Smooth Shading

Adobe® Developers Association

10 October 1997

Technical Note #5600

LanguageLevel 3

Adobe Systems Incorporated

Corporate Headquarters
345 Park Avenue
San Jose, CA 95110-2704
(408) 536-6000 Main Number

Eastern Regional Office
24 New England
Executive Park
Burlington, MA 01803
(617) 273-2120

Adobe Systems Europe Limited
Adobe House, Mid New Cultins
Edinburgh EH11 4DU
Scotland, United Kingdom
+44-131-453-2211

Adobe Systems Japan
Yebisu Garden Place Tower
4-20-3 Ebisu, Shibuya-ku
Tokyo 150 Japan
+81-3-5423-8100

PN LPS5600

Adobe Systems Incorporated

Contents

Adobe Systems Incorporated

Figures

Adobe Systems Incorporated

Tables

Adobe Systems Incorporated

Examples

Adobe Systems Incorporated

Preface

This Document

This document provides a detailed description of smooth shading, a LanguageLevel 3 feature of Adobe® PostScript® that enables a developer to add higher-quality monochrome or color gradient fills to an application.

Intended Audience

This document is written for software developers who are interested in learning about smooth shading or adding smooth shading capabilities to an application that supports PostScript display or printing devices. It is assumed that the developer has an adequate background in mathematics. This knowledge will help in the understanding of the complex formulae and functions used to describe the implementation of the shading methods.

Organization of This Document

Section 1, “Smooth Shading,” provides an overview of the LanguageLevel 3 feature and all of its parts. A comparison of current and previous methods of shading is made. The uses for, and benefits of, this feature in a PostScript language environment are also covered.

Section 2, “Implementing Smooth Shading,” defines the PostScript language extensions for smooth shading. Each shading method and the underlying mathematical elements supporting the method are described in detail. Several examples defining dictionary parameters (keys) are included as well as several workable code samples for each of the supported shading methods. The examples shown include use of the **Pattern** and **Function** dictionaries, and the **shfill, makepattern**, and **setpattern** operators.

Section 3, “Smooth Shading Tips,” gives helpful information on using smooth shading and functions and selecting the best smooth shading method for specific application needs.

Related Publications

Supplement: PostScript Language Reference Manual (LanguageLevel 3 Specification and Adobe PostScript 3™ Version 3010 Product Supplement), available from the Adobe Developers Association, describes the formal extensions to the PostScript language that have occurred since the publication of the PostScript Language Reference Manual, Second Edition. This supplement also includes all LanguageLevel 3 extensions available in version 3010.

PostScript Language Reference Manual, Second Edition (Reading, MA: Addison-Wesley, 1991) is the developer's reference manual for the PostScript language. It describes the syntax and semantics of the language, the imaging model, and the effects of the graphical operators.

Statement of Liability

THIS PUBLICATION AND THE INFORMATION HEREIN IS FURNISHED AS IS, 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, MAKES NO WARRANTIES OF ANY KIND (EXPRESS, IMPLIED, OR STATUTORY) WITH RESPECT TO THIS PUBLICATION, AND EXPRESSLY DISCLAIMS ANY AND ALL WARRANTIES OF MERCHANTABILITY, FITNESS FOR PARTICULAR PURPOSES, AND NONINFRINGMENT OF THIRD-PARTY RIGHTS.

Smooth Shading

1 Smooth Shading

1.1 Overview of Smooth Shading

Smooth shading can be used to accurately describe both monochrome and color *gradient fills* (blends) for onscreen display or for printing to a PostScript printer. A gradient fill is simply a smooth transition from one color to another color. One of the intentions of smooth shading is to separate the geometry of the area to be filled (the object) from the *geometry* of the color gradient fill or transition (the description of the colors to be used to create the gradient fill).

Smooth shading has many uses, including:

- painting oval, circular, or polygonal radial gradient fills.
- painting an object or a region with a gradient fill color.
- rendering gradient fills between objects using Bézier patch meshes.
- rendering three-dimensional objects with triangle meshes.

In previous levels of the PostScript language, a gradient fill was approximated by a large series of concentric, filled objects, known as *contours*. The geometries and solid-color fill values were interpolated between first and last objects. This method tended to be very inefficient and device-dependent, but, with enough contours, it could produce the illusion of a continuous gradient fill. The major task for developers was to determine the best number of contours for a particular gradient fill. If the number of contours chosen was too high, the resulting gradient fill would waste resources and device memory. If the number of contours was too low, the resulting output would contain banding; in other words, the gradient fill would become discontinuous in one or more places of the region to be shaded.

In LanguageLevel 3, smooth shading of objects and regions is defined in terms of a complex paint (gradient fill) that provides smooth transitions between colors across the painted area(s). The two language extensions for creating smooth shading are the **shfill** operator and the Type 2 **Pattern** dictionary.

When the object to be painted is a relatively simple shape, or when the geometry of the object to be painted with a gradient fill is the same as the geometry of the gradient fill itself, the **shfill** operator can be used. **shfill** accepts a single operand, which is a **Shading** dictionary. The **Shading** dictionary contains details of the type of shading, the geometry of the area or object to be shaded, and the geometry of the color gradient fill. In addition, the **Shading** dictionary can contain a **Function** dictionary – which is required for some types of shading and optional for others – that defines how the color or colors varies across the area or object to be shaded.

When the object to be painted is complex – such as a complex character path or an imagemask – a type 2 **Pattern** dictionary can be used. The **Pattern** dictionary has a **Shading** dictionary as one of its elements to define the shading type used. Additionally, this **Shading** dictionary may also have an associated **Function** dictionary. The type 2 **Pattern** dictionary can be used as an argument to the **setpattern** or **setcolor** operators; the resulting *color* can then be used with the **fill**, **stroke**, **show**, or **imagemask** operators to paint a path or mask, using a smooth transition between colors across the area or object to be painted. The number of steps in this transition no longer has to be specified as it was in previous levels of the PostScript language.

There are seven new shading methods that can be described with **Shading** dictionaries. They are summarized as follows:

- Function-based shading: the color of every point in the domain is defined by a mathematical or sampled function. This mathematical function does not necessarily have to be smooth or continuous. Function-based shading is defined as **ShadingType 1**.

- Axial shading: the color at any one point is created by a gradient fill along a line (an axis) between two endpoints. The gradient fill can be extended beyond the endpoints by continuing the colors at the two endpoints. Axial shading is defined as **ShadingType 2**.

- Radial shading: the color at any one point is created by a gradient fill between two circles. This shading method is often used to emulate three-dimensional spheres, cylinders, and cones. Radial shading is defined as **ShadingType 3**.

- Free-form triangle meshes using Gouraud shading: the color at any one point is created by an interpolation of the colors of the three vertices of a triangle in which it is contained using the Gouraued shading method. The triangles form a mesh that defines the area to be shaded. Free-form triangle mesh shading is defined as **ShadingType 4**.

- Lattice-form triangle meshes using Gouraud shading: this is similar to the Free-form method. The main difference is that the mesh is generated in a pseudo-rectangular lattice structure. Lattice-form triangle mesh shading is defined as **ShadingType 5**.

- Coons patch meshes: the color at any one point is created by a bilinear interpolation of the colors defining the four corner points of the patch. Each patch is defined by four Bézier curves and contains twelve control points. Coons patch mesh shading is defined as **ShadingType 6**.

- Tensor Product patch meshes: this is similar to the Coons Patch Mesh method. The main difference is that the patches are defined by 16 control points instead of 12. Tensor Product patch mesh shading is defined as **ShadingType 7**.

1.2 Benefits of Using Smooth Shading

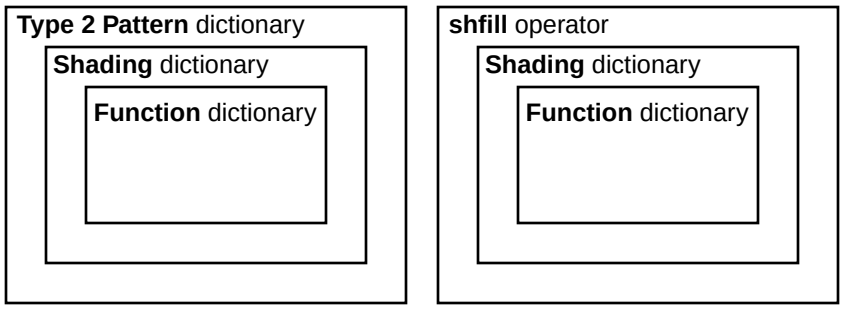
Smooth shading, and its associated LanguageLevel 3 extensions, has several benefits over older methods of providing gradient fills:

- Smooth shading specifies gradient fills in a device-independent manner. This is achieved by using mathematical functions to assign color values to each point or pixel in the region to be shaded.
- Smooth shading produces smoother gradient fills and higher quality output on high-resolution screen and printing devices.
- The same PostScript code can be used to take full advantage of the printing qualities and characteristics of every PostScript printer.
- Use of smooth shading code can significantly reduce the size of the resulting PostScript language file.
- Smooth shading code will process more quickly on LanguageLevel 3 devices than on older devices that use older techniques for shading.
- Smooth shading can be used to greatly improve gradient fills on monochrome screen and printing devices.
- Function-based, triangle, and Bézier patch shading (**ShadingType 1**, **4**, **5**, **6**, and **7**) can create gradients that interpolate along two axes. This is not possible with contours.
- The use of functions adds a concise representation of complicated gradients.
- **Pattern** dictionaries can effectively set a gradient colorspace to be used for filling paths.
- The **setsmoothness** and **currentsmoothness** operators give the developer and application the power to control the trade-off between performance and quality of smooth shading.

2 Implementing Smooth Shading

Section 2.1 describes the **shfill** operator and how it is used for certain types of shading (tiling patterns). Sections 2.2 and 2.3 cover the definition of **Pattern** dictionaries and how they are used to create smooth shading. Sections 2.4 through 2.12 cover the definition and use of **Shading** dictionaries, individual shading types, and their associated mathematical elements. Sections 2.13 through 2.17 cover the **Function** dictionaries that can be used by **Shading** dictionaries to define color transitions for smooth shading. Figure 1 shows the relationship of these three dictionary types. Finally, Section 2.18 describes the **currentsmoothness** and **setsmoothness** operators and how they might work with the various shading methods.

Figure 1 *Hierarchy of dictionaries used for smooth shading*



Complete descriptions of all the PostScript language extensions for smooth shading, plus other required language features, can be found in the *Supplement: PostScript Language Reference Manual*.

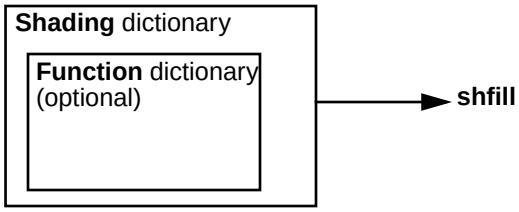
2.1 Shfill Operator

The **shfill** operator can be used to produce a smoothly shaded or varying gradient fill when the gradient fills themselves are geometric objects (where the geometry of the object to painted with a gradient fill is similar to or the same as the geometry of the gradient fill itself). It can also be used for tiling patterns containing a gradient fill. In other words, **shfill** can be used for a repeated pattern of shading. In this particular case, the **shfill** operator must be called from within the **PaintProc** procedure of a Type 1 **Pattern** dictionary.

The **shfill** operator takes as input one **Shading** dictionary, with an optional **Function** subdictionary (See Figure 2). **shfill** then paints the shape and color transitions described by the **Shading** dictionary. This paint process is limited by, or clipped to, the current clipping region. The current path is ignored by **shfill**, and no other changes are made to the current graphics state.

All of the geometric coordinates defined in the **Shading** dictionary are interpreted relative to the current user space. All color values are interpreted relative to the **ColorSpace** key of the **Shading** dictionary. The **Background** key of the **Shading** dictionary is ignored.

Figure 2 *Inputs to the shfill operator*



Note *The **shfill** operator should only be used for bounded and/or geometrically-defined shading; otherwise, the paint could occur across the entire current clipping region.*

Note *If the **currentfile** operator is used as a source for reading large blocks of data from a PostScript stream, the data should immediately follow the call to the **shfill** operator. This approach is the same as for the **image** operator.*

Note *The **shfill** operator can return the following errors: **rangecheck** and **undefinedresult**.*

Example 1 *Using shfill for smooth shading*

```
%AXSHO1.PS
%This is a simple illustration of ShadingType 2 using a
%FunctionType 2.The shading dictionary is called by shfill
%Set up color space and other graphics state variables
/inch {72 mul} def
...
%Create an object to shade
...
gsave          % save graphics state
    clip         % clip to constructed path
    newpath      % clear out current path
    % Define the shading and function dictionaries
    << /ShadingType 2
      /ColorSpace /DeviceGray
      /Coords [0 0 8.5 inch 11 inch]
      %Define the Function
      /Function << /FunctionType 2
          % Value is C0 + t ** N * (C1 - C0)
          /Domain [0 1]
          /C0 0 % result for input 0 = black
          /C1 1 % result for input 1 = white
          /N 1 % Exponent = linear
      >>
    >>
    shfill
grestore
showpage
```

Example 2 *Using shfill in a PaintProc procedure*

```
%PATTYP1.PS
%This example illustrates the use of smooth shading in
%conjunction with a Type 1 pattern to obtain a pattern
%fill whose tiles are smooth shaded areas.
%Define graphics state and other variables
/inch {72 mul} def%  define inch procedure
...
%Define a shading function dictionary
/FunctionDict 10 dict def
FunctionDict begin
    /FunctionType 2    def
    /Domain [0 1]   def
    /C0      [0 1 1] def
    /C1      [1 0 1] def
    /N       1         def
end
%Define the shading dictionary
/ShadingDict 10 dict def
ShadingDict begin
    /ShadingType 2          def
    /ColorSpace /DeviceRGB def
    /Coords [0 0 100 100] def
    /Function FunctionDict def
end
%Now define the pattern dictionary
/PatternDict 10 dict def
PatternDict begin
    /PatternType 1 def
    /PaintType 1 def
    /TilingType 1 def
    /BBox [0 0 100 100] def
    /XStep 100 def
    /YStep 100 def
    /PaintProc {
        begin
            ShadingDict shfill
        end
} def
end
PatternDict [0.25 5 sin 0 0.25 0 0] makepattern /P1 exch def
...
/Times-Bold findfont 480 scalefont setfont
0.5 inch 6 inch moveto (P) P1 setpattern show
...
showpage
```

Note *Complete PostScript language files containing these examples accompany this document.*

2.2 Pattern Dictionaries

LanguageLevel 3 includes a new **Pattern** dictionary with a **PatternType 2**. This dictionary can then be used with the **makepattern** and **setpattern** operators to define the complex paints needed for creating gradient fills (See Figure 3).

