Slug

User Manual

Version 7.0

Terathon Software LLC
Lincoln, California

Slug User Manual
Version 7.0

Published by Terathon Software LLC
terathon.com

Contents

Contents **3**

1 Slug Library Overview **9**

2 Typography **11**

2.1. Glyphs 11

2.2. Metrics 12

2.3. Kerning 14

2.4. Combining Marks 15

2.5. Sequence Replacement 16

2.6. Alternate Substitution 17

2.7. Transform-Based Scripts 23

2.8. Underline and Strikethrough 24

2.9. Bidirectional Text Layout 24

2.10. Paragraph Attributes 25

2.11. Text Alignment 25

2.12. Tab Spacing 25

2.13. Grid Positioning 26

3 Vector Graphics **27**

3.1. Fills 27

3.2. Strokes 28

4 Rendering **31**

4.1. Font and Album Resources 31

4.2. Building a Slug 36

4.3. Multi-Line Text 37

4.4. Custom Glyph Layout 39
4.5. Placeholders 40
4.6. Multiple Fonts 40
4.7. Text Colors 44
4.8. Color Glyph Layers 44
4.9. Optical Weight 44
4.10. Adaptive Supersampling 44
4.11. Clipping 45
4.12. Effects 45
4.13. Icons and Pictures 45
4.14. Bounding Polygons 46
4.15. Optimization 47

5 Programming Reference 49

AlbumHeader structure 50
AssembleSlug() function 51
AssembleSlugEx() function 53
BreakMultiLineText() function 55
BreakMultiLineTextEx() function 59
BreakSlug() function 61
BreakSlugEx() function 65
BuildMultiLineText() function 67
BuildMultiLineTextEx() function 70
BuildIcon() function 72
BuildPicture() function 74
BuildSlug() function 76
BuildSlugEx() function 79
BuildTruncatableSlug() function 81
BuildTruncatableSlugEx() function 84
CalculateGlyphCount() function 87
CalculateGlyphCountEx() function 89
ColorData structure 91

CompiledCharacter structure 92
CompiledGlyph structure 93
CompiledText and CompiledStorage structures 94
CompileString() function 96
CompileStringEx() function 98
CountFill() function 100
CountIcon() function 103
CountMultiLineText() function 105
CountMultiLineTextEx() function 108
CountPicture() function 110
CountSlug() function 112
CountSlugEx() function 114
CountStroke() function 116
CreateData structure 118
CreateFill() function 119
CreateStroke() function 121
ExtendedGlyphData structure 124
ExtractBandTexture() function 125
ExtractCurveTexture() function 126
ExtractFontTextures() function 127
FillData structure 128
FillWorkspace structure 129
FontBoundingBoxData structure 130
FontDecorationData structure 131
FontDesc structure 132
FontHeader structure 133
FontHeightData structure 136
FontMap structure 137
FontMetricsData structure 139
FontOutlineData structure 140
FontPolygonData structure 141
FontScriptData structure 142

GeometryBuffer structure 143
GetAlbumHeader() function 144
GetBandTextureStorageSize() function 145
GetCompactCompiledStorageSize() function 146
GetCurveTextureStorageSize() function 147
GetFontHeader() function 148
GetFontKeyData() function 149
GetFragmentShaderSourceCode() function 151
GetGlyphContourCurveCount() function 154
GetGlyphContourData() function 155
GetGlyphData() function 157
GetGlyphIndex() function 158
GetIconData() function 159
GetKernValue() function 160
GetShaderIndices() function 161
GetUnicodeCharacterFlags() function 163
GetVertexShaderSourceCode() function 164
GlyphData structure 167
GlyphRange structure 169
GraphicData structure 170
IconData structure 171
ImportIconData() function 172
ImportMulticolorIconData() function 174
LayoutData structure 176
LayoutMultiLineText() function 186
LayoutMultiLineTextEx() function 189
LayoutSlug() function 192
LayoutSlugEx() function 195
LineData structure 198
LocateSlug() function 200
LocateSlugEx() function 202

LocationData structure ... 204
MakeCompactCompiledText() function ... 206
MeasureSlug() function ... 207
MeasureSlugEx() function ... 209
PictureData structure ... 211
PlaceholderBuffer structure ... 212
PlaceholderData structure ... 213
ResolveGlyph() function ... 214
RunData structure ... 215
SetDefaultFillData() function ... 216
SetDefaultLayoutData() function ... 217
SetDefaultStrokeData() function ... 220
SlugFileHeader structure ... 221
StrokeData structure ... 223
StrokeWorkspace structure ... 225
TestData structure ... 226
TestSlug() function ... 227
TestSlugEx() function ... 230
TextureBuffer structure ... 232
Triangle structure ... 233
UpdateLayoutData() function ... 234
Vertex structure ... 235

6 Format Directives ... 237

7 Font Conversion ... 243

8 Album Creation ... 251

A Release Notes ... 255

Slug 7.0 ... 255
Slug 6.5 ... 256
Slug 6.4 ... 257
Slug 6.3 ... 258
Slug 6.2 ... 258

Slug 6.1 259
Slug 6.0 259
Slug 5.5 260
Slug 5.1 261
Slug 5.0 261
Slug 4.2 263
Slug 4.1 263
Slug 4.0 264
Slug 3.5 264
Slug 3.0 265
Slug 2.0 266

1

Slug Library Overview

Slug was created as a software library that performs complex text layout and renders high-quality, resolution-independent glyphs on the GPU. It is intended to be used by a graphics-intensive application for all of its text rendering needs, which may include drawing graphical user interfaces, rendering heads-up displays, showing debugging information, and placing text inside a 3D world or virtual environment. The original scope of the Slug Library included only text, but the same rendering technology has now been extended to arbitrary shapes defined independently from fonts, allowing Slug to render many types of general vector graphics.

The Slug Library gets its name from the history of typography. A "slug" is what typesetters used to call a full line of text cast as one piece of hot lead by a Linotype machine. Since the primary function of the library has been to lay out and render individual lines of text, the name Slug was adopted.

Slug consists of a run-time library that performs text layout and rendering inside an application, and two standalone tools that convert fonts and vector graphics to the format required by Slug. Most of this manual discusses the functionality of the run-time library. The two conversion tools are discussed in Chapter 7 and Chapter 8.

The text layout services provided by Slug calculate the positions of the glyphs that are drawn for a given string of Unicode characters. In addition to basic bounding box and advance width calculations, Slug can perform a number of typographic manipulations that include kerning, ligature replacement, combining diacritical mark positioning, glyph composition, alternate substitution, and bidirectional text layout. Some of these encompass several different capabilities, and in particular, alternate substitution includes 18 separate features. These are discussed in detail in Chapter 2.

The rendering component of the Slug run-time library draws glyphs, icons, and arbitrary filled or stroked paths on the GPU directly from outline data composed of quadratic Bézier curves to produce crisp graphics at any scale or from any perspective. There are no precomputed texture images or signed distance fields. Slug uses a unique mathematical algorithm that can achieve perfect robustness with high performance. Details about using Slug to render text and vector graphics are discussed in Chapter 4.

The font conversion tool reads the TrueType and PostScript flavors of the OpenType format, which have the `.ttf` and `.otf` file extensions, respectively. (Font collections having the `.ttc` or `.otc` extension are also supported.) This tool generates a new file with the `.slug` extension containing all of the information necessary to render the font on the GPU. The `.slug` file includes the glyph outline data, color layer data

(if available in the original font), optimization data used by the Slug shader, and typesetting data used during text layout.

The tool used to convert vector graphics to the Slug format reads files in the Scalable Vectors Graphics (SVG) format having the `.svg` file extension and the Open Vector Graphics Exchange (OpenVEX) format having the `.ovex` file extension. This tool generates an "album" file that contains all of the curve data necessary to render the graphics on the GPU. Albums have the `.slug` extension and contain much of the same types of information that is included in font files, with the main exception being typesetting data. The graphics in an album file can be organized into a set of icons or a set of "pictures". Pictures can contain arbitrary vector graphics consisting of filled and stroked paths.

Vector graphics can also be created at run time without the step of converting an external SVG file to the Slug format. These capabilities enable the unrestricted implementation of complex user interfaces and visualizations with high performance and resolution independence.

2

Typography

Typography is the process through which a string of characters is transformed into a set of glyphs that are then displayed in some medium. At a broad level, this process involves applying fonts, sizes, and general spacing parameters that are generally specified directly by the user. At a narrower level, typography involves subtle adjustments to glyph positions, composition of ligatures or accented characters, and substitution of stylistic alternates. These fine details are usually automated to a high degree and do not directly involve the user except when various features are simply being enabled or disabled. This chapter introduces typographic terminology and discusses the standard typographic processes that are implemented in the Slug library.

All of the typographic features of Slug are controlled by the `LayoutData` structure. This structure contains information about the font size, rendering colors, text decorations, special effects, geometric transformations, and all of the layout options. This chapter makes references to many of the fields of the `LayoutData` structure, and the precise programming details are provided in Chapter 5.

2.1. Glyphs

A *glyph* is an individual shape corresponding to some legible component of written language. A glyph can simply represent a single character, such as a letter, number, or symbol, it can represent a composition of multiple characters as in the case of ligatures, or it can represent something like an accent that must be combined with another glyph to have any meaning. Every font contains a set of glyphs that is numbered independently from the set of characters that the font can display. When a string of text needs to be rendered, the characters in that text are converted into a sequence of glyphs by a complex series of steps accounting for a number of typographic transformations. The number of glyphs actually displayed does not have to match the number of characters in the original text and is often different. Once the proper set of glyphs has been determined, their rendering positions are finally calculated.

The appearance of each glyph in a font is defined by a set of one or more closed *contours*, as shown in Figure 2.1. Each contour is composed of a continuous sequence of quadratic Bézier curves that mathematically describe the exact shape of the glyph's outline. The only information needed to determine this shape is the set of control points for the Bézier curves, and this allows a glyph to be rendered with high precision at any scale. In conventional rasterizers, the contour data is used to generate glyph images one time at each specific size for which a font is displayed, and these images are then repeatedly copied to the display as needed to render text. The Slug library uses the contour data directly during the rendering process without an intermediate glyph image generation stage.

