 y_over_x)` `float4 atan( float4 y_over_x)`	Returns the arc tangent of the input. The result is in the range `[-pi/2..pi/2]`.

`float atan( float y, float x )` `float2 atan( float2 y, float2 x )` `float3 atan( float3 y, float3 x )` `float4 atan( float4 y, float4 x )`	Returns the arc tangent of `y/x`. The result will be in the range `[-pi..pi]`. Undefined if `x==0` and `y==0`.
`float pow( float x, float y )` `float2 pow ( float2 x, float2 y )` `float3 pow ( float3 x, float3 y )` `float4 pow ( float4 x, float4 y )`	Returns x^y. Undefined if `x < 0`.
`float exp( float x )` `float2 exp( float2 x )` `float3 exp( float3 x )` `float4 exp( float4 x )`	Returns e^x.
`float exp2( float x )` `float2 exp2( float2 x )` `float3 exp2( float3 x )` `float4 exp2( float4 x )`	Returns 2^x.
`float log( float x )` `float2 log( float2 x )` `float3 log( float3 x )` `float4 log( float4 x )`	Returns the natural logarithm of `x`. Undefined if `x<=0`.
`float log2( float x )` `float2 log2( float2 x )` `float3 log2( float3 x )` `float4 log2( float4 x )`	Returns the base-2 logarithm of `x`. Undefined if `x<=0`.
`float sqrt( float x )` `float2 sqrt( float2 x )` `float3 sqrt( float3 x )` `float4 sqrt( float4 x )`	Returns the positive square root of `x`. Undefined if `x < 0`.
`float inverseSqrt( float x )` `float2 inverseSqrt( float2 x )` `float3 inverseSqrt( float3 x )` `float4 inverseSqrt( float4 x )`	Returns the reciprocal of the positive square root of `x`. Undefined if `x<=0`.
`float abs( float x )` `float2 abs( float2 x )` `float3 abs( float3 x )` `float4 abs( float4 x )`	If `x >= 0`, returns `x`, otherwise returns `-x`.
`float sign( float x )` `float2 sign( float2 x )` `float3 sign( float3 x )` `float4 sign( float4 x )`	If `x < 0`, returns -1 If `x == 0`, returns 0 If `x > 0`, returns 1
`float floor( float x )` `float2 floor( float2 x )` `float3 floor( float3 x )` `float4 floor( float4 x )`	Returns `y`, the nearest integral value where `y <= x`.
`float ceil( float x )` `float2 ceil( float2 x )` `float3 ceil( float3 x )` `float4 ceil( float4 x )`	Returns `y`, the nearest integral value where `y >= x`.

`float fract( float x )` `float2 fract( float2 x )` `float3 fract( float3 x )` `float4 fract( float4 x )`	Returns `x - floor(x)`.
`float mod( float x, float y )` `float2 mod( float2 x, float2 y )` `float3 mod( float3 x, float3 y )` `float4 mod( float4 x, float4 y )` `float2 mod( float2 x, float y )` `float3 mod( float3 x, float y )` `float4 mod( float4 x, float y )`	Returns `x - y * floor(x/y)`. Undefined if `y==0`.
`float min( float x, float y )` `float2 min( float2 x, float2 y )` `float3 min( float3 x, float3 y )` `float4 min( float4 x, float4 y )` `float2 min( float2 x, float y )` `float3 min( float3 x, float y )` `float4 min( float4 x, float y )`	If `x < y`, returns `x`, otherwise returns `y`.
`float max( float x, float y )` `float2 max( float2 x, float2 y )` `float3 max( float3 x, float3 y )` `float4 max( float4 x, float4 y )` `float2 max( float2 x, float y )` `float3 max( float3 x, float y )` `float4 max( float4 x, float y )`	If `x > y`, returns `x`, otherwise returns `y`.
`float step( float x, float y )` `float2 step( float2 x, float2 y )` `float3 step( float3 x, float3 y )` `float4 step( float4 x, float4 y )` `float2 step( float x, float2 y )` `float3 step( float x, float3 y )` `float4 step( float x, float4 y )`	If `y < x`, returns 0.0, otherwise returns 1.0
`float clamp(float x, float minval, float maxval)` `float2 clamp(float2 x, float2 minval, float2 maxval)` `float3 clamp(float3 x, float3 minval, float3 maxval)` `float4 clamp(float4 x, float4 minval, float4 maxval)` `float2 clamp( float2 x, float minval, float maxval )` `float3 clamp( float3 x, float minval, float maxval )` `float4 clamp( float4 x, float minval, float maxval )`	If `x<minval`, returns `minval` If `x>maxval`, returns `maxval` otherwise returns `x`.
`float mix(float x, float y, float a)` `float2 mix(float2 x, float2 y, float2 a)` `float3 mix(float3 x, float3 y, float3 a)` `float4 mix(float4 x, float4 y, float4 a)` `float2 mix( float2 x, float2 y, float a )` `float3 mix( float3 x, float3 y, float a )` `float4 mix( float4 x, float4 y, float a )`	Returns `x * (1.0 - a) +y * a` (that is, a linear interpolation between `x` and `y`).
`float smoothStep(float edge0, float edge1, float x)` `float2 smoothStep(float2 edge0, float2 edge1, float2 x)` `float3 smoothStep(float3 edge0, float3 edge1, float3 x)` `float4 smoothStep(float4 edge0, float4 edge1, float4 x)` `float2 smoothStep( float edge0, float edge1, float2 x )` `float3 smoothStep( float edge0, float edge1, float3 x )` `float4 smoothStep( float edge0, float edge1, float4 x )`	If `x <= edge0`, returns 0 If `x >= edge1`, returns 1, otherwise performs smooth hermite interpolation.