Figure 3 *Inputs to the makepattern and setpattern operators*

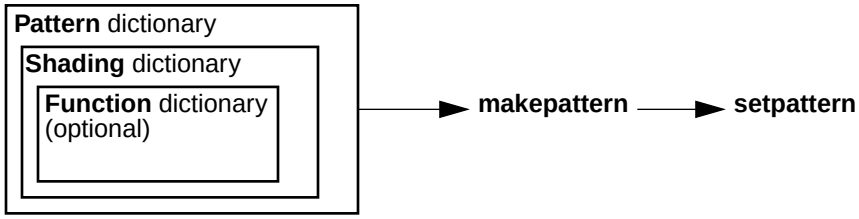


Table 1 shows the keys that define a **Pattern** dictionary. The keys are described in more detail, below.

Table 1 *Keys for the PatternType 2 Pattern dictionary*

Key	**Type**	
PatternType	integer	required
Shading	dictionary	required
XUID	array	optional
Implementation	user-defined	

The **PatternType** key is required and must have the value of 2. The **Shading** dictionary contains the information describing the desired shading method. It is described in more detail in Sections 2.4 through 2.12. The **XUID** key is an optional array that contains an extended unique ID that identifies the pattern. The **Implementation** key is defined by the **makepattern** operator. The type and value of this key are implementation-dependent.

Note *There is a new instance of the implicit resource category called **PatternType**. This new instance is 2. Currently, the only supported instances of this category type are 1 and 2.*

The keys of the **Pattern** dictionary are described in more detail in Section 4.4.1 of the *Supplement: PostScript Language Reference Manual*.

2.3 Painting With a Pattern Dictionary

For painting operations using the new **PatternType 2 Pattern** dictionary, the **Pattern** dictionary acts as the current color in the current graphics state. Once the pattern is created and set with the **makepattern** and **setpattern** operators, or with the **setcolorspace** and **setcolor** operators, the **fill**, **stroke**, **show**, and

imagemask operators can then be used with this pattern as the current color to paint a path or mask with the gradient fill. The pattern coordinate space is obtained in the same way as with **PatternType 1** patterns. However, instead of executing a **PatternType 1 PaintProc** procedure, the shape and color transitions described by the **Shading** dictionary are interpreted relative to this coordinate space to create a *logical paint* with which graphical objects can be rendered.

Example 3 *Using a PatternType 2 Pattern dictionary for shading*

```
%PATTYP2.PS
%This example demonstrates the use of a Type 2 Pattern
%dictionary for smooth shading.
%Define various graphics state variables
/inch {72 mul} def
...
%Define some object to shade
...
%Define the Pattern dictionary, Shading dictionary, and
%Function dictionary
gsave
<<
    /PatternType 2
    /Shading <<
        /ShadingType 2
        /ColorSpace /DeviceRGB
        /Background [0 1 1]% A Cyan background
        /Coords [0 0 8.5 inch 11 inch]
        /Domain [0 1]
        /Function <<
            /FunctionType 2
            /Domain [0 1]
            /C0 [1 0 1]  % Magenta
            /C1 [0 1 1]  % Cyan
            /N 1
        >>
    >>
>>
makepattern
setpattern
%Now perform shade of object by calling a standard
%PostScript rendering operator such as fill or image
...
grestore
showpage
```

Note *Complete PostScript language files containing these examples accompany this document.*

If the **BBox** key is present in the current **Shading** dictionary, it is used to clip the logical painting region. This region may also be affected by the geometry of the shading. All color values are interpreted relative to the **ColorSpace** key in the **Shading** dictionary.

If a **Background** color is defined in the **Shading** dictionary, that color is used first to fill the background of the object or region being painted. This is equivalent to executing a **fill** operation or other painting operation first with the background color and then again with the gradient fill pattern.

Note *The **Background** key is provided because the two-step sequence of operations described above would be verbose, especially for text or the **imagemask** operator. The new approach is most beneficial for gradient fill patterns that do not cover the entire area of the object being rendered.*

Some smooth shading methods allow for use of arbitrarily large streams of data through the **DataSource** key. Since gradient fills defined by **PatternType 2** pattern resources may be used multiple times, this data must be provided in a reusable form. Data stored in a string is reusable, but the strings are limited in size to 64 kilobytes (Kb). Data stored in files pointed to by **currentfile** or simply in files stored on a hard disk are not reusable. In order to use data stored as internal or external files, it must be converted into a reusable stream by means of the **ReusableStreamDecode** filter (see Section 3.3.7 of the *Supplement: PostScript Language Reference Manual*).

Note *A non-reusable stream of data from a **Shading** dictionary may only be used with the **shfill** operator. In other words, if a non-reusable stream of data is needed with the current **Shading** dictionary, then the **shfill** operator must be used instead of a **Pattern** dictionary to render the object. The only exception to this is for shading with sampled functions **(ShadingType 0**). In this case, non-reusable streams cannot be used, even if **shfill** is used.*

Note ***PatternType 2** gradient fills do not tile (create a repeated pattern). To create a tiling or repeating pattern containing a gradient fill, use the **shfill** operator in the **PaintProc** procedure of a **PatternType 1** pattern resource. See Section 2.1 for more information on the **shfill** operator.*

2.4 Shading Dictionaries

A **Shading** dictionary is used to describe the various smooth shading methods supported in LanguageLevel 3. There are currently seven supported types of smooth shading, each associated with a specific value of the key **ShadingType**. All **Shading** dictionaries contain the following keys: **ShadingType**, **ColorSpace**, **Background**, **BBox**, and **AntiAlias**. From this list, only the **ShadingType** and **ColorSpace** keys are required in the **Shading** dictionary definition. Some types of **Shading** dictionaries also include a **Function** dictionary key (see Sections 2.13 through 2.17). In such

cases, the **Shading** dictionary usually defines the geometry of the shading, while the **Function** dictionary defines the color transitions across that geometry.

In addition to these keys, a **Shading** dictionary must have entries specific to each shading type (the value of the **ShadingType** key). Table 2 through Table 11 list the keys specific to each of the seven shading types. For complete descriptions of each key defined for each **Shading** dictionary, see Section 4.4 of the *Supplement: PostScript Language Reference Manual*.

A **Shading** dictionary can be defined within a **Pattern** dictionary (see Sections 2.2 and 2.3) or used as the parameter to the **shfill** operator (see Section 2.1).

Note *There is a new implicit resource category called **ShadingType**. Currently, the only supported instances of this category type are 0 through 7, corresponding to the **Shading** types discussed in Sections 2.6 through 2.12.*

2.5 ColorSpace Key for Shading Dictionaries

The **ColorSpace** key defines not only the color space in which color values are specified in the shading, but also the color space in which the gradient fill calculations are performed. The gradient fills between colors defined by most shadings are implemented using a variety of interpolation algorithms, and these algorithms are sensitive to the characteristics of the color space. Linear interpolation, for example, may have observably different results if specified in CMYK color space as opposed to CIE L*a*b* color space, even if the starting and ending colors are perceptually identical. The difference arises because the two color spaces are not linear relative to one another. Smooth shaded objects, paths, or masks are rendered using the following rules:

- If the value of the **ColorSpace** key is device-dependent and different from the process color space of the device, then the resulting color values will be converted to device colors using standard conversion formulae. To maximize performance, these conversions may take place at any time. Thus, any shadings defined with device-dependent color spaces may have color gradient fills that are somewhat device-dependent. This does not apply to any of the axial and radial shadings, since these perform gradient fill calculations on a single variable and then convert to device colors after the interpolation.

- If the value of the **ColorSpace** key is device-independent, then all gradient fill calculations will occur in the device-independent color space. Conversion to device colors will occur only after all interpolation calculations are performed. Thus, the color gradient fills will be device-independent for the colors generated at each point.

- If the value of the **ColorSpace** key is **Separation** or **DeviceN** (See Sections 3.1, 6.4, 4.2 of the *Supplement: PostScript Language Reference Manual*) and the specified colorant(s) is/are not defined by **ProcessColorModel** or **SeparationColorNames** so that the **alternativeSpace** key must be used, then the gradient fill calculations will be performed in the special color space prior to conversion to the alternative color space. Thus, non-linear **tintTransform** functions will be accommodated for the best possible representation of the shading method. If the specified colorant(s) is/are supported, then no color conversion calculations are needed.

- If the value of the **ColorSpace** key is **Indexed** (See Section 4.2 of the *Supplement: PostScript Language Reference Manual*), then all color values specified in the shading will be immediately converted to the base color space. Depending on whether the base color space is device-dependent or device-independent, gradient fill calculations will be performed as stated above. Interpolation never occurs in the **Indexed** color space, which is *quantized* (discrete steps as opposed to continuous color) and inappropriate for calculations that assume a continuous range of colors. Also, as described for the available **ShadingType** entries, the **Indexed** color space may not be allowed in some shadings (see Sections 2.6 through 2.12). For example, the **Indexed** color space is not allowed for axial or radial shadings that perform interpolation calculations on a single variable and then convert to parametric colors, which are assumed to represent a continuous range of colors. Similarly, the **Indexed** color space is not allowed for function-based shadings, which interpolate between sampled color values.

2.6 ShadingType 1: Function-Based Shading

ShadingType 1 is intended for sophisticated gradient fills in cases where the other types of shading – such as axial, radial, triangle mesh, and Coons patch – are not sufficient. **ShadingType 1** specifies function-based shading. In other shading types, a function can be used to describe the color transitions across the geometry of the shading. In this case, the function describes the shading itself.

Using the **ShadingType 1** shading method, the color of every point in the domain is defined by a two-dimensional object that uses a mathematical or sampled function to map each point in the domain to a specific color value. The mathematical function does not necessarily have to be smooth or continuous.

Table 2 shows the keys that define a **Shading** dictionary for **ShadingType 1**. The keys are described in more detail below.

Table 2 *Keys for ShadingType 1 Shading dictionaries*

Key	Type	
ShadingType	integer	required
ColorSpace	name or array	required
Background	array	optional
BBox	array	optional
AntiAlias	boolean	optional
Domain	array	optional
Matrix	array	optional
Function	dictionary or array	required

The **ShadingType** key is required for every **Shading** dictionary, regardless of its type. It specifies the shading type or method to be used. In this case, the value must be 1.

The **ColorSpace** key is also required for every **Shading** dictionary, regardless of its type. The value may be any device-dependent (including **DeviceN**), device-independent, or special color space, except **Pattern**. The **Indexed** color space requires some special handling, as discussed in Section 2.5). All color values for this shading are interpreted relative to the color space defined by this key.

The **Background** key is optional for every **Shading** dictionary. It is an array of color components appropriate to the **ColorSpace** key. It specifies a single color value.

The **BBox** key is optional for every **Shading** dictionary. It is an array of four numbers interpreted as the lower-left and upper-right coordinates in the current coordinate space at the time the shading is imaged. If this key is present, then the shading is clipped to the intersection of this bounding box and the current clipping path. If the key is not present, then the shading is clipped to the bounding box of the clipping region at the time the shading is imaged.

The **AntiAlias** key is optional for every **Shading** dictionary. It is a Boolean value with a default value of false. If true, the shading function, defined by the key **Function**, is combined with a convolution function to average shading values across device pixels. This produces a more device-independent representation when the spatial frequency of the shading is more than about half the device resolution. It also makes shadings more resistant to variations in appearance due to changes in the current transformation matrix (CTM).

Note *The implementation of the **AntiAlias** key is device specific. Some devices may have a Null implementation, in which case, the key is ignored.*

Domain is an optional array of four numbers specifying the rectangular domain of arguments with which the color function(s) are called. The default domain value is [0 1 0 1].

Matrix is an optional transformation matrix that specifies the mapping from the **Domain** value (see above) into the coordinate space in which the shading is being imaged. The default matrix is the identity matrix.

The **Function** key is optional for **ShadingType 1**. It specifies a single 2-in *n*-out **Function** dictionary or an array of *n* 2-in 1-out **Function** dictionaries, where *n* is the number of color components in the **ColorSpace** entry.

Note *The **Domain** value defined in the **Function** dictionary must be a superset of the **Domain** value of its **Shading** dictionary. If the values returned by the function are out of range for the given color component, then the values will be adjusted to the nearest allow value (clipped).*

Any points that are within the region defined by the **BBox** value but are outside the **Domain** value will be left unpainted. However, in the case of gradient fill patterns with a **Background** color specified, such points will be painted with the background color.

If the function is undefined at any point within its declared **Domain** value, an **undefinedresult** error may occur, even if such points are outside the region defined by the **BBox** value.