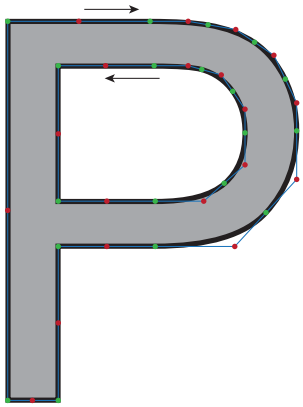

Figure 2.1. A glyph is defined by one or more closed contours composed of a set of quadratic Bézier curves. This glyph has two contours, one corresponding to the outer boundary of the filled area and another corresponding to the inner boundary of the empty area. The green dots represent the on-curve control points, and the red dots represent the off-curve control points. The blue lines are tangent to the glyph's outline.

2.2. Metrics

The position of each control point belonging to a glyph is expressed relative to a box known as the *em square*, and the coordinates of the control points are expressed in *em space*. As shown in Figure 2.2, the em square corresponds to the box extending from 0 to 1 in both the x and y directions. In em space, the x axis points to the right, and the y axis points up. The origin point represents the drawing position on the baseline of the text. When text is drawn at a font size S, all of the em-space control point coordinates are uniformly scaled by a factor of S.

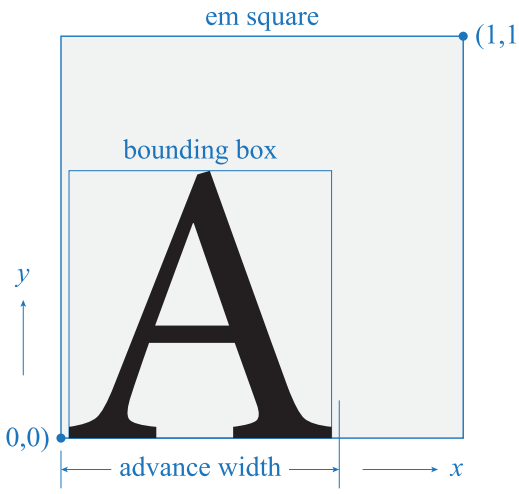

Figure 2.2. The em square extends from $(0, 0)$ to $(1, 1)$. Each glyph has a bounding box and advance width defined in em-space coordinates.

The em square contains the bulk of most glyphs, but control points are allowed to occur outside it and often do. In particular, some of the control points belonging to a glyph that descends below the baseline have negative y coordinates. Sometimes, glyphs are wider or taller than the em square and extend beyond the right and top edges. Although less common, glyphs may also extend past the left edge of the em square.

The smallest box containing every point on a glyph's outline is called its *bounding box*. Ordinarily, the left edge of the bounding box is near the left edge of the em square and slightly inside. For each glyph, a font defines an *advance width* that indicates how far to move the drawing position, after scaling by the font size, in order to render the following glyph. The advance width is usually slightly larger than the width of the bounding box, but this is not a requirement, and it may be smaller in cases when part of a glyph is intended to extend beyond the drawing position for the next glyph.

The heights of characters in a font are usually smaller than the full height of the em square. There is no standard rule, but uppercase roman letters are typically about 70% as tall as the em square. This means that the actual height of capital letters is roughly 70% of whatever font size has been specified because the font size corresponds to the physical distance between 0 and 1 in em space. So that it's possible to know how large characters will actually appear when rendered, a font defines two values called a *cap height* and an *ex height* that correspond to the ordinary heights of uppercase and lowercase letters, as illustrated in Figure 2.3. Fonts may also define independent values called the *ascent* and *descent* that correspond to the maximum typical distances above and below the baseline to which letters extend. Parts of glyphs for lowercase letters that extend beyond the ex height are called *ascenders*, and parts that extend below the baseline are called *descenders*.

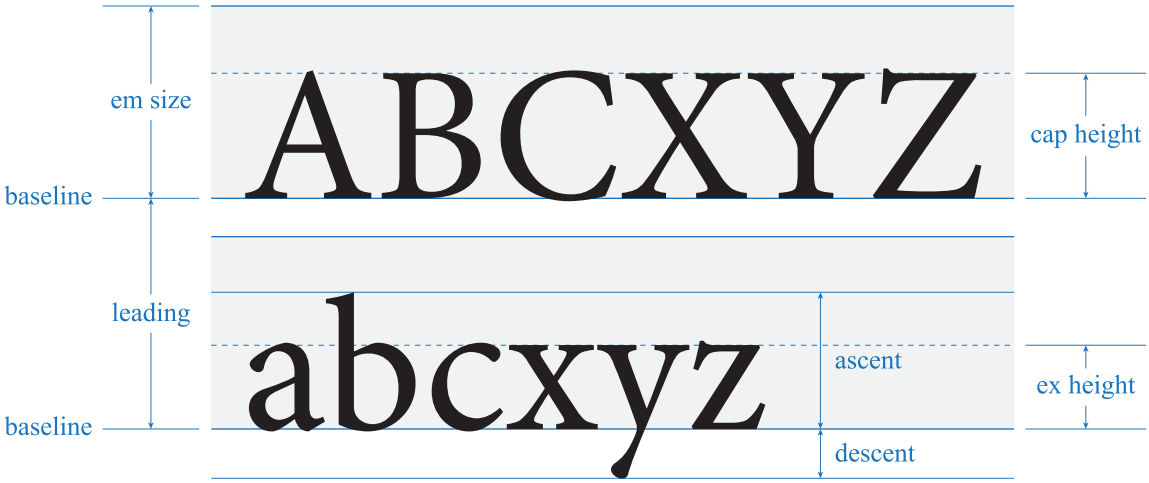

Figure 2.3. The cap height corresponds to the general height of uppercase letters, and the ex height corresponds to the general height of lowercase letters without ascenders. The leading corresponds to the vertical distance between two consecutive baselines.

When text is rendered as multiple lines, the vertical distance from one baseline to the next baseline is called the *leading* (pronounced like the element lead with -ing appended), as shown in Figure 2.3. The leading value is expressed in units of the em square and must be scaled by the font size to calculate the physical distance between lines. Leading is not defined by the font but specified by the user, and it is typically chosen to be in the range of 1.0 to 1.5 em. The default leading used by Slug is 1.2 em, and it can be changed by modifying the `textLeading` field of the `LayoutData` structure.

The horizontal distance between adjacent glyphs is determined by the each glyph's advance width and by the kerning values discussed in the next section. The user may increase or decrease this natural distance by specifying a *tracking* value that is uniformly applied between all pairs of glyphs. As shown in Figure 2.4, positive tracking causes glyph spacing to be expanded, and negative tracking causes glyph spacing to be condensed. As with most other measurements, tracking is specified in em units so that it is independent of font size. In Slug, the amount of tracking is controlled by the `textTracking` field of the `LayoutData` structure.

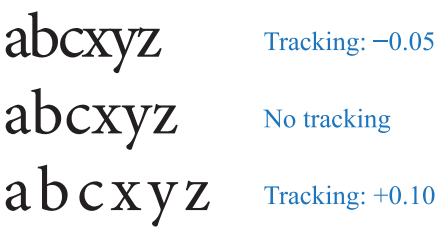

Figure 2.4. Tracking adds space to the natural distance between adjacent glyphs.

2.3. Kerning

When certain pairs of glyphs are rendered adjacently, the empty space between them can appear to be inconsistent with the typical spacing among other glyphs in the text. This happens when one or both glyphs in a pair have larger than usual regions inside their bounding boxes containing no part of their outlines. For example, a capital T has a significant amount of empty space in the lower-right corner of its bounding box, and when a T is followed by most lowercase letters, the greater size of the void can make the letter spacing within the entire word seem choppy.

Kerning is the process by which the spacing is adjusted between specific pairs of glyphs to give text a more consistent and more appealing overall distribution. In the example shown in Figure 2.5, the first line of text is laid out using only the advance widths defined for each glyph. The second line of text includes kerning, and it is most noticeable where the amount of empty space has been reduced after the capital T and capital W. Kerning can also be applied to punctuation, and this is exemplified by the adjustment to the period and closing quote in the second line. Kerning doesn't always have to reduce the amount of spacing and can sometimes be used to increase spacing in cases where glyphs may come a little too close to each other. The spacing between the opening quote and the capital T in the second line has been subtly increased to move the serif in the T away from the quote occupying the same vertical position.

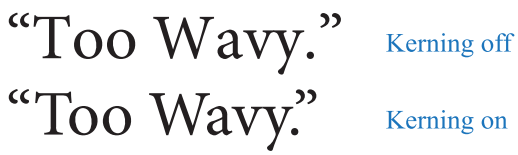

Figure 2.5. Kerning is enabled in the second line of text, and it causes the excess spacing between some pairs of glyphs to be closed up.

The data needed for kerning is stored with a font, and it is imported when a font is converted to the Slug format. Whether kerning is actually applied to a string of text can be controlled by the `layoutFlags` field of the `LayoutData` structure. Kerning is enabled by default.

2.4. Combining Marks

Unicode supports a wide variety of accents and other types of embellishments collectively called *marks* that can be combined with a *base* glyph to construct a complete character. While many common accented characters such as ä or é are usually available in a precomposed form, it would be impractical to include all possible combinations of bases and marks in a font, especially considering that multiple marks can be applied to a single base. Instead, Unicode defines some code points to be *combining*, which means that they always get attached to the nearest preceding base character.

A base character such as an uppercase or lowercase letter typically defines a group of anchor points, and a mark defines a point that must be aligned with one of those anchor points when it is attached to a base character. Which anchor point each mark gets attached to is something defined by the font, and it is not selected by the user. In Figure 2.6, one mark is attached to an anchor point at the top of a base character, and a second mark is attached to another anchor point at the bottom of the same base character. Because they are attached to different anchor points, these two marks could follow the base character in either order in the text string. For example, both the strings “a◌̆◌̥” and “a◌̥◌̆” produce the same output ḁ̆. When multiple marks are attached to the same anchor point, they stack, and each mark is actually attached to the nearest preceding mark that was attached to that anchor point.

The data needed for mark positioning is stored with a font, and it is imported when a font is converted to the Slug format. Whether marks are actually attached to base glyphs in a string of text can be controlled by the `layoutFlags` field of the `LayoutData` structure. Mark positioning is enabled by default.