Geometric functions

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

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

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

`float2x2 matrixCompMult(float2x2 x, float2x2 y)` `float3x3 matrixCompMult(float3x3 x, float3x3 y)` `float4x4 matrixCompMult(float4x4 x, float4x4 y)`	Returns the component-wise product of `x` and `y`.

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

`bool2 lessThan(int2 x, int2 y)` `bool3 lessThan(int3 x, int3 y)` `bool4 lessThan(int4 x, int4 y)` `bool2 lessThan(float2 x, float2 y)` `bool3 lessThan(float3 x, float3 y)` `bool4 lessThan(float4 x, float4 y)`	Returns the component-wise compare of `x < y`.
`bool2 lessThanEqual(int2 x, int2 y)` `bool3 lessThanEqual(int3 x, int3 y)` `bool4 lessThanEqual(int4 x, int4 y)` `bool2 lessThanEqual(float2 x, float2 y)` `bool3 lessThanEqual(float3 x, float3 y)` `bool4 lessThanEqual(float4 x, float4 y`	Returns the component-wise compare of `x <= y`.

`bool2 greaterThan(int2 x, int2 y)` `bool3 greaterThan(int3 x, int3 y)` `bool4 greaterThan(int4 x, int4 y)` `bool2 greaterThan(float2 x, float2 y)` `bool3 greaterThan(float3 x, float3 y)` `bool4 greaterThan(float4 x, float4 y)`	Returns the component-wise compare of `x > y`.
`bool2 greaterThanEqual(int2 x, int2 y)` `bool3 greaterThanEqual(int3 x, int3 y)` `bool4 greaterThanEqual(int4 x, int4 y)` `bool2 greaterThanEqual(float2 x, float2 y)` `bool3 greaterThanEqual(float3 x, float3 y)` `bool4 greaterThanEqual(float4 x, float4 y)`	Returns the component-wise compare of `x >= y`.
`bool2 equal(int2 x, int2 y)` `bool3 equal(int3 x, int3 y)` `bool4 equal(int4 x, int4 y)` `bool2 equal(float2 x, float2 y)` `bool3 equal(float3 x, float3 y)` `bool4 equal(float4 x, float4 y)` `bool2 equal(bool2 x, bool2 y)` `bool3 equal(bool3 x, bool3 y)` `bool4 equal(bool4 x, bool4 y)`	Returns the component-wise compare of `x == y`.
`bool2 notEqual(int2 x, int2 y)` `bool3 notEqual(int3 x, int3 y)` `bool4 notEqual(int4 x, int4 y)` `bool2 notEqual(float2 x, float2 y)` `bool3 notEqual(float3 x, float3 y)` `bool4 notEqual(float4 x, float4 y)` `bool2 notEqual(bool2 x, bool2 y)` `bool3 notEqual(bool3 x, bool3 y)` `bool4 notEqual(bool4 x, bool4 y)`	Returns the component-wise compare of `x != y`.

These vector functions operate only on vectors of Boolean type:

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

Sampling functions

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

```
float sample( image1 im, float2 v, sample_options)
float2 sample( image2 im, float2 v, sample_options )
float3 sample( image3 im, float2 v, sample_options )
float4 sample( image4 im, float2 v, sample_options )
```

Combine option constants with a logical OR; for example:

```
float3 color = sample( image, coords,
   PB3D_NEAREST | PB3D_MIPDISABLE | PB3D_CLAMP | PB3D_2D );
```

The sampling option constants are as follows; the default option within each category is bold:

Filtering	**PB3D_NEAREST** PB3D_LINEAR
Mipmapping	**PB3D_MIPDISABLE** PB3D_MIPLINEAR PB3D_MIPNEAREST
Repeat	**PB3D_CLAMP** PB3D_REPEAT
Dimension	**PB3D_2D** PB3D_CUBE

Intrinsic functions

Pixel Bender includes this function that allows access to the system's compile-time or run-time properties.

`int` *`arrayVariable`*`.length()`	Returns the number of elements of an array.