Example 4 *Function-based shading (ShadingType 1)*

```
%FUNSH01.PS
%This example demonstrates smooth shading using a sampled
%function and shfill
%Set up graphics state and other variables
/inch {72 mul} def
...
%Define the shading dictionary
gsave
<<
    /Domain [0 6.5 inch 0 9 inch]
    /Matrix [1 0 0 1 1 inch 1 inch]
    /ShadingType 1
    /ColorSpace /DeviceRGB
    /Function <<
        /FunctionType 0
        /Order 1
        /Domain [0 1 0 1]
        /Range   [0 1 0 1 0 1]
        /Decode [0 1 0 1 0 1]
        /DataSource <
            FF 00 00   80 80 00   44 44 00   00 00 C0
            80 C0 00   FF FF FF   FF FF FF   FF 00 00
            FF FF FF   FF FF FF   FF FF FF   FF 00 FF
            80 00 80   00 FF 00   FF FF 00   C0 C0 00
         >
        /BitsPerSample 8
        /Size [4 4]
    >>
>>
shfill
grestore
showpage
```

Note *Complete PostScript language files containing these examples accompany this document.*

For a complete list and description of the keys in the **ShadingType 1 Shading** dictionary, see Table 4.8 in the *Supplement: PostScript Language Reference Manual*.

2.7 ShadingType 2: Axial Shading

Axial shading is so called because the geometry of the gradient fill is defined along a line or axis defined by a pair of endpoints. It is called axial rather than *linear* because linear is just one form of interpolation that can be used to the define the gradient fill of the shading. Thus, the transition from one color to another could vary linearly or non-linearly along the line or axis.

ShadingType 2 defines a color gradient fill along a line (axis) between two endpoints. This gradient fill can optionally be extended beyond the endpoints by continuing the boundary (endpoint) colors. This gradient fill is determined by a one-dimensional interpolation specified by the **Function** key.

Table 3 shows the keys that define a **Shading** dictionary for **ShadingType 2**. The keys are described in more detail below.

Table 3 *Keys for ShadingType 2 Shading dictionaries*

Key	Type	
ShadingType	integer	required
ColorSpace	name or array	required
Background	array	optional
BBox	array	optional
AntiAlias	boolean	optional
Domain	array	optional
Extend	array	optional
Function	dictionary or array	required
Coords	array	required

The **ShadingType**, **ColorSpace**, **Background**, **BBox**, and **AntiAlias** keys are defined as for **ShadingType 1**.

ShadingType must be 2.

The **Coords** key is a required array of four numbers that specify the start and end coordinate pairs [x_0, y_0, x_1, y_1].

The **Domain** key is an optional array of two numbers. A parametric variable *t* is considered to vary linearly between these two values as the gradient fill varies between the start and endpoints, respectively (from **Coords**). The variable *t* becomes the argument to the color function(s). The default value of **Domain** is [0 1].

Extend is an optional array of two Boolean values that specify whether or not to extend the start and end colors past the start and endpoints, respectively. The default value for each element of the array is false.

Function is an optional key. It can be either a single 1-in *n*-out function dictionary or an array of *n* 1-in 1-out function dictionaries, where *n* is the number of components in the **ColorSpace** entry. The **Function(s)** is/are called with the parameter *t* defined in **Domain** (see above).

Note *The **Domain** value defined in the **Function** dictionary must be a superset of the **Domain** value of its **Shading** dictionary. If the values returned by the function are out of range for the given color component, then the values will be adjusted to the nearest allow value (clipped).*

ShadingType 2 defines a field of color that varies along the line between the start and end coordinates and extends infinitely away from the line. If the **Extend** Boolean values are true, the field may also extend infinitely far along the line, past either or both endpoints, using the constant color of that endpoint.

The gradient fill is accomplished by linearly mapping the range between the endpoints to the value of **Domain** defined in the **Shading** dictionary, as follows. Every point *(x,y)* is mapped to a coordinate space where *(0,0)* corresponds to (x_0,y_0) and *(1,0)* corresponds to (x_1,y_1). Since all points on a line perpendicular to the line from *(0,0)* to *(1,0)* in that space will have the same color, only the new value of *x*, called *x'*, needs to be computed in that space:

$$x' = ((x_1 - x_0)(x - x_0) + (y_1 - y_0)(y - y_0)) / ((x_1 - x_0)^2 + (y_1 - y_0)^2)$$

Once *x'* is calculated, the value of the parametric value *t* can be determined. This value is used as the input arguement to the **Function** key, and the returned value(s) are used to paint the gradient fill.

Note *This parametric gradient fill may not be used with the value of **ColorSpace** set to **Indexed**.*

The value of *t* is determined as follows:

- If $x' < 0$ and the first value in the **Extend** array is true, the parameter *t* is set to the value of t_0. However, if the first value in the **Extend** array is false, that point is not painted.
- If $x' > 1$ and the second value in the **Extend** array is true, the parameter *t* is set to the value of t_1. However, if the first value in the **Extend** array is false, that point is not painted.
- If $0 <= x' <= 1$, then $t = t_0 + (t_1 - t_0)x'$.

Example 5 *Axial shading (ShadingType 2)*

```
%AXSHO2.PS
%This is a simple illustration of Shading type 2.
%Define graphics state and other variables
/inch {72 mul} def
...
gsave
<<%define the shading dictionary
    /ShadingType 2
    /ColorSpace /DeviceRGB
    /Coords [3 inch 3 inch 5.5 inch 8 inch]
    /BBox   [1 inch 1 inch 7.5 inch 10 inch]
    /Extend [true false]% Extend only one end
    /Function
      <<
        /FunctionType 2
        /Domain [0 1]
        /C0      [1 0 1]  %magenta
        /C1      [0 1 1]  %cyan
        /N       1
      >>
>>
shfill
grestore
showpage
```

Note *Complete PostScript language files containing these examples accompany this document.*

For a complete list and description of the keys in the **ShadingType 2 Shading** dictionary, see Table 4.9 in the *Supplement: PostScript Language Reference Manual*.

2.8 ShadingType 3: Radial Shading

ShadingType 3 defines a color gradient fill between two circles or cylinders. This method is most commonly used to produce the visual effect of a three-dimensional sphere or cone. **ShadingType 3** is accomplished by one-dimensional interpolation along the radius of the circle, from the center of the circle outward. The resulting path can be either circular or elliptical.

Table 4 shows the keys that define a **Shading** dictionary for **ShadingType 3**. The keys are described in more detail below.

Table 4 *Keys for ShadingType 3 Shading dictionaries*

Key	Type	
ShadingType	integer	required
ColorSpace	name or array	required
Background	array	optional
BBox	array	optional
AntiAlias	boolean	optional
Domain	array	optional
Extend	array	optional
Function	dictionary or array	required
Coords	array	required

The **ShadingType**, **ColorSpace**, **Background**, **BBox**, and **AntiAlias** keys are defined as for **ShadingType 1**.

ShadingType must be 3.

Coords is a required array of six numbers that specify the center coordinates and radii of the start and end circles [x_0, y_0, r_0, x_1, y_1, r_1]. The radii r_0 and r_1 must be greater than or equal to zero. If one radius is zero, that circle is treated as a point. If both radii are zero, nothing is rendered.

The **Domain**, **Extend**, and **Function** keys are defined exactly the same as with **ShadingType 2** (See Section 2.7).

When using **ShadingType 3**, the following results can be observed:

- If the circle with the smaller radius is extended by the **Extend** Boolean value (a value of true means to shade beyond the endpoint), the interior of that circle will be painted (shaded) with the constant color of that circle. That is, the color defined at the radius of the smaller circle will be used.

- If the circle with the larger radius is extended by the **Extend** Boolean values, the exterior of that circle will be painted with the constant color of that circle. The resulting paint (shading) is limited by the value of the **BBox** key.

- If the start and end circles are not concentric and the larger radius is given first (specified by **Coords**), then the resulting gradient fill will depict a cone pointing out of the page (toward the viewer).

- If, under the same conditions, the smaller radius is given first (specified by **Coords**), then the resulting gradient fill will depict a cone pointing into the page (away from the viewer).

- If a spherical gradient fill is needed, then the larger circle will entirely contain the smaller circle.

The gradient fill is accomplished by mapping the region between the start and end circles to a linear parametric variable whose domain is the value of the **Domain** key. The resulting parametric value is used as the input argument to the **Function** key. The returned value(s) from **Function** is/are used to paint the gradient fill.

The parametric variable $s = (t - t_0) / (t_1 - t_0)$ varies linearly between 0 and 1 as *t* varies across the value of **Domain**. The parametric equations for the center and radius of the gradient fill circle moving between the start circle and the end circle as a function of *s* are as follows:

$$x_c(s) = x_0 + s * (x_1 - x_0)$$

$$y_c(s) = y_0 + s * (y_1 - y_0)$$

$$r(s) = r_0 + s * (r_1 - r_0)$$

Given a geometric coordinate position (x, y) in or along the gradient fill, the corresponding value of *s* can be determined by solving the quadratic constraint equation:

$$[x - x_c(s)]^2 + [y - y_c(s)]^2 = [r(s)]^2$$

Given *s*, the value of *t* can be found, which is then passed to the **Function** key. The value(s) returned by **Function** is/are used to determine the color at the position *(x, y)*. If both roots of the equation are in the domain [0 1], then the larger value of *s* defines the color because it comes after the smaller value and thus overlays it. For values of *s* outside the domain [0,1], the **Extend** values determine how the shading will be painted. The following rules hold true for pixel coordinates *(x,y)* satisfying the above equation.

- If the start (first) **Extend** value is false, then pixels corresponding to values of $s < 0$ are left unpainted or are painted with the **Background** color, if one is specified.
- If the end (last) **Extend** value is false, then pixels corresponding to values of $s > 1$ are left unpainted or are painted with the **Background** color, if one is specified.
- If the start **Extend** value is true and $r[s] >= 0$, then pixels corresponding to the values of $s < 0$ are painted with the start color.
- If the end **Extend** value is true and $r[s] >= 0$, then pixels corresponding to values of $s > 1$ are painted with the end color.

Note *For cases where one circle is not completely contained within the other, **Extend** values of true can cause painting to extend beyond the areas defined by the two circles.*

Note *This parametric gradient (vignette) may not be used with the value of **ColorSpace** set to **Indexed** color space.*

Example 6 *Radial shading (ShadingType 3)*

```
%RADSAMP.PS
%This example illustrates the use of ShadingType 3.
%Define the graphics state and other variables
/inch {72 mul} def
...
% Setup up the shading and function dictionaries
gsave
<<
    /ShadingType 3
   /Coords [3.25 inch 3.5 inch 3 inch 3.25 1.5 add inch 3.5
3.5 add inch 3.25 inch]
    /ColorSpace /DeviceRGB
    /Function <<
        /FunctionType 0
        /Order 1
        /BitsPerSample 16
        /Domain [0 1]
        /Decode [0.5 0 1 0.5 0 0.99]
        /Range   [0 1 0 1 0 1]
        /Size [36]
        /DataSource <
         0000 0000 0000
         164F 164F 164F
         2C74 2C74 2C74
         ...
         2C74 2C74 2C74
         164F 164F 164F
         0000 0000 0000
          >
    >>
>>
shfill
grestore
showpage
```

Note *Complete PostScript language files containing these examples accompany this document.*

For a complete list and description of the keys in the **ShadingType 3 Shading** dictionary, see Table 4.10 in the *Supplement: PostScript Language Reference Manual*.

2.9 ShadingType 4: Free-Form Gouraud-Shaded Triangle Meshes

ShadingType 4 defines a common construct used by many three-dimensional applications for imaging complex colored and shaded objects. Gouraud-shaded triangle meshes construct paths composed entirely of triangles. The color of each vertex of a triangle is specified, and Gouraud interpolation is used to determine the color of the interior points. A primary use of these meshes is to allow the specification of polygon vignettes as triangle meshes with nonlinear interpolation functions.

Table 5 shows the keys that define a **Shading** dictionary for **ShadingType 4**. The keys are described in more detail below.

Table 5 *Keys for ShadingType 4 Shading dictionaries*

Key	Type	
ShadingType	integer	required
ColorSpace	name or array	required
Background	array	optional
BBox	array	optional
AntiAlias	boolean	optional
DataSource	various	required
BitsPerCoordinate	integer	required (see note)
BitsPerComponent	integer	required (see note)
BitsPerFlag	integer	required (see note)
Decode	array	required (see note)
Function	dictionary or array	optional

Note *The **BitsPerCoordinate**, **BitsPerComponent**, **BitsPerFlag**, and **Decode** keys are required unless the value of **DataSource** is an array.*

The **ShadingType**, **ColorSpace**, **Background**, **BBox**, and **AntiAlias** keys are defined as for **ShadingType 1**.

ShadingType must be 4.

The **BitsPerCoordinate** key is required unless **DataSource** is an array. This integer value specifies the number of bits used to represent each vertex coordinate. The data is decoded based on the value of the **Decode** key. Allowed values are 1, 2, 4, 8, 12, 16, 24, and 32.

The **BitsPerComponent** key is required unless **DataSource** is an array. This integer value specifies the number of bits used to represent each color component. The data is decoded based on the value of the **Decode** key. Allowed values are 1, 2, 4, 8, 12, and 16.

BitsPerFlag is an integer value that is required unless **DataSource** is an array. It specifies the number of bits used to represent the edge flag for each vertex. Allowed values are 2, 4, and 8; the allowed values for the edge flag are 0, 1, and 2.

The **Decode** key is required unless **DataSource** is an array. It specifies how to decode coordinate and color component data into the ranges of values appropriate for each. The ranges are specified as $[x_{min}\ x_{max}\ y_{min}\ y_{max}\ c_{1,min}, c_{1,max}, \ldots, c_{n,min}, c_{n,max}]$.