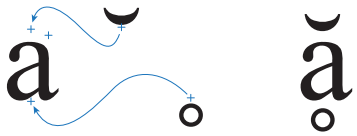

Figure 2.6. Two combining mark characters are attached to different anchor points on a base character to compose the final appearance of a letter.

2.5. Sequence Replacement

When characters are translated into glyphs, specific sequences of glyphs are identified as special groups that should be replaced by a substitute glyph. In Slug, this process is referred to as *sequence replacement*, and it includes two general substitution categories called ligatures and glyph composition.

A *ligature* is a special glyph that is drawn as a replacement for a specific sequence of two or more ordinary glyphs to improve the aesthetic appearance and readability of the text. Ligatures are often provided for pairs of characters that frequently appear next to each other and tend to touch or overlap slightly. A font may define ligatures to replace any sequences of characters, but it is most commonly done for the lowercase letter f followed by another letter f, a letter i, or a letter l. A few examples are shown in Figure 2.7.

OpenType fonts can classify ligatures as required, standard, discretionary, or historical, and these classes can be separately enabled or disabled in Slug by setting bits in the `sequenceMask` field of the `LayoutData` structure. Note that classifications for the same ligatures sometimes differ among fonts. Ligatures that are classified as standard in one font may be classified as discretionary in another font.

Figure 2.7. In the first line of text, each character is drawn as a separate glyph. In the second line, special ligature glyphs are drawn in place of the character pairs Th, fi, fl, and ft.

Glyph composition is another type of sequence replacement that combines multiple input glyphs into a single output glyph. This feature is typically used by a font to perform replacements in various non-English writing systems that must occur for text to be rendered correctly and therefore not to be considered optional. Glyph composition is enabled by default in Slug, and it can be controlled by the `kSequenceGlyphComposition` bit in the `sequenceMask` field of the `LayoutData` structure.

Aside from language-specific applications, glyph composition is also used to build specialized emoji glyphs from generic components. For example, an emoji having a specific skin tone is generated by following a generic version of the emoji with one of five skin tone modifiers beginning with Unicode character U+1F3FB. The pair of glyphs 👦🏾 composes the boy emoji with the type-4 skin tone modifier to produce the single glyph 👦🏾. More elaborate replacements are often performed by sequences that include the zero-width joiner (ZWJ) character U+200D. For example, the sequence 👩 ZWJ 🔬 composes the woman emoji, ZWJ character, and microscope emoji to produce the woman scientist emoji 👩‍🔬.

Some fonts contain special glyphs that can replace a sequence of glyphs representing a fraction. For example, the three glyphs making up the sequence “7/8” could be replaced by a single glyph that looks

like ⅞. These are called *alternative fractions*, and they are enabled by setting the `kSequenceAlternative-Fractions` bit in the `sequenceMask` field of the `LayoutData` structure.

The data needed for sequence replacement is stored with a font, and it is imported when a font is converted to the Slug format. All sequence replacement for a string of text can be controlled by the `layoutFlags` field of the `LayoutData` structure independently of the `sequenceMask` field. By default, general sequence replacement is enabled, but discretionary ligatures, historical ligatures, and alternative fractions are disabled by the `sequenceMask` field.

2.6. Alternate Substitution

OpenType fonts often contain a potentially large set of special glyphs that are not directly accessible through any Unicode character values but are instead designated as *alternates* for other glyphs. Alternate glyphs are grouped based on the type of transformation they apply to the default appearances of the glyphs they replace. For example, one group of alternates turns lowercase letters into small capitals, and another group of alternates turns numbers into subscripts. Slug supports a variety of alternate types, as shown in Table 2.1, and each one is enabled by setting a bit in the `alternateMask` field of the `LayoutData` structure.

The data needed for alternate substitution is stored with a font, and it is imported when a font is converted to the Slug format. All alternate substitution for a string of text can be controlled by the `layoutFlags` field of the `LayoutData` structure independently of the `alternateMask` field. By default, general alternate substitution is enabled, but no bits in the `alternateMask` field are set, so no substitution takes place without enabling one or more alternates in the mask.

Stylistic Variants

It is common for fonts to include *stylistic* variants, as shown in Figure 2.8. These variants can comprise as little as a set of alternate forms for a handful of glyphs or as much as several complete sets of every letter of the alphabet with increasing levels of flare. OpenType defines a group of 20 substitution features for stylistic variants, and they are accessed in Slug by specifying the `kAlternateStylistic` option and selecting a style index in the `LayoutData` structure.

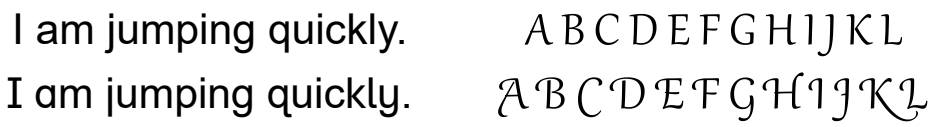

Figure 2.8. These are examples of stylistic variants. On the left, the roman letters in the Arial font are shown in their default forms on the first line and in stylistic set #3 on the second line, which makes some subtle changes to the appearance of specific glyphs. A uniform and more stylized difference is illustrated on the right by the uppercase letters in the Gabriola font, which are shown in their default forms on the first line and in stylistic set #4 on the second line.

Historical Variants

A *historical* variant is an alternate glyph whose form represents the past appearance of some character that is no longer in common use. The most common historical variant pertains to the lowercase letter s. Fonts don't typically include many historical variants, but when they do, these variants can be enabled in Slug by specifying the `kAlternateHistorical` option.

Table 2.1. These are the alternate glyph types supported by Slug. Each type of alternate is activated by setting the bit in the second column in the `alternateMask` field of the `LayoutData` structure. The four-character code shown in the third column is the OpenType feature to which each type of alternate corresponds.

Alternate type	Slug enumerant	OpenType feature
Stylistic variants	`kAlternateStylistic`	`ss01` – `ss20`
Historical variants	`kAlternateHistorical`	`hist`
Lowercase small caps	`kAlternateLowerSmallCaps`	`smcp`
Uppercase small caps	`kAlternateUpperSmallCaps`	`c2sc`
Titling caps	`kAlternateTitlingCaps`	`titl`
Unicase	`kAlternateUnicase`	`unic`
Case-sensitive forms	`kAlternateCaseForms`	`case`
Subscripts	`kAlternateSubscript`	`subs`
Superscripts	`kAlternateSuperscript`	`sups`
Scientific inferiors	`kAlternateInferiors`	`sinf`
Ordinals	`kAlternateOrdinals`	`ordn`
Lining figures	`kAlternateLiningFigures`	`lnum`
Old-style figures	`kAlternateOldstyleFigures`	`onum`
Tabular figures	`kAlternateTabularFigures`	`tnum`
Proportional figures	`kAlternateProportionalFigures`	`pnum`
Slashed zero	`kAlternateSlashedZero`	`zero`
Fractions	`kAlternateFractions`	`frac, numr, dnom`
Hyphen minus	`kAlternateHyphenMinus`	–

Small Caps

The term *small caps* refers to the technique of replacing lowercase letters with versions of the corresponding uppercase letters that are rendered at a smaller size. In the past, small caps were typically drawn by simply scaling the glyphs for uppercase letters down to a smaller size. Modern fonts include special glyphs for small caps, and as demonstrated in Figure 2.9, these come with several advantages:

- Because the special glyphs are not scaled-down versions of ordinary capitals, they don't suffer from thinning strokes that can give the appearance of a lighter weight. The small caps instead look like they fit in with the rest of the font.
- The font can include specialized kerning data for small caps glyphs. For example, a small caps glyph following an ordinary capital T can often be kerned.
- A font may provide alternate glyphs for symbols and punctuation that match the size of its small caps glyphs and substitute these variants when small caps is enabled.

OpenType defines two features for substituting small caps, one for replacing lowercase letters, and another for replacing uppercase letters. (The latter is less commonly available.) These can be enabled in Slug by specifying the `kAlternateLowerSmallCaps` and `kAlternateUpperSmallCaps` options.

If a font does not contain small caps, or if a font does not have small caps variants for certain symbols, the smaller size can still be achieved, without the above advantages, by using embedded format directives to change the scale of the glyphs. See the `scale` directive in Chapter 6.

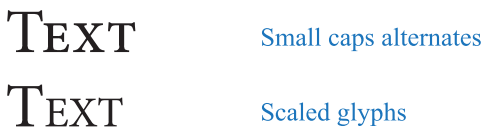

Figure 2.9. In the first line of text, small caps are rendered with specialized alternate glyphs having stroke weight and spacing designed to match the rest of the font. In the second line, small caps are simulated by scaling down ordinary capital letters, and this leads to a lighter appearance and tighter spacing.

Titling Caps

Some fonts include an extra set of capitals that are meant to be used in titles or headlines consisting only of uppercase letters. These are called *titling caps*, and they usually have slightly different weights and spacing to improve readability. If titling caps are available in a font, they can be enabled in Slug by specifying the `kAlternateTitlingCaps` option.

Unicase

Unicase is a somewhat obscure OpenType feature that causes both lowercase and uppercase letters to be transformed into a single set of glyphs consisting of a mixture of capital and small forms sharing the same overall height. An example is shown in Figure 2.10. When available, Unicase can be enabled in Slug by specifying the `kAlternateUnicase` option.

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
aBCDeFGHIJKLmnOPQRSTUVWXYZ

Figure 2.10. The unicase alternates in the Arial font apply to both uppercase and lowercase letters. Here, both cases are transformed into a single set of mixed forms having the same overall height.

Case-sensitive Forms

A font may include alternate versions of punctuation and other symbols that fit better with text written in uppercase letters, and these are called *case-sensitive forms*. Typically, the glyphs for characters such as parentheses are shifted upward a little and possibly enlarged. The hyphen character is often repositioned higher to account for the greater size of capital letters. If a font contains case-sensitive forms, they can be enabled in Slug by specifying the `kAlternateCaseForms` option.

Subscripts and Superscripts

As with small caps, there are two ways in which subscripts and superscripts can be typeset. Most fonts include alternate glyphs for subscripts and superscripts that maintain a consistent stroke weight and enable specialized kerning. The use of these glyphs is called *alternate-based scripts*, and this method should be used when possible to achieve the highest quality. Unfortunately, the set of characters covered by this feature varies widely from one font to another and is usually sparse. One font may include subscript glyphs for all letters, numbers, and symbols, and another font may include subscript glyphs only for the numbers 0–9. When alternate glyphs are not available, a different method called *transform-based scripts*, described in Section 2.7 below, must be used instead to scale and offset the glyphs.

Alternate-based subscripts and superscripts are enabled in Slug by specifying the `kAlternateSubscript` and `kAlternateSuperscript` options. In the case that subscripts or superscripts are enabled, but a font does not include the alternate glyph for a particular character, that character remains in its original form.

Scientific Inferiors and Ordinals

Separately from subscripts and superscripts, a font may define similar sets of glyphs called *scientific inferiors* and *ordinals*.

Scientific inferiors are simply subscripts that are meant to be used in text containing things like mathematical expressions or chemical formulas such as H_2O. It's possible for these to have a slightly different appearance than ordinary subscripts, but most fonts just use the same glyphs for both subscripts and scientific inferiors, which means the character coverage is usually the same. Scientific inferiors are enabled in Slug by specifying the `kAlternateInferiors` option.

Ordinals pertain to the raised letters in abbreviations such as 1st, and they may have a different size or position compared to superscripts. The character coverage for ordinals varies considerably from one font to another. In some cases, superscripts and ordinals are supplied for all lowercase letters, and they use the same set of glyphs. In other cases, no superscripts are supplied for letters, and ordinals are supplied for only the letters d, h, n, r, s, and t because those cover all possibilities in the English language. Ordinals are enabled in Slug by specifying the `kAlternateOrdinals` option.

Figure Style and Spacing

In typography, the numbers 0 through 9 are called *figures*, and they generally come in two styles called *lining* figures and *old-style* figures. The difference between the two styles is illustrated in Figure 2.11. Lining figures fill the entire space between the baseline and the cap height for the font. In contrast, the bulk of each old-style figure fills only the space between the baseline and the ex height, but some figures have pieces that extend higher or lower. The default figure style used by a font may be either lining or old-style, and it's usually the case that the other style is available through the alternate substitution mechanism.

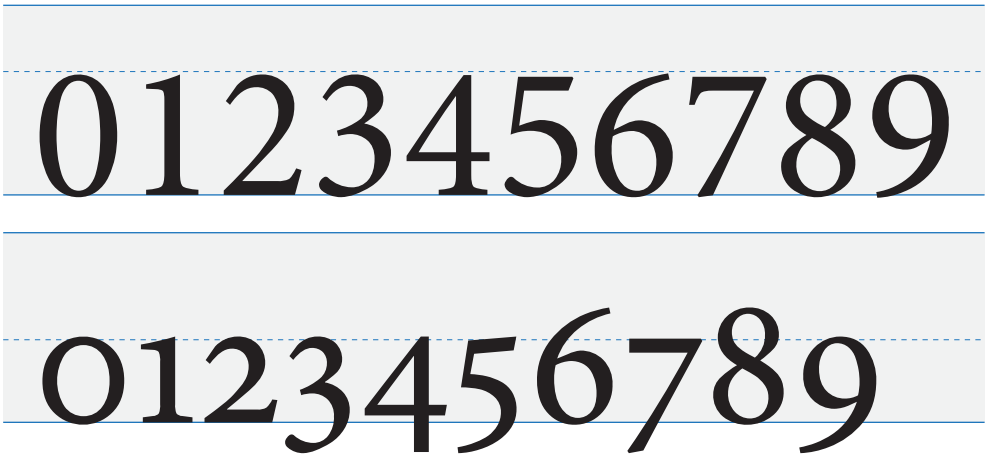

Figure 2.11. Lining figures are rendered in the first line, and they extend to nearly the cap height represented by the blue dashes. Old-style figures are rendered in the second line where the blue dashes instead represent the ex height.

In Slug, default old-style figures can be replaced by alternate lining figures by specifying the `kAlternateLiningFigures` option, and default lining figures can be replaced by alternate old-style figures with the `kAlternateOldstyleFigures` option.

Most fonts provide two spacing options for figures called *tabular* and *proportional*, and these options may be available for both lining figures and old-style figures or only for one of those styles. Tabular figures all have the same advance width and are named for their usefulness in typesetting numerical data that may appear in some kind of table, where it's necessary for each column of digits to line up.

Proportional figures have varying widths that depend on the actual shape of each glyph. As with the style option, the default spacing option may be either tabular or proportional, and the alternate substitution mechanism is used to switch from one to the other.

In Slug, default proportional figures can be replaced by alternate tabular figures by specifying the `kAlternateTabularFigures` option, and default tabular figures can be replaced by alternate proportional figures with the `kAlternateProportionalFigures` option. Note that when using tabular figures, some fonts still kern a little between two consecutive 1 characters, so it's a good idea to disable kerning as well to ensure consistent spacing.

Slashed Zero

Many fonts contain an alternate glyph for the number zero that has a slash through it. When the `kAlternateSlashedZero` option is enabled, this alternate glyph is substituted for any ordinary zero characters having Unicode value U+0030.

Fractions

If a forward slash character having Unicode value U+002F or a fraction slash character having Unicode value U+2044 has numerical digits immediately preceding it and immediately following it, Slug can convert the entire sequence into a common fraction with raised numerator digits and lowered denominator digits, as shown in Figure 2.12. When the `kAlternateFractions` option is enabled, the glyph for the slash character is replaced by a fraction slash, the glyphs for the preceding digits are replaced by numerator alternates, and the glyphs for the following digits are replaced by denominator alternates. These alternate digits are smaller in size compared to ordinary text and have positions that are different from superscripts and subscripts, typically appearing no higher than the capital height and no lower than the baseline.

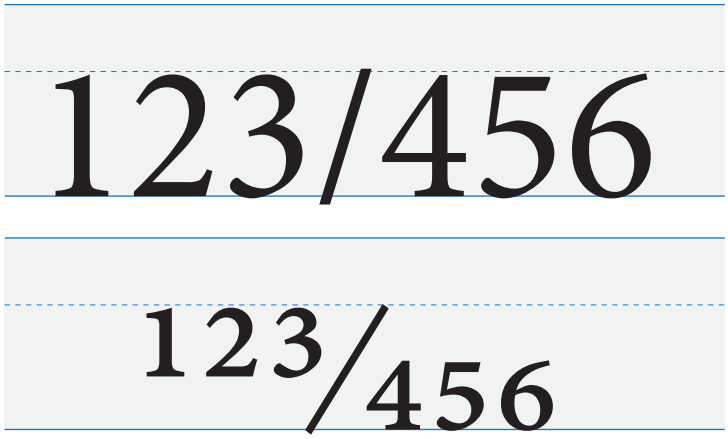

Figure 2.12. Fraction substitution replaces a sequence of digits followed by a slash and another sequence of digits in the top line by special numerator alternates, a fraction slash, and special denominator alternates in the bottom line.

Hyphen Minus

Most fonts contain a glyph for the minus character having Unicode value U+2212, but minus signs are almost universally represented by the ordinary hyphen character having Unicode value U+002D in strings generated by computer programs. Hyphen glyphs are usually shorter than and drawn below the horizontal stroke of a plus sign, and this produces an inconsistent appearance in numerical text. When the `kAlternateHyphenMinus` option is enabled, the glyph for the hyphen character is drawn as the glyph for the minus sign if it's available in the font. This provides a way to render proper minus signs without having to change any characters in a string, which would require replacing a one-byte hyphen character with a two-byte minus character encoded as UTF-8.

The hyphen minus alternate does not correspond to any OpenType feature. The functionality is specific to Slug, and it exists to provide a convenient way to nicely typeset negative numbers and subtraction expressions.

2.7. Transform-Based Scripts

Every OpenType font contains information describing how glyphs should be scaled and offset in order to render subscripts and superscripts. Because it is rare for a font to include alternate subscript and superscript glyphs for every letter, number, and symbol, the geometric transforms must instead be used whenever any script characters are not covered by those alternates. This method of producing subscripts and superscripts is called *transform-based scripts*, and it is accessible in Slug by using the embedded format directive `script()`.

The transform for a subscript or superscript consists of a scale and offset that both have x and y components. The scale resizes the glyphs, and it is typically nonuniform so that script glyphs are slightly expanded in the horizontal direction compared to their original dimensions. The offset changes the position of the glyphs, and it is primarily used to raise or lower the baseline. The x component of the offset is usually zero, but it can be nonzero in italic or oblique fonts to account for the general slant of the text.

Slug supports multiple levels of subscripts and superscripts to a maximum depth of three. At the n-th level, the script transform is simply applied n times. The `script()` format directive takes a single integer argument in the range $[-3, 3]$ that specifies either subscript or superscript and the level. When this directive is applied, it sets internal scale and offset values that function independently of the `textScale` and `textOffset` fields of the `LayoutData` structure. The directive `script(n)` enables subscript rendering for negative values of n and superscript rendering for positive values of n, and it sets the script level to the absolute value of n. The directive `script(0)` discontinues script rendering and resets the internal scale and offset values to the identity transform.

The default subscript and superscript scale and offset values provided by a font may not be exactly what the user wants, and it's not uncommon for the default transforms to be a poor match for the alternate subscript and superscript glyphs. The set of eight values (two scales and two offsets for both subscript and superscript) can be individually overridden when a `.slug` file is generated by specifying new values on the `slugfont` command line, as described in Chapter 7.

2.8. Underline and Strikethrough

Slug is able to render underline and strikethrough for any subset of the glyphs in a line of text. These are called *text decorations*, and information about their preferred position and size is included in every OpenType font. By default, a tiny amount of extra geometric data is stored inside each `.slug` file that is used to render the horizontal strokes for underline and strikethrough decorations with the same shader that is used to render glyphs. This allows those decorations to be rendered in the same draw call as the text, and it gives those decorations an appearance that's consistent with the glyphs at all scales.

Underline and strikethrough can be enabled for a line of text by setting entries of the `decorationFlag` field of the `LayoutData` structure, and they can be controlled at a character granularity by using the embedded format directives `under()` and `strike()`.