Function is an optional key that specifies either a single 1-in *n*-out **Function** dictionary or an array of *n* 1-in one-out **Function** dictionaries (*n* is the number of components in the **ColorSpace** key). If **Functio**n is specified, the vertex color data for the mesh must be specified by single values rather than with color tuples. The **Function** dictionary will then be called with each interpolated color value to determine the actual color of each vertex.

Note *The **Domain** value defined in the function dictionary must be a superset of the **Domain** value of its **Shading** dictionary. If **DataSource** is an array, in which case **Decode** is not defined for this **Shading** dictionary, the **Domain** value defined in the function dictionary must be a superset of the domain [0 1]. In both cases, input values will be clipped to the subset of the function domain. If the value(s) returned by the function(s) is/are out of range for a given color component, the value(s) will be adjusted to the nearest allowed value (clipped).*

Note *The **Function** key may not be used with unencoded vertex data; it may not be used if the **ColorSpace** key is set to **Indexed**.*

ShadingType 4, as well as **ShadingType 5**, **6**, and **7**, require a source of data to define triangle or patch vertices and colors. There are two main ways to specify the data source, each with varying degrees of complexity and flexibility. This data source is defined with the **DataSource** key.

The easiest method for specifying the data source is with an array of numbers that define the vertices and the color components of those vertices. Using an array as a data source is conceptually very simple; the other methods of providing the data allow for much greater flexibility in the way the data is interpreted.

This other method requires the use of a string or a reusable stream as a data supply. A string may be appropriate for use if the data is smaller than 64Kb in length; otherwise, a reusable stream must be used. In either case, the data

must be encoded, and there are a variety of methods for specifying the encoding (bits per value) of vertex data, color data, and flag data. Also, an optional function may then be used to define the color transitions across the geometry of the shading. When a function is used, the **Encode** and **Decode** keys define how the encoded data values are mapped into the domain of the function.

The **DataSource** key provides the sequence of vertex data needed to build each triangle in the mesh.

The data for the i^{th} vertex, v_i, is of the form

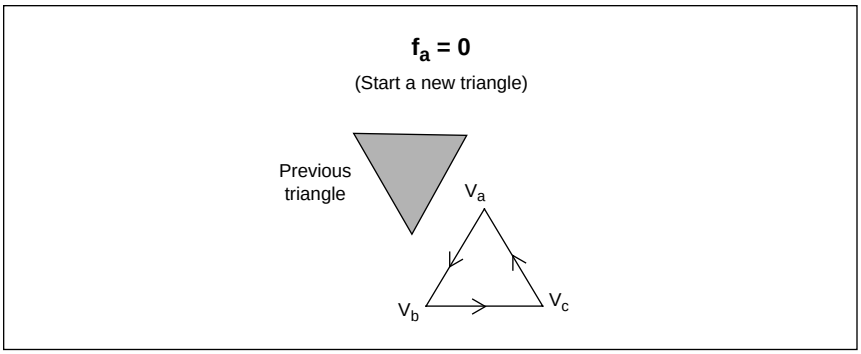
$$f_i\ x_i\ y_i\ c_{i\ ,1}...c_{i,n}$$

where *x* and *y* are vertex coordinates, *c* is a tuple of color values, *f* is the *edge flag* for each vertex, and *n* is the number of color components. The edge flag defines which triangle edges are shared. The number of color components for each vertex is the same as the number of color components defined for the current color space, as specified by the **ColorSpace** key. For example, if the current color space is RGB, then there must be three color components for each vertex. If the **Shading** dictionary contains the **Function** key, then only one color component, $c_{(i,1)}$, is permitted in each sequence of vertex data.

Triangle meshes are built up as follows:

The first vertex, v_a, of the first triangle must have an edge flag value of 0 (that is, $f_a = 0$), which means that this is a new triangle (not attached to any previous triangle). The edge flags of the next two vertices (v_b and v_c) are ignored, but they are a required part of the data. These three vertices define the first triangle, (v_a, v_b, v_c). Figure 4 shows this first triangle.

Figure 4 *Defining a new triangle (f = 0)*



Subsequent triangles are defined by a single new vertex and an edge that is shared with the preceding triangle. This edge contains two vertices of the preceding triangle (see Figure 5). Given triangle (v_a, v_b, v_c), where vertex *a* is

older than vertex *b* and vertex *b* is *older* than vertex *c* (older means earlier in the data source), a new triangle can be formed on side v_{bc} or v_{ac}, creating a new vertex v_d (see Figure 5). If the edge flag $f_d = 1$ (side v_{bc}), the next vertex forms the triangle v_b, v_c, v_d. If the edge flag $f_d = 2$ (side v_{ac}), the next vertex forms the triangle v_a, v_c, v_d. The edge on side v_{ab} is assumed to be shared with the preceding triangle, so is not an appropriate edge for continuing the triangle mesh.

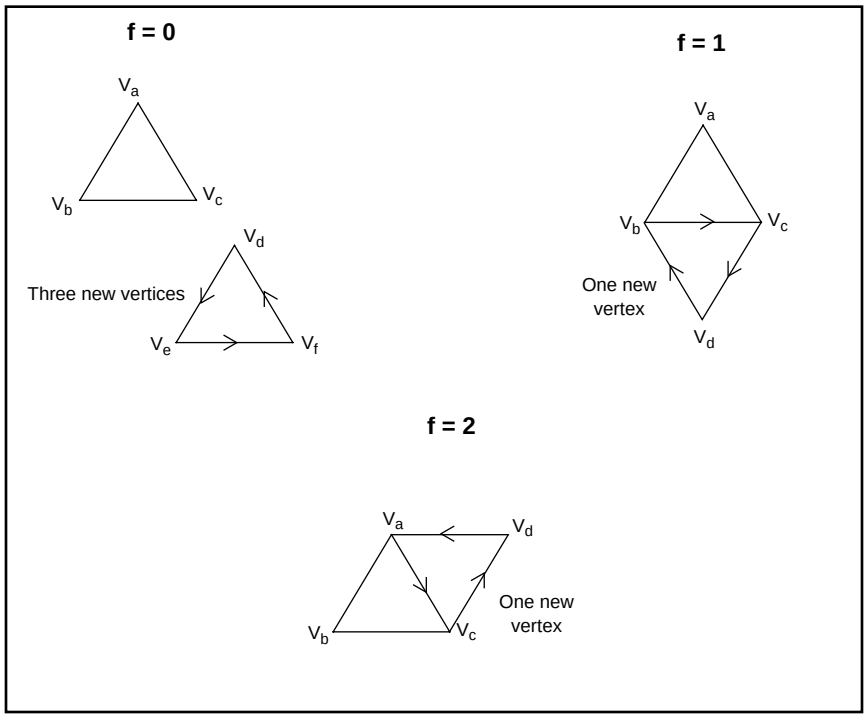
Whenever the edge flag $f = 0$, a new triangle is started. At least two more vertices must be provided, but their edge flags are ignored. Whenever the edge flag $f = 1$ or $f = 2$, a new vertex is added to complete the next triangle in the mesh. An edge flag value $f = 3$ is not allowed.

The data stream for multiple triangles will look something like this:

$$f_1 x_1 y_1 c_{1,1} \ldots c_{1,n}\, f_2 x_2 y_2 c_{2,1} \ldots c_{2,n}\, f_3 x_3 y_3 c_{3,1} \ldots c_{3,n}\, f_4 x_4 y_4 c_{4,1} \ldots c_{4,n}\, f_5 x_5 y_5 c \ldots$$

where *n* is the number of color components. The first three sets of data (shown with the subscripts 1, 2, and 3) represent the first triangle, and each additional set of data (subscript 4 and above) represents a new triangle.

Figure 5 *How the value of the edge flag determines which edge is used for the next triangle*



It is possible to create complex shapes using triangle meshes by simply varying the edge at which the next triangle is formed. Figure 6 shows two very simple examples. To create **Mesh 1**, start with triangle 1 and create each new triangle using the following edge flag values:

Table 6 *Edge flag values for each triangle in Mesh 1*

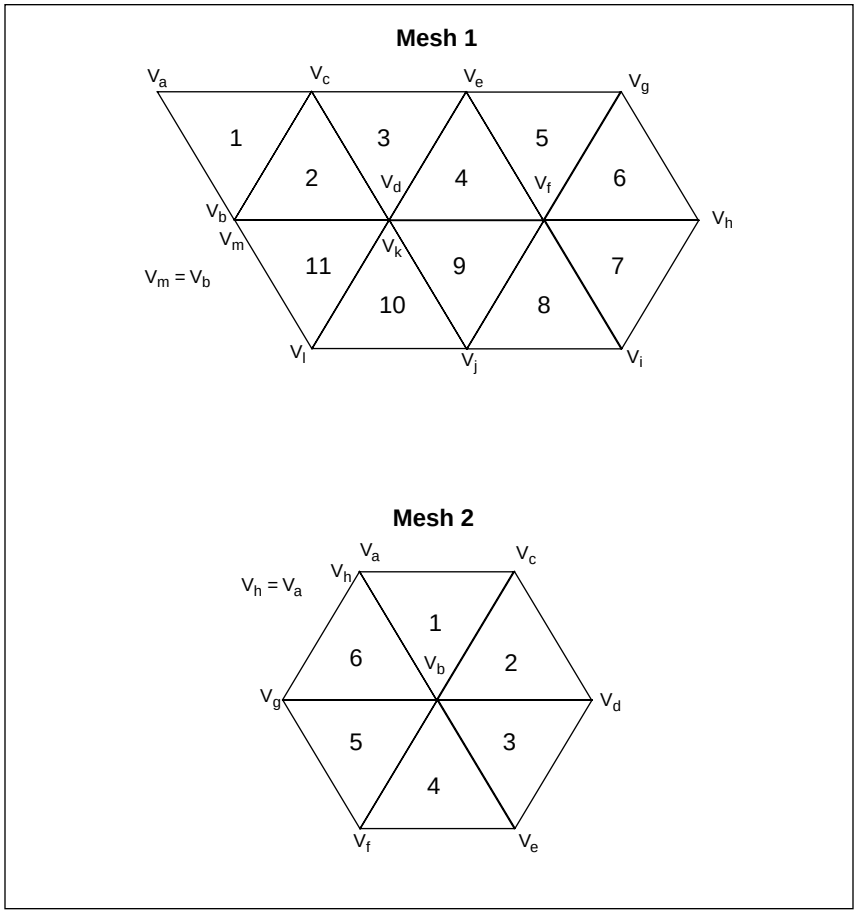
Triangle	Edge Flag Value
1	$f_a = 0$
2	$f_d = 1$
3	$f_e = 2$
4	$f_f = 1$
5	$f_g = 2$
6	$f_h = 1$
7	$f_i = 1$
8	$f_j = 1$
9	$f_k = 2$
10	$f_l = 1$
11	$f_m = 2$

To create **Mesh 2**, start with triangle 1 and create each new triangle using the following edge flags:

Table 7 *Edge flag values for each triangle in Mesh 2*

Triangle	Edge Flag Value
1	$f_a = 0$
2	$f_d = 1$
3	$f_e = 1$
4	$f_f = 1$
5	$f_g = 1$
6	$f_h = 1$

Figure 6 *Varying the value of the edge flag to create different shapes*



This representation optimizes useful tiling meshes, although it can somewhat complicate the data representation. The value of **DataSource** must provide a whole number of triangles with appropriate vertex edge flags; otherwise, a **rangecheck** error will occur. If the mesh contains only a few vertices (less than about 30; however note that up to 64Kb of data is allowed for arrays), the vertices may be represented by a simple array of numbers. In this case, only the **ShadingType**, **ColorSpace**, and **DataSource** keys are required in the **Shading** dictionary. If the mesh contains many vertices (more than about 30), the data should be encoded compactly and drawn from a stream. This encoding is specified by the **BitsPerCoordinate**, **BitsPerComponent**, **BitsPerFlag**, and **Decode** keys. Each vertex coordinate pair *(x, y)* is expressed in *2* * **BitsPerCoordinate** bits, each vertex color tuple *c* is expressed in *n* * **BitsPerComponent** bits, and each vertex edge flag *f* is expressed in **BitsPerFlag** additional bits.

Each set of vertex data (edge flag, coordinate pair, and color tuple) takes an integer number of bytes; therefore, if the total number of bits in the vertex data is not divisible by eight, the vertex data is padded with ignored bits

inserted between the coordinate and color data. The coordinate and color data is decoded based on the **Decode** array, similar to the decoding done with **image** data.

If the **Function** key value is specified, then the vertex color data for the mesh must be specified by single values *t* rather than color tuples *c*. All linear interpolation within the triangle mesh will be done using the values of *t*, and after interpolation, the value(s) returned from **Function** will be used to determine the color of each point.

Note *Using free-form Gouraud-shaded triangle meshes differs from using an* ***Indexed*** *color space for the shading. If an* ***Indexed*** *color space is used, the vertex coordinates are converted to the base color space first, and linear interpolation occurs in that color space. Thus, there is no opportunity to effect a nonlinear interpolation using an* ***Indexed*** *color space.*

Example 7 *Free-form Gouraud-shaded triangle meshes (ShadingType 4)*

```
%TRITYP4.PS
%This example demonstrates ShadingType 4
%Define graphics state and other variables
/inch {72 mul} def
...
/DeviceRGB setcolorspace
...
%Define the shading and function dictionaries
gsave
<<
    /ShadingType 4
    /ColorSpace [/DeviceRGB]
    /DataSource
      [
        0                       % edge flag = new triangle
        0 0    1 0 1            % magenta
       0  % dummy edge flag for second edge
        4 inch 4 inch    0 1 1  % cyan
       0  % dummy edge flag for third edge
        -4 inch 4 inch   0 1 1  % cyan
        2                       % edge flag
        -4 inch -4 inch 0 1 1   % cyan
        2                       % edge flag
        4 inch -4 inch 0 1 1    % cyan
        2                       % edge flag
        4 inch 4 inch 0 1 1     % cyan
      ]
>>
shfill
grestore
showpage
```

Note *Complete PostScript language files containing these examples accompany this document.*

For a complete list and description of the keys in the **ShadingType 4 Shading** dictionary, see Table 4.11 in the *Supplement: PostScript Language Reference Manual*.