The default underline and strikethrough positions and sizes provided by a font may not be exactly what the user wants. The set of four values (a position and size for both underline and strikethrough) can be individually overridden when a `.slug` file is generated by specifying new values on the `slugfont` command line, as described in Chapter 7.

2.9. Bidirectional Text Layout

Slug is able to render text in the right-to-left writing direction for languages such as Arabic, and it can perform bidirectional layout for text containing characters using both left-to-right and right-to-left directionalities. All text has a *primary* writing direction that determines how the drawing position is normally advanced as well as how bidirectional layout behaves. By default, the primary writing direction is left-to-right, but it can be changed to right-to-left by setting the `kLayoutRightToLeft` bit in the `layoutFlags` field of the `LayoutData` structure.

Bidirectional text layout is enabled by setting the `kLayoutBidirectional` bit in the `layoutFlags` field of the `LayoutData` structure. When bidirectional text layout is enabled, Slug considers the Unicode directionality for each character to determine whether various runs of text should be laid out left-to-right or right-to-left. Characters such as spaces and punctuation are directionally neutral, and when they occur at the boundary between runs having different directionalities, those characters are associated with the runs having the primary writing direction (as determined by whether the `kLayoutRightToLeft` bit is set). This can cause punctuation to appear on wrong side of a word having the opposite writing direction, but this problem can be eliminated by following such punctuation with a left-to-right marker (LRM) or right-to-left marker (RLM) control character, as appropriate, defined by Unicode with values U+200E and U+200F, respectively.

Some Unicode characters, such as parentheses, brackets, and a variety of mathematical symbols, have the special property that they are supposed to be mirrored when they appear in right-to-left text. When bidirectional text layout is enabled, Slug automatically replaces these characters with their mirrored counterparts when they occur inside a run of right-to-left text.

2.10. Paragraph Attributes

When text is laid out as multiple lines, it can be partitioned into paragraphs by using hard break characters. Various formatting options can be applied to each paragraph by specifying values in the `LayoutData` structure and enabling paragraph attributes by setting the `kLayoutParagraphAttributes` bit in the `layoutFlags` field. These formatting options are summarized as follows.

- A paragraph can have left and right margins. These are useful for things like block quotes and lists that are offset from the surrounding text.
- The first line of each paragraph can be indented or outdented by a specific distance.
- Extra vertical space can be applied between paragraphs. This adds to the ordinary leading applied between consecutive lines.

By default, paragraph attributes are not enabled.

2.11. Text Alignment

Multi-line text can have left, right, or center alignment as specified by the `textAlignment` field of the `LayoutData` structure. The alignment determines the horizontal position of each line of text with respect to the maximum span passed to functions like `BuildMultiLineText()`. When text is aligned left, all lines start at the left margin and end wherever the next word would not fit on the line. This creates a *ragged* right margin in which the lines of text generally have different physical lengths. When text is aligned right, it's the opposite case in which the right margin is constant and the left margin is ragged. When text is aligned center, all lines are centered in between the left and right margins, and both sides are ragged.

Full justification is specified independently of the alignment by setting the `kLayoutFullJustification` flag in the `layoutFlags` field of the `LayoutData` structure. When full justification is enabled, lines of text are laid out so that they always occupy the full width of the maximum span. In this case, both the left and right margins are constant, and neither side is ragged. Full justification is accomplished by expanding the width of the space characters so that each line is exactly the desired width. The last line in each paragraph is not fully justified, but instead the alignment specified by the `textAlignment` field is applied.

The set of Unicode characters that should be treated as spaces for full justification is specified by the `spaceCount` and `spaceArray` fields of the `LayoutData` structure. Extra advance width is added to each occurrence of one of these space characters on each line as necessary to expand the line's width to the maximum span. A typical set of space characters consists of the ordinary space U+0020 and the non-breaking space U+00A0.

2.12. Tab Spacing

Tab spacing is enabled by setting the `kLayoutTabSpacing` bit in the `layoutFlags` field of the `LayoutData` structure, and the distance between consecutive tab stops is specified in the `tabSize` field. When tab

spacing is enabled, each tab character (with Unicode value U+0009) appearing in the input string causes the drawing position to be moved to the next tab stop.

The `tabRound` field of the `LayoutData` structure can be used to avoid tiny advances when the drawing position is very close to the next tab stop. This value is added to the drawing position before determining where the next tab stop is. A recommended value for the `tabRound` field is 0.125, which is half the typical width of a space character.

Note that the distance between tab stops is measured in absolute units just like the font size, but the tab rounding value is specified in em units.

2.13. Grid Positioning

Slug has a special layout mode called *grid positioning* that places glyphs at regularly spaced intervals determined solely by the current tracking value. Grid positioning is enabled by setting the `kLayoutGrid-Positioning` bit in the `layoutFlags` field of the `LayoutData` structure. When the grid positioning mode is active, the advance width for each glyph is ignored, and kerning is always disabled. Furthermore, each glyph is horizontally centered upon the current drawing position. The distance from the center of one glyph to the center of the next is given by the tracking value multiplied by the font size, the stretch factor, and the x component of the text scale specified in the `LayoutData` structure.

3

Vector Graphics

Vector graphics is the name given to 2D graphics that are defined by mathematical curves as opposed to an array of pixels. Vector graphics can be scaled to any size without loss of fidelity, and they are converted to a raster image matching the resolution of the output device when they are rendered. The glyphs defined by a font are a specific type of vector graphics, but the rendering methods used by Slug can be generalized to support a much wider array of applications.

Slug separates the components of vector graphics into two types, *fills* and *strokes*. For a shape whose outline is defined by a set of Bézier curves, a fill corresponds to the interior area enclosed by the outline, and a stroke corresponds to the outline itself. Fills and strokes can be arbitrarily combined to form detailed graphics. Slug is able to import these graphics from files in the Scalable Vector Graphics (SVG) and Open Vector Graphics Exchange (OpenVEX) formats (see Chapter 8). Slug also provides the ability to generate fills and strokes at run time using the `CreateFill()` and `CreateStroke()` functions.

3.1. Fills

A *fill* is simply a region of space bounded by a set of Bézier curves that is filled with either a solid color or a gradient. Fills can be defined by any arbitrary set of closed contours composed of a continuous sequence of Bézier curves. Fills follow the same winding rules as glyphs, so contours wound the opposite direction inside a larger contour correspond to holes in the filled region. When a fill is created at run time, its properties are specified using the `FillData` structure.

Gradients

A fill can be drawn with a solid color or with a gradient. Slug supports two types of gradients known as *linear* gradients and *radial* gradients, and these both smoothly interpolate between two colors. Examples of both types of gradients are shown in Figure 3.13. A linear gradient varies from the first color to the second color in one specific direction as distance from a given line increases. A radial gradient varies from the first color to the second color in all directions as distance from a given point increases.

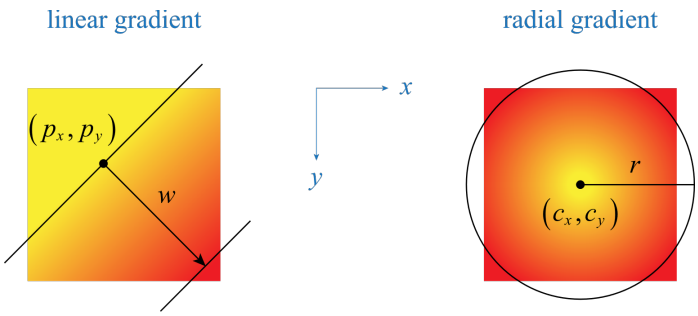

Figure 3.13. (Left) A linear gradient interpolates between two colors (yellow to red in this case) using distance from a line. (Right) A radial gradient interpolates between two colors using distance from a point.

The position, direction, and width of a linear gradient are determined by three numbers defining a scaled line in 2D space. This line coincides with the location where the first color of the gradient is rendered inside a fill, and its scale is the reciprocal of the gradient's width. For a gradient of width w starting at a line containing the point (p_x, p_y) and having the unit-length direction (n_x, n_y), the three components of the line are given by

$$\left(\frac{n_x}{w}, \frac{n_y}{w}, -\frac{n_x p_x + n_y p_y}{w}\right).$$

These values are the components of a 2D bivector, and they should be assigned to the `gradientLine` field of the `FillData` structure when using a linear gradient.

The position and radius of a radial gradient are determined by three numbers defining a circle in 2D space. The center (c_x, c_y) of the circle coincides with the location where the first color of the gradient is rendered inside a fill, and the radius r specifies the distance from the center at which the second color of the gradient is rendered. The values (c_x, c_y, r) should be assigned to the `gradientCircle` field of the `FillData` structure when using a radial gradient.

3.2. Strokes

A *stroke* is a region of space within a small distance of every point on a set of Bézier curves. Whereas fills correspond to the interior region of a path, strokes correspond to the boundary of that region. Unlike fills, the set of Bézier curves to which a stroke is applied does not have to form closed contours. When a stroke is created at run time, its properties are specified using the `StrokeData` structure.

Each stroke has a constant width that is specified in the `strokeWidth` field of the `StrokeData` structure. A stroke is centered on the Bézier curves defining its path, so half of a stroke lies on one side of the curves, and half lies on the other side.

Slug supports various aspects of stroke rendering that are standard in vector graphics, including cap styles, join styles, and dashing. These are described below.

Cap Styles

Since a stroke can be open, it makes sense to provide different ways of capping its beginning and ending points. Slug supports the four styles shown in Figure 3.14, which are named flat, square, round, and triangle. The flat style really means that there is no cap and the stroke ends precisely at the final control point in the set of Bézier curves. The other three cap styles each cause additional rendering to occur to exactly half the stroke width past the final control point in the shape of a semisquare, semicircle, or isosceles triangle. The cap style is specified in the `strokeCapType` field of the `StrokeData` structure.

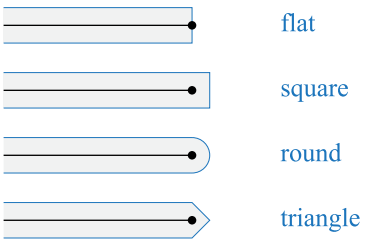

Figure 3.14. Open strokes can have flat ends, or they can have caps that are square, round, or triangular.

Join Styles

When two consecutive Bézier curves belonging to a path do not have the same tangent direction at their endpoints, a corner is formed. The two Bézier curves can be joined at this corner in one of the three different styles shown in Figure 3.15. A full miter join is rendered whenever the length of the edge where the two curves meet falls below a specified value called the *miter limit*. The miter limit is the ratio of the edge length to the width of the stroke, and it is specified in the `miterLimit` field of the `StrokeData` structure. If this limit is exceeded, then the join style is either bevel or round as determined by the `strokeJoinType` field of the `StrokeData` structure. The miter limit can be set to zero, and this forces all joins with discontinuous tangent directions to be rendered as either bevel joins or round joins.

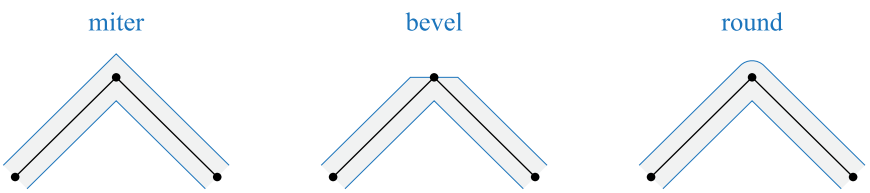

Figure 3.15. When two curves meet at a corner, the stroke is rendered with a miter join whenever the length of the edge where the curves meet falls below a specified multiple of the stroke width, called the miter limit. If the miter limit is exceeded, then either a bevel join or round join is rendered, as chosen by the application.

Dashing

Slug can render strokes with arbitrary dashing. The lengths of the dashes and the sizes of the gaps in between dashes are specified by an array containing an even number of values. Each pair of values corresponds to a dash length and the size of the gap that follows it. As a stroke progresses from its starting point, dashes and gaps are applied in order until they have all been used, and then the cycle repeats until the end of the stroke is reached. This is illustrated in Figure 3.16 for four values corresponding to two dash lengths and two gap lengths.

The number of dashes and the array of dash and gap lengths are specified in the `dashCount` and `dashArray` fields of the `StrokeData` structure. It is also possible to offset the application of dashing at the beginning of a stroke as if there had already been some curve length preceding it, and this offset is specified in the `dashOffset` field.

When the cap style for a stroke is not flat, caps are applied to both ends of each individual dash. Whenever two curves meet inside a dash, as opposed to inside a gap, the miter limit and join style determine how any corners are rendered.

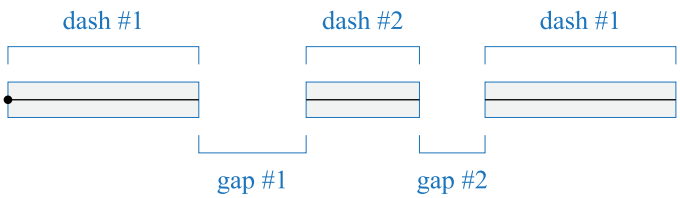

Figure 3.16. Dashes and gaps are applied in order along a stroke until they have all been used, and then the cycle repeats. In this case, there are two different dash and gap lengths.

4

Rendering

The text-handling functions of the Slug run-time library take a Unicode string (encoded as UTF-8), lay out the corresponding glyphs, and generate a vertex buffer containing the data needed to draw them. Slug's vector graphics functions can be used to draw premade icons and pictures or to generate arbitrary fills and strokes at run time. This chapter describes how the Slug library is used by an application to perform these functions with the various options available in the Slug shader.

4.1. Font and Album Resources

Rendering text or pictures with Slug requires two texture maps containing geometric data, a vertex and fragment shader pair corresponding to the rendering options, and buffers containing the vertex and triangle data for the specific items to be rendered. These resources are either retrieved from a font or album file (in the Slug format) or generated by the Slug run-time library, and they are then passed to whatever rendering API is used by the application.

Textures

The two texture maps needed by Slug are called the *curve texture* and the *band texture*, and they are stored in compressed form inside the `.slug` file for each font or album. The curve texture holds all of the Bézier curve data in either a 4×16-bit floating-point format or a 4×32-bit floating-point format, and the band texture holds spatial data structures necessary for high-performance rendering in a 2×16-bit unsigned integer format. These two textures are decompressed into application-allocated storage buffers by calling the `ExtractCurveTexture()` and the `ExtractBandTexture()` functions, as demonstrated in Listing 4.1. The required storage space for the two textures can be calculated by calling the `GetCurveTextureStorageSize()` and `GetBandTextureStorageSize()` functions. These functions apply to both font and album resources, and they each accept a pointer to the `SlugFileHeader` structure at the beginning of every `.slug` file's contents.

Listing 4.1. This code allocates storage for the curve and band textures belonging to a Slug resource and decompresses the texture data from a `.slug` file.

```
void MakeSlugTextures(const Slug::SlugFileHeader *fileHeader)
{
    uint32 curveStorageSize = Slug::GetCurveTextureStorageSize(fileHeader);
    uint32 bandStorageSize = Slug::GetBandTextureStorageSize(fileHeader);
```

```
    char *curveTexture = new char[curveStorageSize];
    char *bandTexture = new char[bandStorageSize];

    Slug::ExtractCurveTexture(fileHeader, curveTexture);
    Slug::ExtractBandTexture(fileHeader, bandTexture);

    /* Pass texture maps to rendering API here. */

    delete[] bandTexture;
    delete[] curveTexture;
}
```

Shaders

Slug renders glyphs and icons using a set of vertex shaders and fragment shaders. The source code for these shaders contains preprocessor directives that automatically handle conditional compilation for the HLSL, GLSL, Metal, and PSSL shading languages. There are also preprocessor directives that cause different variants of the fragment shader to be compiled, and these correspond to different font capabilities and rendering options. The shader variants are associated with constant index values that are determined by the `GetShaderIndices()` function, and these index values are then used to retrieve the corresponding components of the full shader code from the run-time library.

The `GetVertexShaderSourceCode()` and `GetFragmentShaderSourceCode()` functions are called to retrieve the source code for the vertex shader and fragment shader for specific shader index values. The source code is returned as an array of strings that form a complete shader when they are concatenated. This method allows many different variants to be assembled from smaller pieces, and it enables the incorporation of the shaders into an external shading system, as discussed below.

When using the Direct3D rendering API, the shader strings must be concatenated before being passed to the `D3DCompile()` function. The Slug library includes helper functions named `GetShaderString-Length()` and `MakeShaderString()` as a convenient way to perform this concatenation, but their use is not required. Compiling standalone vertex and fragment shaders under the Direct3D API is demonstrated in Listing 4.2. Note that the entry point passed to the `D3DCompile()` function must be `"main"`, and the shader target must correspond to shader model 4.0 or higher.

When using the OpenGL rendering API, the shaders strings do not need to be concatenated, but can be passed directly to the shader compiler as an array. GLSL compilers require that the first line of any shader contain a version directive, so it is necessary to begin the array of strings with either the statement

```
#version 330
```

in desktop OpenGL or the statement

```
#version 300 es
```

in OpenGL ES. (These version numbers represent the minimum requirement of the Slug library, and higher versions can also be specified.) Compiling standalone vertex and fragment shaders under the OpenGL API is demonstrated in Listing 4.3.

The vertex shader requires that a model-view-projection (MVP) matrix be supplied as four rows of four 32-bit floating-point values. (These are the 16 entries of a matrix that transforms column vectors.) It also requires that the dimensions of the viewport be supplied as two additional 32-bit floating-point values. In Direct3D, this information must be stored as 18 consecutive numbers in constant buffer 0, where the first 16 numbers are the MVP matrix in row-major order, and the last two numbers are the width and height of the viewport. In OpenGL, this information is passed to the vertex shader as two uniform inputs having the names `"slug_matrix"` and `"slug_viewport"`. Their locations must be retrieved with the `glGetUniformLocation()` function.

The curve texture and band texture must be bound to texture slots 0 and 1, respectively, for use by the fragment shader. In Direct3D, this is done by calling the `PSSetShaderResources()` function to bind two texture views beginning at slot 0. In OpenGL, this is done by calling the `glBindTexture()` function twice to bind the texture objects to units 0 and 1 with the target `GL_TEXTURE_2D`. The fragment shader does not use samplers.

Listing 4.2. This code compiles the HLSL vertex and fragment shaders needed to render with specific options given by the `renderFlags` parameter. The `vertexBlob` and `fragmentBlob` parameters point to `ID3DBlob` objects that receive the compiled shader code. The shader strings returned by the Slug library are concatenated before being passed to the `D3DCompile()` function.

```
void CompileSlugShaders(uint32 renderFlags,
                        ID3DBlob *vertexBlob,
                        ID3DBlob *fragmentBlob)
{
    uint32          vertexIndex, fragmentIndex;
    const char      *vertexCode[Slug::kMaxVertexStringCount];
    const char      *fragmentCode[Slug::kMaxFragmentStringCount];

    // Retrieve shader indices for rendering options.
    Slug::GetShaderIndices(renderFlags, &vertexIndex, &fragmentIndex);

    // Store the shader source code components in the arrays.
    int32 vStrCnt = Slug::GetVertexShaderSourceCode(vertexIndex, vertexCode);
    int32 fStrCnt = Slug::GetFragmentShaderSourceCode(fragmentIndex,
            fragmentCode);

    // Concatenate the vertex shader strings.
    int32 vStrLen = Slug::GetShaderStringLength(vStrCnt, vertexCode);
    char *vertexString = new char[vStrLen];
    Slug::MakeShaderString(vStrCnt, vertexCode, vertexString);
```

```
    // Compile the vertex shader with the full shader string.
    D3DCompile(vertexString, vStrLen, nullptr, nullptr, nullptr,
            "main", "vs_4_0", D3DCOMPILE_OPTIMIZATION_LEVEL3, 0,
            &vertexBlob, nullptr);

    // Concatenate the fragment shader strings.
    int32 fStrLen = Slug::GetShaderStringLength(fStrCnt, fragmentCode);
    char *fragmentString = new char[fStrLen];
    Slug::MakeShaderString(fStrCnt, fragmentCode, fragmentString);

    // Compile the fragment shader with the full shader string.
    D3DCompile(fragmentString, fStrLen, nullptr, nullptr, nullptr,
            "main", "ps_4_0", D3DCOMPILE_OPTIMIZATION_LEVEL3, 0,
            &fragmentBlob, nullptr);

    delete[] fragmentString;
    delete[] vertexString;
}
```