2.10 ShadingType 5: Lattice-Form Gouraud-Shaded Triangle Meshes

The **ShadingType 5** shading method is almost identical to **ShadingType 4**, with a few important exceptions. For **ShadingType 4**, vertices are specified in a free-form geometry; for **ShadingType 5**, vertices must be in a pseudo-rectangular *lattice* geometry. That is, the lattice need not be strictly rectangular, but the set of vertices must be organized into rows. (The rows do not need to be geometrically linear). In addition, the lattice-form triangle mesh does not require the use of edge flags but instead defines the number of vertices in each row of the lattice-form triangle mesh using the **VerticesPerRow** key. Finally, the interpretation of the **DataSource** key is different (see below).

Given *m* rows of *n* vertices, where the number of vertices is given in the value of the **VerticesPerRow** key, triangles are constructed using the following triplets of vertices:

$(V_{i,j}, V_{i,j+1}, V_{i+1,j})$ *for* $0 <= i <= (m - 2), 0 <= j <= (n - 2)$

$(V_{i,j+1}, V_{i+1,j}, V_{i+1,j+1})$ *for* $0 <= i <= (m - 2), 1 <= j <= (n - 1)$

Conceptually, the simplest possible lattice triangle mesh contains four points (vertices) in two rows of two vertices, as shown in Figure 7. Also shown in this figure are examples of ideal and pseudorectangular lattices.

Figure 7 *Simple lattice forms*

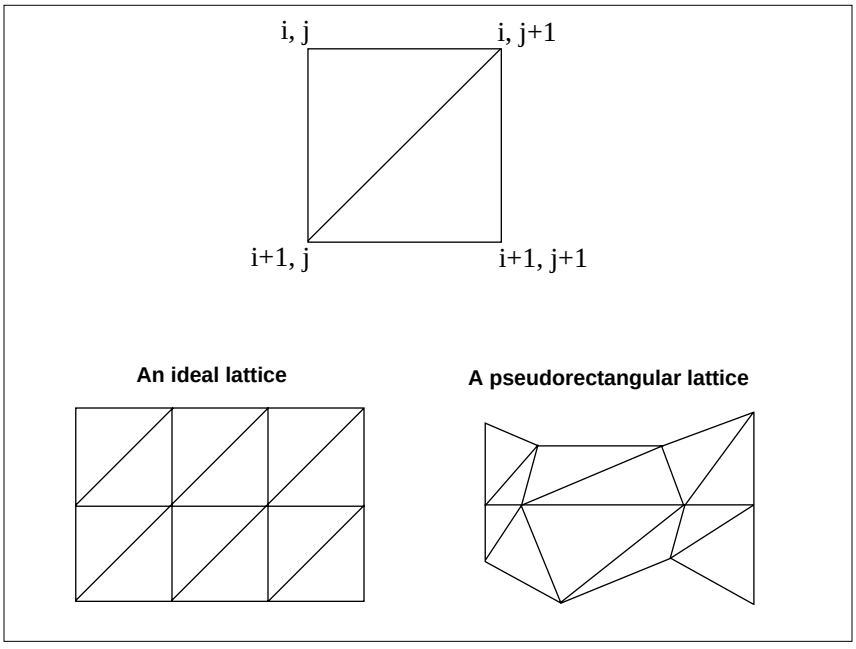


Table 8 shows the keys that define a **Shading** dictionary for **ShadingType 5**. The keys are described in more detail, below.

Table 8 *Keys for ShadingType 5 Shading dictionaries*

Key	Type	
ShadingType	integer	required
ColorSpace	name or array	required
Background	array	optional
BBox	array	optional
AntiAlias	boolean	optional
DataSource	various	required
BitsPerCoordinate	integer	required (see note)
BitsPerComponent	integer	required (see note)
Decode	array	required (see note)
VerticesPerRow	integer	required
Function	dictionary or array	optional

Note *The **BitsPerCoordinate**, **BitsPerComponent**, and **Decode** keys are required unless the value of **DataSource** is an array.*

The **ShadingType**, **ColorSpace**, **Background**, **BBox**, and **AntiAlias** keys are defined as for **ShadingType 1**.

ShadingType must be 5.

The **BitsPerCoordinate**, **BitsPerComponent**, **Decode**, and **Function** keys are all defined as for **ShadingType 4** (See Section 2.9).

DataSource is a required key that provides a sequence of vertex coordinate and color data that specifies the Lattice-form triangle mesh. It can be an array of numbers, a string, or a stream (see the previous discussion for **ShadingType 4**).

The data stream for multiple row of triangles will look something like this:

$$x_1 y_1 c_{1,1} \ldots c_{1,n}\ x_2 y_2 c_{2,1} \ldots c_{2,n}\ x_3 y_3 c_{3,1} \ldots c_{3,n}\ x_4 y_4 c_{4,1} \ldots c_{4,n} \ldots\ x_v y_v c_{v,1} \ldots c_{v,n}$$

$$x_1 y_1 c_{1,1} \ldots c_{1,n}\ x_2 y_2 c_{2,1} \ldots c_{2,n}\ x_3 y_3 c_{3,1} \ldots c_{3,n}\ x_4 y_4 c_{4,1} \ldots c_{4,n} \ldots\ x_v y_v c_{v,1} \ldots c_{v,n}$$

where *n* is the number of color components per vertex and *v* is the number of vertices per row.

VerticesPerRow is a required integer value that defines the number of vertices in each row of the mesh. Although the number of vertices per row must be specified in **VerticesPerRow**, the number of rows does not need to be specified in the **Shading** dictionary.

Example 8 *Lattice-form Gouraud-shaded triangle meshes (ShadingType 5)*

```
%LATTICE1.PS
%This example illustrates lattice triangle mesh shading
%using the simplest possible example with just two triangles.
%Define graphics state and other variables
/inch {72 mul} def
...
/DeviceRGB setcolorspace
...
%Define the shading dictionary
<<
    /ShadingType 5
    /ColorSpace /DeviceRGB
    /VerticesPerRow 2
    /DataSource [
        1 inch 1 inch    1 0 1  %Magenta at bottom left
        7 inch 1 inch    0 1 1  %Cyan at bottom right
        2 inch 10 inch    0 1 1  %Cyan at top left
        8 inch 10 inch    1 0 1  %Magenta at top right
  ]
>> shfill
...
showpage
```

Note *Complete PostScript language files containing these examples accompany this document.*

For a complete list and description of the keys in the **ShadingType 5 Shading** dictionary, see Table 4.12 in the *Supplement: PostScript Language Reference Manual*.

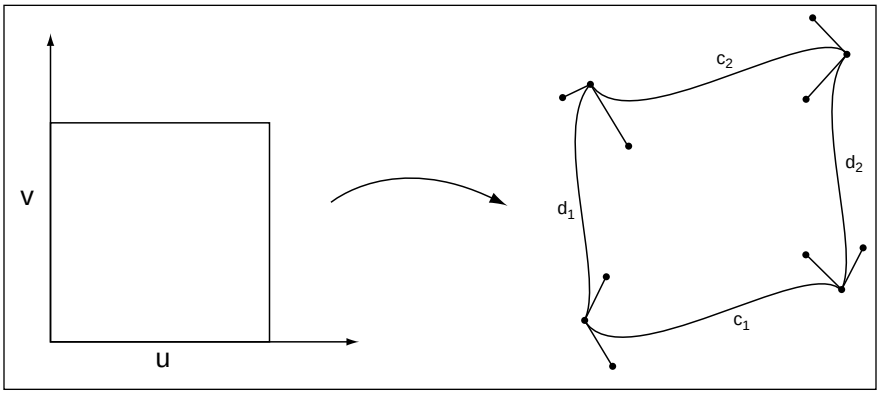
2.11 ShadingType 6: Coons patch meshes

The **ShadingType 6** shading method is used to construct one or more color patches, each bounded by four Bézier curves, that comprise what is known as a Coons patch. A primary use of this patch shading method is to allow the specification of conical vignettes and other complex gradient fills as patch meshes with nonlinear interpolation functions. A Coons patch is defined by 12 control points: four vertices plus eight Bézier control points, two for each side of the patch. The color at any one point in the patch is determined by interpolating the colors of the corner points.

A Coons patch generally has two independent aspects, a *color specification* and a *coordinate mapping*. These two aspects are defined as follows:

- Colors are specified for each of the corners of the unit square, and bilinear interpolation is used to fill in colors over the entire unit square.
- Coordinates are mapped from the unit square onto a four-sided patch whose sides are not necessarily linear. The mapping is continuous; the corners of the unit square map to corners of the patch, and the sides of the unit square map to sides of the patch (see Figure 8).

Figure 8 *Coordinate mapping from a unit square to a four-sided patch*



Thus, **ShadingType 6** shading results from coloring the unit square and then mapping it.

A bicubic Coons patch maps the unit square to a region that is bounded by four Bézier curves, c_1, c_2, d_1, and d_2.

The mathematics that describe this mapping are outlined below (refer to Figure 8 above):

Two surfaces can be described that are linear interpolations over a pair of boundary curves. Along the *u* axis, the surface S_c is described with

$$S_c(u, v) = (1 - v) * c_1(u) + (v) * c_2(u)$$

Along the *v* axis, the surface S_d is described with

$$S_d(u,v) = (1 - u) * d_1(v) + (u) * d_2(v)$$

The four corners of the Coons patch are described with

$$c_1(0) = d_1(0),\ c_1(1) = d_2(0),\ c_2(0) = d_1(1),\ \text{and}\ c_2(1) = d_2(1).$$

A third surface is the bilinear interpolation of the four corners

$$S_b(u,v) = (1 - v) * [(1 - u) * c_1(0) + (u) * c_1(1)] + (v) * [(1 - u) * c_2(0) + (u) * c_2(1)]$$

The coordinate-mapping for the shading is defined as the surface

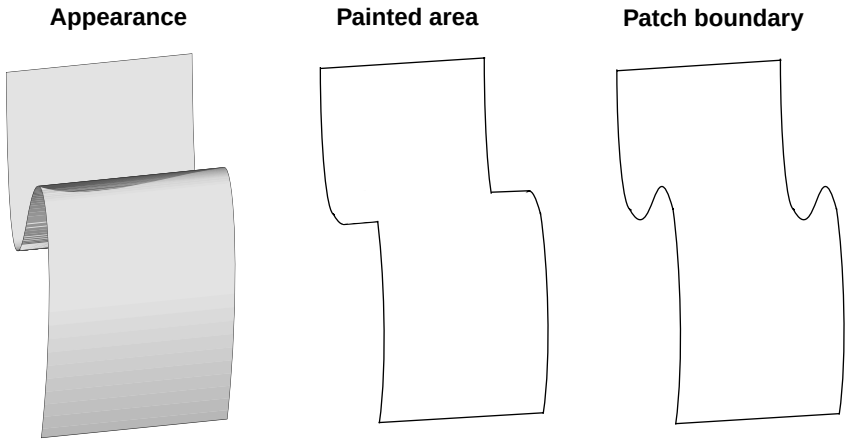
$$S = S_c + S_d - S_b$$

This defines the geometry of each Coons patch. A Coons patch mesh is constructed from a sequence of one or more such colored or shaded patches.

It is sometimes possible for a patch to appear to fold over on itself (see Figure 9). For example, a boundary curve can be self-intersecting. In this case, a fold over would occurs as follows:

Consider the above description of a mapping from *u,v* parameter space to the patch in device space. As the value of *u* or *v* increases in parameter space, the location of the pixels in device space may change direction so that pixels are mapped onto previously mapped pixels. If more than one parameter space location *(u,v)* is mapped to the same location in device space, the value of *(u,v)* selected will be the one with the largest value of *v*, and if multiple *(u,v)* values have the same *v*, the one with the largest value of *u* will be chosen.

Figure 9 *Patch appearance, painted area, and boundary*



Note *A patch is a control surface rather than a painting geometry. The outline of a projected square may not be the same as the boundary of a patch.*

If a mesh contains several patches and if some portions of one patch overlap portions of another patch, then *later* patches will paint over *earlier* patches (earlier and later refer to the order of appearance of the patches in the **DataSource** entry, which is described below).

Table 9 shows the keys that define a **Shading** dictionary for **ShadingType 6**.

Table 9 *Keys for ShadingType 6 Shading dictionaries*

Key	Type	
ShadingType	integer	required
ColorSpace	name or array	required
Background	array	optional
BBox	array	optional
AntiAlias	boolean	optional
DataSource	various	required
BitsPerCoordinate	integer	required (see note)
BitsPerComponent	integer	required (see note)
BitsPerFlag	integer	required (see note)
Decode	array	required (see note)
Function	dictionary or array	optional

Note *The **BitsPerCoordinate**, **BitsPerComponent**, **BitsPerFlag**, and **Decode** keys are required unless the value of **DataSource** is an array.*

The **ShadingType**, **ColorSpace**, **Background**, **BBox**, and **AntiAlias** keys are defined as for **ShadingType 1**.

ShadingType must be 6.

The **BitsPerComponent**, **Decode**, and **Function** keys are all defined as for **ShadingType 4**.