Listing 4.3. This code compiles the GLSL vertex and fragment shaders needed to render with specific options given by the `renderFlags` parameter. The `vertexShaderName` and `fragmentShaderName` parameters are assumed to be valid OpenGL shader object names that were previously established. The calls to the `glShaderSource()` function both pass an array of strings containing the components of the shaders. The first string is the version directive required by GLSL, and the remaining strings are returned by the Slug library.

```
void CompileSlugShaders(uint32 renderFlags,
                        GLuint vertexShaderName,
                        GLuint fragmentShaderName)
{
    uint32         vertexIndex, fragmentIndex;
    const char     *vertexCode[Slug::kMaxVertexStringCount + 1];
    const char     *fragmentCode[Slug::kMaxFragmentStringCount + 1];

    // Retrieve shader indices for rendering options.
    Slug::GetShaderIndices(renderFlags, &vertexIndex, &fragmentIndex);

    // Set the first string to the required version directive.
    vertexCode[0] = fragmentCode[0] = "#version 330\n";

    // Store the source code components after the version string.
    int32 vStrCnt = Slug::GetVertexShaderSourceCode(vertexIndex,
            &vertexCode[1]);
    int32 fStrCnt = Slug::GetFragmentShaderSourceCode(fragmentIndex,
            &fragmentCode[1]);
```

```
    // Compile the vertex shader with version directive and source.
    glShaderSource(vertexShaderName, vStrCnt + 1, vertexCode, nullptr);
    glCompileShader(vertexShaderName);

    // Compile the fragment shader with version directive and source.
    glShaderSource(fragmentShaderName, fStrCnt + 1, fragmentCode, nullptr);
    glCompileShader(fragmentShaderName);
}
```

External Shading Systems

Slug supports the incorporation of Slug shaders into external shading systems so that additional material properties and lighting can be applied, as shown in Figure 4.1. This is accomplished by performing all of the important work inside special functions in the vertex and fragment shaders. In the vertex shader, the `SlugUnpack()` function transforms some vertex attributes from their input format to the format consumed by the fragment shader, and the `SlugDilate()` function performs dynamic glyph dilation. In the fragment shader, the `SlugRender()` is called to calculate the final glyph color and coverage. In standalone shaders, there is a `main()` function that does nothing except call the special functions and return the result. The flags passed as the third parameter to the `GetVertexShaderSourceCode()` and `GetFragmentShaderSourceCode()` functions can be used to omit the `main()` function as well as other components so that only the essential glyph shading code is included in cases where an external shading system will supply the remaining code. This is discussed further in the documentation for the `GetVertexShaderSourceCode()` and `GetFragmentShaderSourceCode()` functions.

Figure 4.1. Slug shaders are incorporated into an external shading system where material properties and lighting are applied.

4.2. Building a Slug

Before Slug can perform layout operations on a block of text, the original Unicode characters in the text string are first translated into glyphs. The application explicitly directs Slug to take this step by calling the `CompileString()` function to generate information about the glyphs, fonts, layout features, and directional runs and store them in a `CompiledStorage` object. Once this step has been completed, other library functions can be called to determine vertex and triangle counts, to make measurements, and to perform final text layout. Each of these functions makes use of the compiled data already stored in the `CompiledStorage` object by a preceding call to the `CompileString()` function. The actual parameter passed to the API functions is a pointer to a `CompiledText` structure, which is a header at the beginning of the compiled storage data. (This allows for compiled storage buffers of varying size. See the `MakeCompactCompiledText()` function.)

Once a text string has been compiled, the vertex and triangle data needed to render the corresponding glyphs are generated by the `BuildSlug()` function, and the required storage sizes for this data are determined by calling the `CountSlug()` function. An application typically calls the `CountSlug()` function first, then allocates GPU-visible vertex buffers using the rendering API, and finally calls the `BuildSlug()` function to write the vertex and triangle data directly to those buffers.

Information about the vertex and triangle storage is passed to the `BuildSlug()` function through the `GeometryBuffer` structure, which holds a pointer to a `Vertex` array, a pointer to a `Triangle` array, and a base vertex index. These fields are each updated by the `BuildSlug()` function, and the difference between their final and initial values must be calculated to determine the actual numbers of vertices and triangles that were generated, as shown in Listing 4.4. The `CountSlug()` function returns the maximum amount of storage that could be required, but the actual amount used by the `BuildSlug()` function could be less depending on factors such as clipping planes that are not able to be considered by the `CountSlug()` function.

Listing 4.4. This code builds the vertex and triangle data for a text string. The pointers stored in the `GeometryBuffer` structure are intended to be mapped from GPU-visible memory. The final vertex and triangle counts are determined by subtracting the original values of these pointers from the updated values after the call to the `BuildSlug()` function.

```
void BuildText(const Slug::FontHeader *fontHeader,
               const Slug::LayoutData *layoutData,
               const char *text)
{
    int32                   vertexCount;
    int32                   triangleCount;
    Slug::GeometryBuffer    geometryBuffer;

    const Slug::CompiledText *compiledText = Slug::CompileString(fontHeader,
            layoutData, text);
```

```
    // Determine maximum vertex and triangle counts.
    Slug::CountSlug(compiledText, nullptr, fontHeader,
            &vertexCount, &triangleCount);

    int32 vertexBufferSize = vertexCount * sizeof(Slug::Vertex);
    int32 triangleBufferSize = triangleCount * sizeof(Slug::Triangle);

    /* Allocate buffers here with rendering API. */

    Slug::Vertex *vertexBase = /* mapped ptr to vertex buffer */;
    Slug::Triangle *triangleBase = /* mapped ptr to triangle buffer */;

    geometryBuffer.vertexData = vertexBase;
    geometryBuffer.triangleData = triangleBase;
    geometryBuffer.vertexIndex = 0;

    Slug::BuildSlug(compiledText, nullptr, fontHeader, Point2D{0.0F, 0.0F},
            &geometryBuffer);

    // Calculate actual vertex and triangle counts.
    vertexCount = geometryBuffer.vertexData - vertexBase;
    triangleCount = geometryBuffer.triangleData - triangleBase;
}
```

4.3. Multi-Line Text

Slug includes high-level functions that can be used to lay out text occupying multiple lines with a given maximum span. These functions break text into multiple lines, count the numbers of vertices and triangles that will be generated by a subset of those lines of the text, and generate the actual geometry needed to render a subset of those lines. After text is broken into lines, the process of rendering multi-line text is similar to that for a single line of text, as demonstrated in Listing 4.5.

Listing 4.5. This code breaks a text string into multiple lines and builds the vertex and triangle data needed to render it. The `maxSpan` parameter specifies the horizontal width of the box into which the text is fit. The pointers stored in the `GeometryBuffer` structure are intended to be mapped from GPU-visible memory. The final vertex and triangle counts are determined by subtracting the original values of these pointers from the updated values after the call to the `BuildMultiLineText()` function.

```
void BuildParagraphs(const Slug::FontHeader *fontHeader,
                     const Slug::LayoutData *layoutData,
                     const char *text, float maxSpan)
{
    static const uint32 softBreakArray[3] = {' ', '-', '/'};
    static const uint32 hardBreakArray[1] = {'\n'};
    static const uint32 trimArray[1] = {' '};
```

```
    int32                   vertexCount;
    int32                   triangleCount;
    Slug::GeometryBuffer    geometryBuffer;
    Slug::LineData          lineData[16];

    const Slug::CompiledText *compiledText = Slug::CompileString(fontHeader,
            layoutData, text);

    // Break the text into multiple lines up to a maximum of 16.
    int32 lineCount = Slug::BreakMultiLineText(compiledText, fontHeader,
            maxSpan, 3, softBreakArray, 1, hardBreakArray,
            1, trimArray, 16, lineData);

    // Determine maximum vertex and triangle counts.
    Slug::CountMultiLineText(compiledText, fontHeader, 0,
            lineCount, lineData, &vertexCount, &triangleCount);

    int32 vertexBufferSize = vertexCount * sizeof(Slug::Vertex);
    int32 triangleBufferSize = triangleCount * sizeof(Slug::Triangle);

    /* Allocate buffers here with rendering API. */

    Slug::Vertex *vertexBase = /* mapped ptr to vertex buffer */;
    Slug::Triangle *triangleBase = /* mapped ptr to triangle buffer */;

    geometryBuffer.vertexData = vertexBase;
    geometryBuffer.triangleData = triangleBase;
    geometryBuffer.vertexIndex = 0;

    Slug::BuildMultiLineText(compiledText, fontHeader, 0,
          lineCount, lineData, Point2D{0.0F, 0.0F}, maxSpan,
          &geometryBuffer);

    // Calculate actual vertex and triangle counts.
    vertexCount = geometryBuffer.vertexData - vertexBase;
    triangleCount = geometryBuffer.triangleData - triangleBase;
}
```

The `BreakMultiLineText()` function determines the locations where a text string should be broken to start new lines, and it provides a flexible method for specifying soft break and hard break characters. A *soft break* character is one after which a line break is allowed but not required, and these characters usually include spaces and hyphens. A *hard break* character is one after which a line break is always required, and these characters usually include newlines (U+000A). The `BreakMultiLineText()` function outputs an array of `LineData` structures that each contain the string length and physical span of a line of text.

So that nonprinting characters at the end of a line don't wrap around to the next line, even if they exceed the maximum span of the text, Slug allows the specification of a set of *trim* characters. Any trim characters occurring at the end of a line are always kept with that line, and they don't participate in measurements used for text alignment. Trim characters usually include spaces and other characters that don't generate any geometry.

The `CountMultiLineText()` function is the multi-line analog of the `CountSlug()` function. It calculates the maximum number of vertices and triangles that could be needed to render a text string that has been broken into multiple lines. It can be used to count only a subset of the lines determined by the `BreakMultiLineText()` function so that it's possible to break text into lines once and then rebuild the text that's actually rendered more than once with that information. This would be useful in cases such as text being scrolled through a view of some kind.