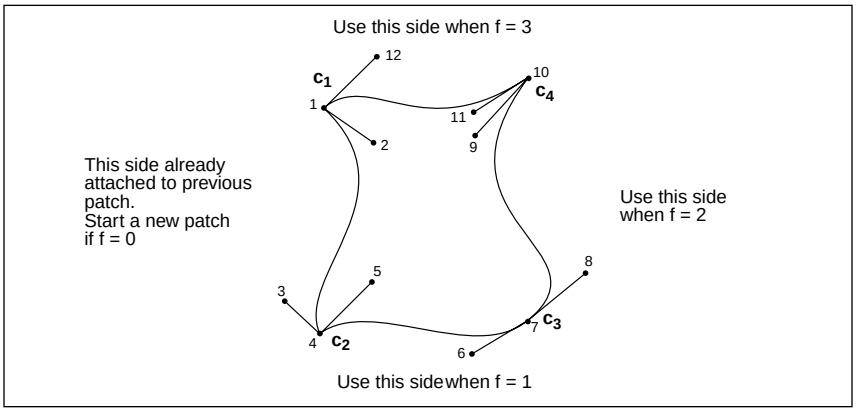
BitsPerCoordinate is an integer value that is required unless **DataSource** is an array. It specifies the number of bits used to represent each geometric coordinate. The data is decoded based on the value of **Decode**. The allowed values are 1, 2, 4, 8, 12, 16, 24, and 32.

BitsPerFlag is an integer value that is required unless **DataSource** is an array. It specifies the number of bits used to represent the edge flag for each patch. The allowed values are 2, 4, and 8, but only the least significant two bits in each flag value are used; the allowed values for the edge flag are 0, 1, 2, and 3.

The **DataSource** key provides a sequence of geometric coordinate and color component values (patch vertex/mesh data). It can be in the form of an array of numbers, a string, or a stream (see the earlier discussion for **ShadingType 4**). If the total number of bits used to represent the patch data is not divisible by eight, the patch data is padded with ignored bits inserted between the color data and the start of the next set of patch vertex data.

This data is interpreted similarly to, and has similar constraints as, the **ShadingType 4** and **5** triangle meshes, except that all of the coordinate pairs for each patch are provided first, followed by its color tuples (with the triangle meshes, the color data is supplied with each vertex). These color values are specified for the corners of the patch in the same order as the control points corresponding to the corners. Thus, c_1 is the color at (x_1, y_1), c_2 is the color at (x_4, y_4), c_3 is the color at (x_7, y_7), and c_4 is the color at (x_{10}, y_{10}), as shown in Figure 10.

Figure 10 *Color values and edge flags in Coons patch meshes*



The i^{th} patch in the data stream is represented by

$$f_i\, x_1 y_1\, x_2 y_2 \ldots x_{12} y_{12}\, c_1\, c_2\, c_3\, c_4$$

where f_i is the edge flag for the patch, all of the *xy* values are the control point coordinates, and the *c* values are the colors at the four corners of the patch.

The above figure also shows how the edge flag values (*f = 0*, *f = 1*, *f = 2*, *f = 3*) correspond to the coordinates that describe the sides of the patch. For each edge flag value, one edge from the previous patch is used as the first edge for the next patch., with the coordinates being traversed in the same direction. This arrangement improves the efficiency of the representation for meshes but complicates the data representation and stream compression, as with the triangle meshes. Therefore, since each new patch shares one edge from the previous patch, only the control points and colors defining the remaining three edges must be specified in the data stream. For each new patch other than the first, only eight control points and two corner colors must be specified:

$$f_{i+1} x_1 y_1 x_2 y_2 \ldots x_8 y_8\, c_1\, c_2$$

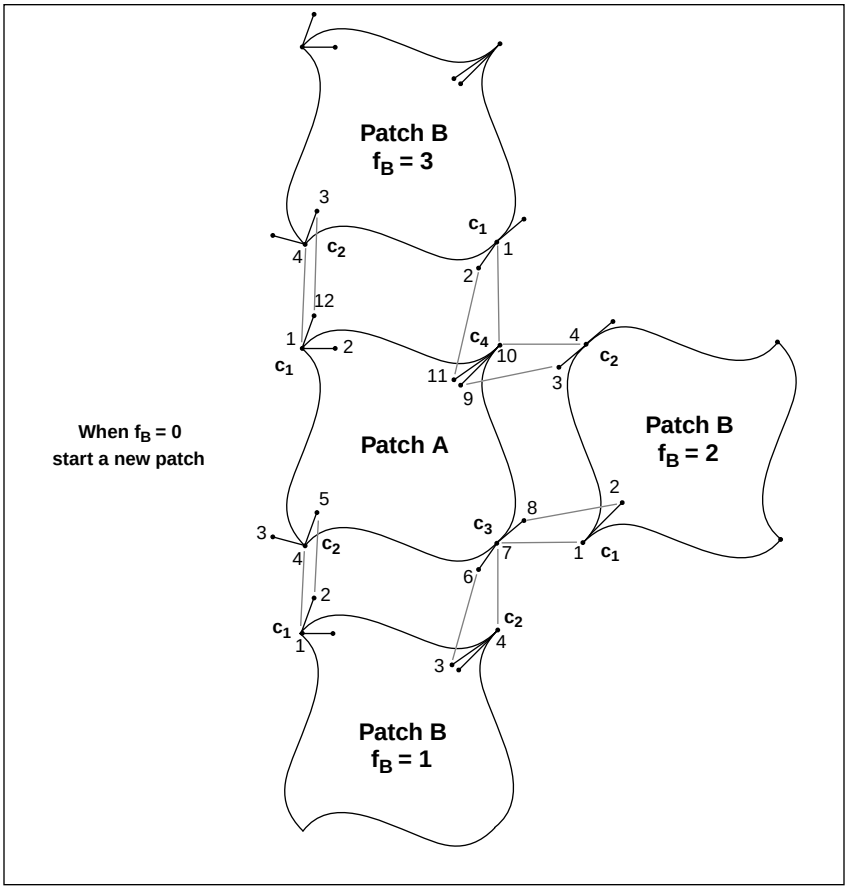
The edge flag, *f*, of the first patch must have a value of 0, which means start a new patch. The twelve control points for this patch, $x_1 y_1\, x_2 y_2 \ldots x_{12} y_{12}$, specify the Bézier curves that define the boundary curves of the patch. $c_1\, c_2\, c_3\, c_4$ represents a sequence of $4 * n$ color values, where *n* is the number of color components specified by **ColorSpace**. The edge flag for each subsequent patch can be 1, 2, or 3, depending on the desired position of the patch.

Table 10 lists the coordinates that define each adjacent patch. Figure 11 shows the adjacent patches.

Table 10 *Coordinates for adjacent patches*

Edge Flag	Next Set of Vertices
f = 0	$x_1y_1x_2y_2x_3y_3x_4y_4x_5y_5x_6y_6x_7y_7x_8y_8x_9y_9x_{10}y_{10}x_{11}y_{11}x_{12}y_{12}$ $c_1c_2c_3c_4$
f = 1	$x_5y_5x_6y_6x_7y_7x_8y_8x_9y_9x_{10}y_{10}x_{11}y_{11}x_{12}y_{12}$ c_3c_4 Implicit Values: $x_1y_1 = x_4y_4$ (of the previous patch) $c_1 = c_2$ (of the previous patch) $x_2y_2 = x_5y_5$ (of the previous patch) $c_2 = c_3$ (of the previous patch) $x_3y_3 = x_6y_6$ (of the previous patch) $x_4y_4 = x_7y_7$ (of the previous patch)
f = 2	$x_5y_5x_6y_6x_7y_7x_8y_8x_9y_9x_{10}y_{10}x_{11}y_{11}x_{12}y_{12}$ c_3c_4 Implicit Values: $x_1y_1 = x_7y_7$ (of the previous patch) $c_1 = c_3$ (of the previous patch) $x_2y_2 = x_8y_8$ (of the previous patch) $c_2 = c_4$ (of the previous patch) $x_3y_3 = x_9y_9$ (of the previous patch) $x_4y_4 = x_{10}y_{10}$ (of the previous patch)
f = 3	$x_5y_5x_6y_6x_7y_7x_8y_8x_9y_9x_{10}y_{10}x_{11}y_{11}x_{12}y_{12}$ c_3c_4 Implicit Values: $x_1y_1 = x_{10}y_{10}$ (of the previous patch) $c_1 = c_4$ (of the previous patch) $x_2y_2 = x_{11}y_{11}$ (of the previous patch) $c_2 = c_1$ (of the previous patch) $x_3y_3 = x_{12}y_{12}$ (of the previous patch) $x_4y_4 = x_1y_1$ (of the previous patch)

Figure 11 *How the value of edge flag, f, determines the edge for the next patch*



Note *The data for at least one complete patch must be specified in* ***DataSource****.*

Note *Degenerate Bézier curves are allowed and are useful for certain graphical effects. For example, a quadrant of a circle can be described by a Coons patch with one degenerate side.*

If the **Function** key value is specified, then the vertex color data for the mesh must be specified by single values *t*, rather than color *tuples c*. All linear interpolation within the triangle mesh will be done using the values of *t*, and after interpolation, the value(s) that is/are returned from **Function** will be to determine the color of each point.

Note *Using* ***ShadingType 6*** *differs from using an* ***Indexed*** *color space for the shading. If an* ***Indexed*** *color space is used, the vertex coordinates are converted to the base color space first, and linear interpolation occurs in that color space. Thus, there is no opportunity to effect a nonlinear interpolation using an* ***Indexed*** *color space.*

Example 9 *Coons patch meshes (ShadingType 6)*

```
%CONICAL.PS
%This example illustrates ShadingType 6 and the use of
%four Coons patches to create a “conical blend”.
%Define graphics state and other variables
/inch {72 mul} def
%Define the variables CX, CY, R, R3, R6, C, X3, X6, Y3, Y6
%Define startcolor, midcolor, and endcolor
...
% Define the shading and function dictionaries
gsave
<<
    /ShadingType 6
    /ColorSpace /DeviceRGB
    /DataSource
    [
      % patch 1 data
      ...
      % patch 2 data
      ...
      % patch 3 data
      ...
      % patch 4 data
      ...
    ]
    >>
shfill
grestore
showpage
```

Note *Complete PostScript language files containing these examples accompany this document.*

For a complete list and description of the keys in the **ShadingType 6 Shading** dictionary, see Table 4.13 in the *Supplement: PostScript Language Reference Manual*.

2.12 ShadingType 7: Tensor Product Patch Meshes

The **ShadingType 7** shading method is almost identical to **ShadingType 6**, except that instead of using a bicubic Coons patch defined by twelve control points, a bicubic tensor product patch defined by sixteen control points is used. The extra control points allow for more control of the color interpolation across the patch. Each set of twelve coordinate pairs in the **DataSource** key (see below) is replaced by a set of sixteen coordinate pairs.

As with the Coons patch surface, the tensor product surface is defined by a mathematical mapping from a square patch *(u,v)* to the patch coordinate system *(x, y)*.

This is described as follows:

$$S(u, v) = \sum_{i=0}^{3} \sum_{j=0}^{3} Pij \times Bi(u) \times Bj(v)$$

Pij is the control point for the *i,j* row and column of the tensor.

Since each *Pij = (Xij, Yij)*, the surface can also be expressed as

$$x(u, v) = \sum_{i=0}^{3} \sum_{j=0}^{3} Xij \times Bi(u) \times Bj(v)$$

$$y(u, v) = \sum_{i=0}^{3} \sum_{j=0}^{3} Yij \times Bi(u) \times Bj(v)$$

Bi(u) and *Bj(v)* are Bernstein polynomials, where

$$B0(t) = (1-t)^3$$

$$B1(t) = 3t(1-t)^2$$

$$B2(t) = 3t^2(1-t)$$

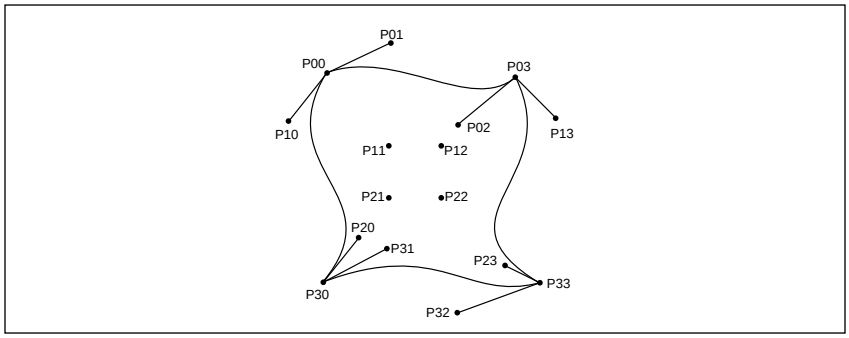
$$B3(t) = t^3$$

The control points *Pij* are defined as follows:

P00	P01	P02	P03
P10	P11	P12	P13
P20	P21	P22	P23
P30	P31	P32	P33

This is shown graphically in Figure 12.

Figure 12 *Pij control points*



This is a convenient numbering scheme for the mathematical description, but a better numbering scheme for the language is as follows:

0	11	10	9
1	12	15	8
2	13	14	7
3	4	5	6

This allows the Coons patch numbering to be a subset of the Tensor Product patch numbering.

The tensor product patch mapping, like the Coons patch mapping, is controlled by the location and shape of the four boundary curves. Unlike the Coons patch, however, the tensor product patch has four more internal control points to adjust the mapping. Each control point follows a trajectory defined by the four control points along a row or a column. Each row or column of control points defines its own cubic Bézier curve, and this is the trajectory each of the control points of the moving curve take. The tensor product patch gives more control over mapping than does the Coons patch. However, the Coons patch is easier to use and more concise because the internal control points are implicitly specified by the boundary control points.

Table 11 shows the keys that define a **Shading** dictionary for **ShadingType 7**. The keys are described in more detail, below.

Table 11 *Keys for ShadingType 7 Shading dictionaries*

Key	Type	
ShadingType	integer	required
ColorSpace	name or array	required
Background	array	optional
BBox	array	optional
AntiAlias	boolean	optional
DataSource	various	required
BitsPerCoordinate	integer	required (see note)
BitsPerComponent	integer	required (see note)
BitsPerFlag	integer	required (see note)
Decode	array	required (see note)
Function	dictionary or array	optional

Note *The **BitsPerCoordinate**, **BitsPerComponent**, **BitsPerFlag**, and **Decode** keys are required unless the value of **DataSource** is an array.*

All of the keys are defined as for **ShadingType 6**. **ShadingType** must be 7. The only difference is that each set of 12 coordinate pairs in the **DataSource** key is replaced with a set of 16 coordinate pairs.

See the earlier discussion of **DataSource** for **ShadingType 4** and **6.**

Example 10 *Tensor Product patch meshes (ShadingType 7)*

```
%TENSOR.PS
%This example demonstrates smooth shading using Tensor
%Patches.
%Define graphics state and other variables.
/DeviceRGB setcolorspace
/inch {72 mul} def
...
%Define the variables LowerLeft, CP01, CP02, LowerRight,
%CP03, CP04, UpperRight, CP05, CP06, UpperLeft, CP07, CP08
...
<<
    /ShadingType 7
    /ColorSpace /DeviceRGB
    /DataSource [
        0%start a new patch
        LowerLeft CP01 CP02
        LowerRight CP03 CP04
        UpperRight CP05 CP06
        UpperLeft CP07 CP08
        LowerLeft 3 {2 copy} repeat
        0 1 1 0.5 0 1 0.75 0 0 1 1 0.5
  ]
>> shfill
...
showpage
```

Note *Complete PostScript language files containing these examples accompany this document.*

2.13 Functions

Function objects are tightly associated with smooth shading, since functions may be used to provide close control over the shape of the color transitions across the geometry of the shading.

Functions may be thought of as "*m*-in, *n*-out" numerical transformations. Each function dictionary implicitly declares the sizes of *m* and *n*, and explicitly declares a domain of input values for which the function is defined and a range (of output values outside of which no result value will fall). Domain and range intervals must be bounded and rectangular in the input or output space of the function. They are assumed to be *closed* in the mathematical sense; that is, the edges of the interval are included in the interval, as in [0,1]. The function must be defined (but not necessarily continuous or smooth) across its entire domain. If a function is called with input values outside the declared domain, the inputs will be clipped to that domain. If any input in the declared domain of the function would cause the function to output a value outside the declared range, that output value is clipped to that range.

Each **Shading** dictionary that uses a function object must specify how it uses the function and how it maps the **Shading** domain into the domain of the function. If the output of the function is modified by the **Shading** dictionary before use, this modification must also be specified. **Shading** dictionaries that use functions must note that the declared domain of the function may be smaller than the actual domain of the function, and the declared range may be larger than the actual range of the function. Because of this, it is usually necessary to selectively specify the function so that its domain and range are appropriate for use in the **Shading** dictionary.

Three types of functions are supported in LanguageLevel 3. They are as follows:

- Sampled functions: these are the most general type of function. The mapping from input to output is controlled via a sample table that can be used to approximate any desired mathematical function. Sampled functions have been used to approximate logarithmic, sinusoidal, and Gaussian functions, to name just a few.

- Exponential interpolation functions: these are conceptually the simplest type of function. These can be used wherever a simple linear or exponential gradient fill is required.

- Stitching functions: these are used to stitch or join together the output of two or more other types of functions.

Note *Functions with high spatial frequency (or discontinuous) color transitions may display aliasing effects when imaged at low effective resolutions.*

2.14 Function Dictionaries

Each class of a **Function** dictionary has a **FunctionType** key whose value specifies the representation of the function, a set of keys that parameterize the representation, and additional data needed by that representation.

All **Function** dictionaries share the following keys: **FunctionType**, **Domain**, and **Range**. **FunctionType** and **Domain** are required for all **Function** dictionaries. The **Range** key is required only for **FunctionType 0 Function** dictionaries.

In addition, each type of **Function** dictionary must include keys appropriate, or unique, to the function type. The output *dimensionality* (range) of a function can usually be deduced from other keys of the function; if not, the **Range** key is required. The dimensionality of the function inferred from the **Domain** and **Range** declarations must be consistent with the dimensionality inferred from other keys of the function.

The **Domain** value of a **Function** dictionary must be a superset of the **Domain** value of its associated **Shading** dictionary.

Each of the three **Function** types are supported within all **Shading** dictionaries with the following exception: **ShadingType 1** dictionaries only support **FunctionType 0 Function** dictionaries.

Note *There is a new implicit resource category called **FunctionType**. Currently, the only supported instances of this category type are 0, 2, and 3, corresponding to the **Function** types discussed in Section 2.13 through 2.17.*

For a complete list and description of the keys in **Function** dictionaries, see Table 3.14 in the *Supplement: PostScript Language Reference Manual*.

2.15 FunctionType 0: Sampled Functions

Sampled functions use a sequence of sample values to provide an approximation for functions whose domains and ranges are bounded. The samples are organized in a table or array. The dimensionality of the sample table or array is equal to the dimensionality of the input domain. Samples may have more than one component. The number of components in each sample is equal to the dimensionality of the output range.

Sampled functions are highly general and offer reasonably accurate representations of arbitrary analytic functions at a low expense. For example, a single-input (*m* equal to 1) sinusoidal function can be represented over the range [0 180] with an average error of only 1%, using just ten samples and linear interpolation (**Order** equals 1). Two-input functions will take

significantly more samples, but usually not a prohibitive number, as long as the function does not have *high-frequency variations* (when the sample values vary greatly over different locations).

The dimensionality of a sampled function is restricted only by implementation limits. However, the number of samples required to represent high-dimensionality functions multiplies very rapidly unless the sampling resolution is very low; also note that the process of multilinear interpolation becomes computationally intensive if the input dimensionality is greater than two. The multidimensional spline interpolation is even more computationally intensive.

Note *Functions are assumed to be reusable; therefore, the internal representation of a sampled function must fit entirely within system memory, or a* ***VMerror*** *will occur. This limit is dependent on the amount of available system memory.*

Table 12 shows the keys that define a **Function** dictionary for **FunctionType 0**. The keys are described in more detail below.

Table 12 *Keys for FunctionType 0 Function dictionaries*

Key	Type	
FunctionType	integer	required
Domain	array	required
Range	array	optional
Order	integer	optional
DataSource	various	required
BitsPerSample	integer	required
Encode	array	optional
Decode	array	optional
Size	array	required

The **FunctionType**, **Domain**, and **Range** keys are common to all three **Function** types. **FunctionType** and **Domain** are required keys. Range is required for **FunctionType 0**, only.

FunctionType in an integer value specifying the **Function** type, which is, in this case, 0.

Domain is an array of numbers, interpreted in pairs. Each pair of numbers defines the domain of one input value. The smaller bound must precede the larger bound in each pair. The size of the array implicitly defines the input

dimensionality *m* of an *m*-in *n*-out function; this is true because *m* represents one-half of the number of elements in the array. Input values that are outside the declared domain are clipped to the nearest boundary value.

The **Range** key specifies an array of numbers that are also interpreted in pairs. Each pair defines the range of one output value. The smaller bound must precede the larger bound in each pair. The size of the array implicitly defines the output dimensionality *n* of an *m*-in *n*-out function; *n* represents one-half of the number of elements in the array. Output values are clipped to the defined range. If the range is not defined, no clipping will be performed.

Order is an optional integer value that specifies the order of interpolation between samples. The value 1, which is the default value, specifies a linear interpolation. The value 3 specifies a cubic spline interpolation.

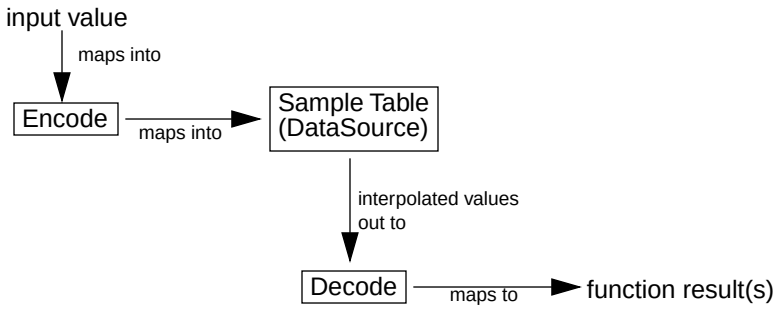
DataSource is a required key that may be a string or a reusable stream. It provides the sequence of sample values that specifies the function. If the amount of sampled data is greater than 64Kb, a reusable stream must be used (See Section 3.3.7 of the *Supplement: PostScript Language Reference Manual*).

The **BitsPerSample** key is a required integer value that specifies the number of bits used to represent each sample value. The values are 1, 2, 4, 8, 12, 16, 24, and 32.

Encode is an optional array that specifies the linear mapping of input values into the domain of the sample table for the function. The default value is as follows: [0 ($Size_0$ - 1)...].

Decode is an optional array that specifies the linear mapping of sample values into the range of values appropriate for the output variables of the function. The default value is the same as for **Range**.

Figure 13 *Mapping input values to function results (output values)*



Size is a required array that specifies the number of samples in each input dimension of the sample table.

The **Domain**, **Encode**, and **Size** keys determine how the input variable values of the function will be mapped into the sample table. For example, if the **Domain** is [-1 1 -1 1] and the **Size** is [21 31], the default **Encode** is [0 20 0 30], which maps the entire **Domain** into the full set of sample table entries. Other values of **Encode** may be used.

In general, for the i^{th} input variable d_i, the corresponding encoded value e_i, is

$$e_i = (d_i - D_{2i}) \times \frac{(E_{2i+1} - E_{2i})}{(D_{2i+1} - D_{2i})} + E_{2i}$$

where D_i and E_i are elements of the **Domain** and **Encode** arrays, respectively. If a resultant encoded value e_i falls outside the domain [0, $Size_n$ - 1], the value is clipped to the nearest allowed value. The encoded input values are real numbers, not restricted to integers, and multi-variable interpolation is used to determine an output value from the surrounding nearest-match sample table values.

Similarly, the **Range, Decode**, and **BitsPerSample** keys determine how the sample values of the function are mapped into output values. This is essentially identical to the way **image** sample values are decoded. The value of **BitsPerSample** implies that all sample values must be in the range [0 ($2^{BitsPerSample}$ - 1)]. This range is linearly transformed by the **Decode** array to an output range. The default **Decode** array is equal to the **Range** array, indicating a mapping of the entire possible sample range into the entire possible output range. Other values of **Decode** may be used.

In general, for the i^{th} sample component s_i, the corresponding output value r_i, is

$$r_i = s_i \times \frac{(D_{2i+1} - D_{2i})}{(2^{BitPerSample} - 1)} + D_{2i}$$

where D_i are elements of the **Decode** array.

As was mentioned previously, samples are encoded and interpreted similarly to **image** samples, except that function sample data for a new row must continue to be packed with the previous row and need not necessarily start on a byte boundary. No row padding is done with sampled function data. As with image data, a sequence of samples is considered to represent an array in which the first dimension of the array varies fastest; that is, in a two-dimensional array of data, the *x* component varies faster than the *y* component.

Consider the same sampled function with 4-bit samples in an array containing 21 columns and 31 rows, and consider using this function to represent a halftone spot function. A spot function takes two arguments, *x* and *y*, in the domain [-1 1], and returns one value, *z*, in the range [-1 1]. In the **Function** dictionary, the value of **Domain** would be [-1 1 -1 1], the value of **Size** would be [21 31], and the value of **Encode** would be [0 20 0 30]. The

value of **BitsPerSample** would be 4, the value of **Range** would be [-1 1], and the value of **Decode** would be [-1 1]. The *x* argument would be linearly transformed by the encoding to the domain [0 20] and the *y* argument to the domain [0 30]. Using bilinear interpolation between sample points, the function computes a value for *z*, which will be in the range [0 15], and the decoding transforms *z* to a number in the range [-1 1] for the result. The sample array is stored in a stream of

*326 bytes = [31 rows * 21 samples/row * 4 bits/sample / 8 bits/byte]*.

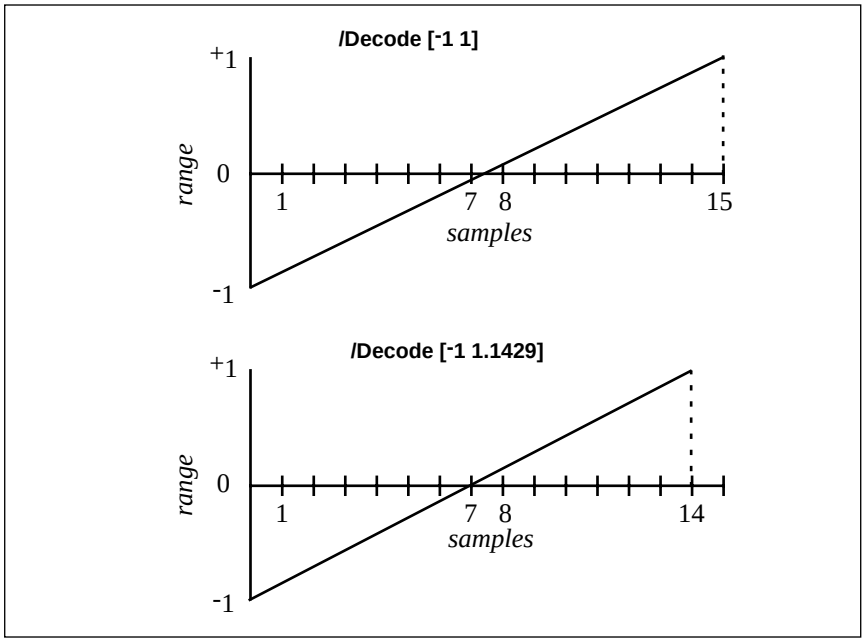
The first byte contains the sample for the point (-1, -1) in the high-order 4 bits of the byte and the sample for the point (-0.9, -1) in the low-order 4 bits of the byte.

The **Encode** key gives the linear mapping between the keys **Decode** and **Size**. The default value of **Encode** is [0 (s_0 - 1) 0 (s_1 - 1)], where s_i is the i^{th} value in the **Size** array. A non-default encoding can be specified, but the beginning and ending points for the encoding must be contained between *0* and (s_i - 1).

The **Decode** key may be used to increase the accuracy of encoded samples corresponding to certain values in the range. For example, if the desired range of the function is [-1 1] and the value of **BitsPerSample** is 4, the usual value of **Decode** would be [-1 1], and the sample values would be integers in the interval [0 15]. But if these values are used, the midpoint of the range of the function (0) would not be represented exactly by any sample value, since it would fall halfway between 7 and 8. Instead, one could use a **Decode** array of [-1 +1.1428571] and sample values in the interval [0 14]. In this way, the desired effective range of [-1 1] would be achieved, and the range value 0 would be precisely represented by the sample value 7. This example is illustrated in Figure 14.

The value of the **Size** of an input dimension can be 1, in which case all input values in that dimension will be mapped to the single allowed value. If the **Size** of an input dimension is less than 4, cubic spline interpolation is not possible, so if **Order** 3 is specified, it is ignored.

Figure 14 *Mapping with the Decode Array*



Example 11 *Sampled function (FunctionType 0)*

```
%AXSHLOG.PS
<<
    /FunctionType 0
    /Order 1
    /BitsPerSample 16
    /DataSource
      < 0000 0000 0000
        4D10 4D10 4D10
        ...
        F448 F448 F448
        FFFF FFFF FFFF >
    /Domain [0 1]
    /Decode [1 0 0 1 1 0.5]
    /Range   [0 1 0 1 0 1]
    /Size [10]
>>
```

Note *Complete PostScript language files containing these examples accompany this document.*

For a complete list and description of the keys in the **FunctionType 0 Function** dictionary, see Table 3.15 in the *Supplement: PostScript Language Reference Manual*.

2.16 FunctionType 2: Exponential Interpolation Function

Exponential interpolation is conceptually the simplest function type of the three types supported in LanguageLevel 3. **FunctionType 2** functions are always 1-in (*m* equals 1), *n*-out, defining an exponential interpolation in one variable. In the simplest case, with the exponent equal to one, the function defines a linear interpolation over its input domain.

Table 13 shows the keys that define a **Function** dictionary for **FunctionType 2**. The keys are described in more detail, below.

Table 13 *Keys for FunctionType 2 Function dictionaries*

Key	Type	
FunctionType	integer	required
Domain	array	required
Range	array	optional
C0	number or array	optional
C1	number of array	optional
N	number	required

The **FunctionType**, **Domain**, and **Range** keys are defined as for **FunctionType 0**.

FunctionType must be 2.

The mapping of the input value of the function to its output value(s) is determined by the three keys **C0**, **C1**, and **N**.

The **C0** key is an optional number or array that defines the function result (output value) for an input value of 0. It must be the same size as **C1**. The size of the function is *n*-out, where *n* is the size of the array. The default value is 0.

The **C1** key is an optional number array that defines the function result (output value) for an input value of 1. It must be the same size as **C0**. The default value is 1.

N is a required number that defines the interpolation exponent (to which the input variable is raised). Each input value *t* to the function will return the value specified by

$$c_0 + t^N (c_1 - c_0)$$

Values of **Domain** must constrain *t* such that, if *N* is not an integer, all values of *t* must be greater than or equal to zero, and if *N* is negative, no value of *t* may be zero.

For typical use as an exponential interpolation function, the value of **Domain** will be declared as [0 1], and the value of *N* will be a number greater than 0. The **Range** key may be used to clip the output to a desired range.

Example 12 *Exponential Interpolation function (FunctionType 2)*

```
%AXSHO1.PS
<<
    /FunctionType 2
    /Domain [0 1]
    /C0      0       % result for input 0 = black
    /C1      1       % result for input 1 = white
    /N       1       % Exponent = linear
>>
```

Note *Complete PostScript language files containing these examples accompany this document.*

For a complete list and description of the keys in the **FunctionType 2 Function** dictionary, see Table 3.16 in the *Supplement: PostScript Language Reference Manual*.

2.17 FunctionType 3: 1-Input Stitching Function

Stitching Functions join or stitch the outputs of two or more separate function domains across a single domain. That is, **FunctionType 3** functions define a stitching of the subdomains of several one-input functions (*m* equals 1) to produce a single, new one-input function. One example use of this function would be to create a rainbow by stitching together the separate bands of color. One dictionary and one function would be needed for each band of the rainbow to define the start and end colors in the gradient fill.

Stitching functions can be used to obtain what is known in some applications as a mid-linear blend, for example, a gradient fill that runs from green to red and back to green. One complex way to obtain this gradient fill would be to use a sampled function to define a color ramp that goes from 0 to 1 then back to 0. A simpler way to due this would be to use a one-input stitching function that contains two exponential interpolation functions with exponents of one (see the previous section). The first function would then define a ramp from 0 to 1, and the second function would define the ramp from 1 back to 0. By setting the **Bounds** key of the stitching function to 0.5, the end color of the first color ramp will occur in the middle of the domain.

The stitching function is designed to make it easy to combine several functions to be used within one shading, over different parts of the domain defined in the **Shading** dictionary. The same effect can be achieved by creating a separate **Shading** dictionary for each function, where the dictionaries have adjacent domains. However, since each **Shading** dictionary

would have similar keys, and because the overall effect desired is one **Shading** dictionary, it is more convenient to have a single **Shading** dictionary with a multiple function definition.

FunctionType 3 Function dictionaries provide a general mechanism to invert the domains of "*1*-in" functions.

Table 14 shows the keys that define a **Function** dictionary for **FunctionType 3**. The keys are described in more detail below.

Table 14 *Keys for FunctionType 3 Function dictionaries*

Key	Type	
FunctionType	integer	required
Domain	array	required
Range	array	optional
Functions	array	required
Bounds	array	required
Encode	array	required

The **FunctionType**, **Domain**, and **Range** keys are defined the same as with **FunctionType 0**.

FunctionType must be 3.

Functions is a required array of one-in function dictionaries making up the stitching function. Output dimensionality of all functions must be compatible with the value of **Range**.

Bounds is a required array of numbers, the size of which must be one less than the size of the **Functions** array. Elements of this array must be in order of increasing magnitude, and each element must be within the value of **Domain**. The **Bounds** and **Domain** keys define the intervals for which each function from the **Functions** array is used to determine the value of the stitched function. Each interval is mapped through the **Encode** array into the domain of the corresponding function.

The **Encode** array is also required and must be and array of numbers. The size of this array must be twice the size of the **Functions** array. A pair of **Encode** array values is associate with each function. The values map each subset of the domain defined by **Domain** and the **Bounds** array to the domain of the corresponding function.

An input *d* to the stitching function in the subdomain

$$B_{2i-1} \le d \le B_{2i}$$

will be encoded as follows:

$$e = (d - B_{2i-1}) \times \frac{E_{2i+1} - E_{2i}}{B_{2i} - B_{2i-1}} + E_{2i}$$

where B_i and E_i are elements of the **Bounds** array and **Encode** array, respectively, and the resulting value *e* is routed as input to the i^{th} function in the **Function** array. This is similar to the **Encode** definition in the sampled function description. For these purposes, B_{-1} is considered to be the first element of the **Domain** array, and B_n (where *n* is the number of subdomains) is considered to be the second element of the **Domain** array. The subdomain mappings may be inverted by allowing E_{2i+1} to be less than E_{2i}.

Example 13 *Stitching function (FunctionType 3)*

```
%AXSTITCH.PS
%This is a very simple illustration of axial shading
%using a stitching function with two exponential
%interpolation shading functions.
%Set up graphics state and other variables
/inch {72 mul} def
...
%Define the two exponential shading functions
/Function1 7 dict def Function1 begin
 /FunctionType 2    def
 /Domain [0 1] def
 /C0      [1 0 1] def
 /C1      [1 1 0] def
 /N       1         def
end
/Function2 7 dict def Function2 begin
 /FunctionType 2    def
 /Domain [0 1] def
 /C0      [1 1 0] def
 /C1      [0 1 1] def
 /N       1         def
end
...
gsave
    rectclip
    newpath
% define shading dictionary and stitching function
    <<
        /ShadingType 2
        /ColorSpace /DeviceRGB
        /Coords [1 inch 1 inch 7.5 inch 10 inch]
        /Function <<
            /FunctionType 3
            /Functions [Function1 Function2]
            /Domain [0 1]
            /Bounds [0.5]
            /Encode [0 1 0 1]
        >>
    >> shfill
grestore
...
showpage
```

Note *Complete PostScript language files containing these examples accompany this document.*

For a complete list and description of the keys in the **FunctionType 3 Function** dictionary, see Table 3.17 in the *Supplement: PostScript Language Reference Manual*.

2.18 Currentsmoothness and Setsmoothness Operators

The **currentsmoothness** operator returns the current value of the smoothness parameter in the graphics state. The returned value is in the range [0,1].

The **setsmoothness** operator is used to set the smoothness parameter in the graphics state. It takes as input an integer or real value in the range [0,1]. This operator is used to control the quality of smooth shaded output, indirectly affecting rendering performance. The trade-off of quality and performance depends on the value: a higher (larger) value will result in less smoothness but better performance, a lower (smaller) value will result in more smoothness but a slower or lesser performance.

Smoothness, in this context, is defined as the allowed color error between the following: smooth shading that is approximated with piecewise linear interpolation and the true shading of a linear or non-linear shading function.

The error is measured for each color component, a comparison is then made, and the maximum error value is used. Each error values is specified as a percentage of the range of its associated color component. The percentage is expressed as a value in the range [0,1].

For example, a value of 0.1 represents an allowed error of 10% for each color component.

Smoothness is dependent on several factors, including the number of displayable or printable colors, the resolution of the screen or print device, and the acceptable level of performance for the device.

See Chapter 8 of the *Supplement: PostScript Language Reference Manual* for more information on the smoothness operators.

3 Smooth Shading Tips

The following is a list of tips that can be used by developers who are interested in implementing smooth shading. In addition, the examples given in this document, and the accompanying sample files, can be used as guides for implementing each smooth shading type or method.

Pattern dictionaries vs. shfill

- Use a **Pattern** dictionary to fill paths.
- Use **shfill** for creating shading geometries.

PatternType 1 pattern dictionaries using shfill in PaintProc

To produce *tiling* patterns (repeated patterns), **shfill** can be called from within the **PaintProc** of a **Type 1 Pattern** dictionary.

Using the smoothness operators

Although these operators are available to applications developers, they are more useful to printing device manufacturers and developers for balancing performance and quality issues with the product.

Best uses for each function type

Exponential interpolation functions are best suited for the axial and radial shading methods (**ShadingType 2** and **3**) used by draw and illustration applications.

Best uses of each smooth shading type

- Function-based shading (**ShadingType 1**) can be used to generate objects such as an RGB color cube. The color cube can be implemented with three shading dictionaries that use sampled functions. The sample tables in the sampled functions only use one-bit data to specify the color components at the corners of the cube.

- Axial and radial shading methods (**ShadingType 2** and **3**) are most commonly used for creating gradient fills in draw and illustration applications.

- Axial shading is good for gradient fills that vary smoothly (either linearly or exponentially) from one point to the next.

- Radial shading is most commonly used to generate the illusion of spherical, conical, and cylindrical shapes or objects.

- Triangle meshes (**ShadingType 4** and **5**) can be used for creating polygonal gradient fills.

- Coons patch meshes (**ShadingType 6** and **7**) can be used for creating *conical* gradient fills. A conical gradient fill is a color fan that revolves around some point (not necessarily a central point). The color fan can be implemented with four Coons patches, one for each quadrant.

Smooth shading and compression

All of the stream (file) data can be compressed by using a standard compression filter. This applies to the **DataSource** keys for **ShadingType 4**, **5**, **6**, **7**, and **FunctionType 0** and **3** (indirectly). This does not apply if the data is in the form of an array or string.

Adobe Systems Incorporated

Appendix A

Bibliography of Outside Sources

While this in not an exhaustive list of references, it will give the reader some sources for the mathematical concepts covered in this document.

Farin, Gerald, *Curves and Surfaces for CAGD, Third Edition*, Academic Press, Inc. Harcourt Brace Jovanovich, Publishers, 1993. ISBN 0-12-249052-5. Chapter 16 covers information on Tensor product patches. Chapter 20 covers information on Coons patches.

Foley, J. and A. van Dam, *Fundamentals of Interactive Computer Graphics, Second Edition*, Addison-Wesley, 1982. ISBN 0-201-14468-9. Chapter 11 covers information on Parabolic Bicubic surfaces. Chpater 16, Section 2.4, covers information on Gouraud shading.

Walberg, George, *Digital Image Warping, Third Edition*, IEEE Computer Society Press, 1994. ISBN 0-8166-8944-7. Chapter 5 covers information on interpolation.

Adobe Systems Incorporated

Index