Finally, the `BuildMultiLineText()` function is the multi-line analog of the `BuildSlug()` function. It builds the vertex and triangle arrays for the same set of lines that were processed by the `CountMultiLineText()` function. This function uses the `textAlignment` and `textLeading` fields of the `LayoutData` structure to properly calculate the horizontal position of the text on each line and the spacing between consecutive lines. The `BuildMultiLineText()` function can also apply the paragraph attributes specified by the `paragraphSpacing`, `leftMargin`, `rightMargin`, and `firstLineIndent` fields of the `LayoutData` structure.

4.4. Custom Glyph Layout

Instead of immediately generating vertex and triangle data, Slug can generate arrays containing glyph indices, drawing positions, transforms, and colors. This information can then be modified by the application in order to perform custom text layout. Afterward, Slug can generate vertex and triangle data using the modified per-glyph data. This information is also useful in cases where the application renders glyphs using some external method. To use this functionality, the `LayoutSlug()` and `LayoutMultiLineText()` functions are used in place of the `BuildSlug()` and `BuildMultiLineText()` functions. The `CountSlug()` and `CountMultiLineText()` functions are still needed to determine the number of glyphs that would be generated for a given text string. After any modifications are made to the per-glyph data, the `AssembleSlug()` function is called to generate the vertex and triangle data based on the arrays of glyph indices, positions, transforms, and colors.

Before the `LayoutSlug()` or `LayoutMultiLineText()` function is called, memory must be allocated for separate arrays that will receive glyph indices, glyph positions, glyph transforms, and glyph colors. Glyph indices and positions are always generated, but transforms and colors can optionally be omitted from the data that is returned. The number of glyphs for which space needs to be allocated is the return value of the corresponding `CountSlug()` or `CountMultiLineText()` function. Text decorations (underline and strikethrough) and special effects (drop shadow and outline) are ignored when calculating the number of glyphs and generating their positions. However, special effects are still applied by the `AssembleSlug()` function if they are enabled.

4.5. Placeholders

There may be cases in which the application needs to insert its own custom graphics into a string of text. For example, the application may want to display an icon inside a sentence telling the user what button to press to perform some action. Slug is able to reserve space for these kinds of graphics through the use of *placeholders*. Whenever a placeholder appears in a text string, Slug leaves open the amount of blank horizontal space associated with that placeholder, and it reports its location so the application can later fill the space with its own graphics by other means.

A placeholder is identified in a text string by any Unicode value falling in a special range designated by the application as placeholder representatives. By default, placeholder identifiers begin at U+F0000, which is the first value in a large supplementary private use area defined by the Unicode standard. However, the application may choose any nonzero Unicode value as the first placeholder identifier by setting the `placeholderBase` field of the `LayoutData` structure. The number of placeholder types is given by the `placeholderCount` field, and placeholder functionality is enabled when this count is not zero. Each type of placeholder may appear an unlimited number of times in a text string.

When placeholders are enabled with `placeholderBase` set to b and `placeholderCount` set to n, any Unicode value u in the range $[b, b+n-1]$ is interpreted as the type of placeholder with index $i = u - b$. The widths of the various types of placeholders are supplied by the application through the `placeholderWidthArray` field of the `LayoutData` structure, which must contain n entries. Whenever a placeholder of a specific type index i is encountered in a text string, Slug looks up its width under entry i in the array and advances the drawing position by that amount.

The `CountSlug()` or `CountMultiLineText()` functions return the total number T of placeholders of any kind occurring in a text string. To obtain the types and positions of those placeholders when the text is laid out, the application allocates an array of `PlaceholderData` structures large enough to hold T entries and passes it the `BuildSlug()`, `BuildMultiLineText()`, `LayoutSlug()`, or `LayoutMultiLineText()` function through a `PlaceholderBuffer` structure. Each `PlaceholderData` structure contains number of glyphs preceding the placeholder, the index i of the type of placeholder that occurred in the text, and the placeholder's (x, y) position. In a manner similar to the numbers of vertices and triangles written through pointers contained in the `GeometryBuffer` structure, the actual number of placeholders written through the pointer contained in the `PlaceholderBuffer` structure is determined by subtracting the original value of the pointer from the its value after text has been laid out.

4.6. Multiple Fonts

The functions for building slugs and multiple lines of text described up to this point all work with a single font. Slug includes extended versions of each of these functions that support the use of multiple fonts in the same text string through a flexible data structure called a *font map*. A font map defines the relationship among three types of information. First, the application supplies a master list of fonts that could be used by a text string. Second, the application defines a set of 32-bit type codes that are used to select font styles inside a text string using format directives. These styles can specify typical variations such as italic and bold, or they can be used to change typefaces completely. Third, the application provides a table that maps type codes to the indices of the fonts to be used in the master font list. This table is two-dimensional because several fonts can be specified for each type code. Whenever a glyph is

not available in the first font, later fonts are searched until one containing the glyph is found. This is particularly useful when a text string contains unusual characters because fonts rarely contain glyphs for the whole of Unicode.

The font mapping mechanism is illustrated in Figure 4.2. In this example, type codes are specified for four different styles. The code 0 is used for the regular Times font. Four-character codes are used for the other three styles so that it is easy to understand the meaning of the `font()` directives embedded in the text string. (See Chapter 6 about format directives.) Associated with each style, identified by its type code, is an array of indices into the master font list. These indices specify the primary font and a set of fallback fonts for the style. A different number of fonts can be specified for each style. Here, fallback fonts are specified for emoji and miscellaneous symbols for the regular, `'ital'`, and `'bold'` styles, but no fallbacks are specified for the `'code'` style.

Each of the functions in the Slug library that operates with a single font has a counterpart with the `Ex` suffix that operates with multiple fonts. For most of these functions, the parameter taking a pointer to a `FontHeader` structure is replaced by two parameters that take a font count and a pointer to an array of `FontDesc` structures. In the cases of the `LayoutSlugEx()` and `AssembleSlugEx()` functions, as well as their multi-line equivalents, one more parameter holding a pointer to an array of per-glyph font indices is also added. The `CompileStringEx()` function takes a pointer to a `FontMap` structure so that the final font to which every glyph belongs can be determined at the time when a text string is compiled into a sequence of glyphs.

The `FontDesc` structure holds a pointer to a `FontHeader` structure associated with a particular font resource. It also holds a scale value and vertical offset that are applied to the font whenever glyphs are taken from it. This provides a way to compensate for the fact that different fonts tend to have different capital heights within the em square, and some fonts like those containing emoji may need to be shifted up or down to align well with the other fonts specified by the font map.

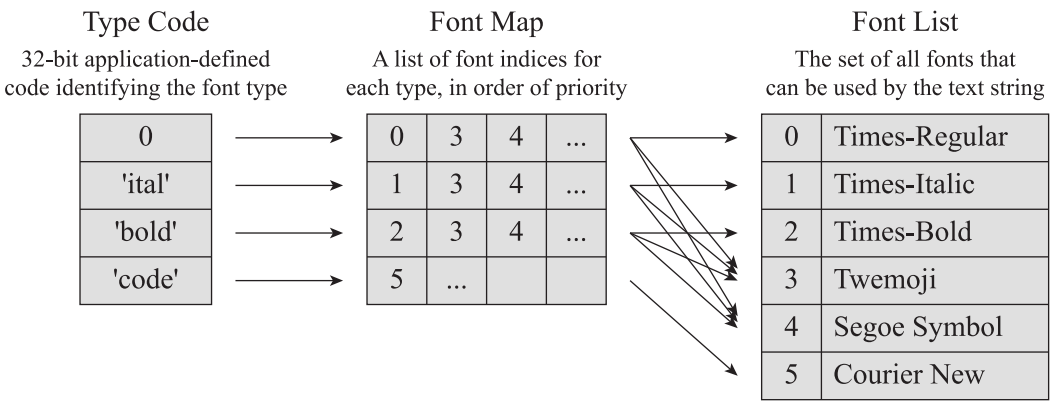

Figure 4.2. In this example of a font map structure, there are four styles identified by 32-bit type codes. For each style, there is an array of font indices that correspond to a primary font and a set of fallback fonts. These indices refer to entries in the master font list, which contains six fonts in this case.

Because each font has its own pair of texture maps, a text string using multiple fonts cannot be rendered with a single draw call. Instead, a separate draw call has to be issued for each font after the corresponding curve and band textures have been bound. No other rendering state needs to change because the same shaders and vertex buffers can be used. The `CountSlugEx()` and `CountMultiLineTextEx()` functions return separate vertex and triangle counts for each font. Enough vertex buffer space can then be allocated for the total number of vertices and triangles, and the geometry can be generated at different locations within the buffers by the `BuildSlugEx()` and `BuildMultiLineTextEx()` functions. This is demonstrated in Listing 4.6.

Listing 4.6. This code builds the vertex and triangle data for a text string using multiple fonts, up to a maximum of `kMaxFontCount`. Compare to Listing 4.4, and note how vertex counts, triangle counts, and geometry previously generated for one font are now stored in arrays corresponding to many fonts.

```
void BuildText(int32 fontCount,
               const Slug::FontDesc *fontDesc,
               const Slug::FontMap *fontMap,
               const Slug::LayoutData *layoutData,
               const char *text)
{
    int32                   vertexCount[kMaxFontCount];
    int32                   triangleCount[kMaxFontCount];
    Slug::GeometryBuffer    geometryBuffer[kMaxFontCount];
    Slug::Vertex            *vertexBase[kMaxFontCount];
    Slug::Triangle          *triangleBase[kMaxFontCount];

    const Slug::CompiledText *compiledText = Slug::CompileStringEx(fontCount,
            fontDesc, fontMap, layoutData, text);
```

```
    // Determine maximum vertex and triangle counts.
    Slug::CountSlugEx(compiledText, nullptr, fontCount, fontDesc,
            vertexCount, triangleCount);

    int32 totalVertexCount = vertexCount[0];
    int32 totalTriangleCount = triangleCount[0];
    for (int32 i = 1; i < fontCount; i++)
    {
        totalVertexCount += vertexCount[i];
        totalTriangleCount += triangleCount[i];
    }

    int32 vertexBufferSize = totalVertexCount * sizeof(Slug::Vertex);
    int32 triangleBufferSize = totalTriangleCount * sizeof(Slug::Triangle);

    /* Allocate buffers for total size here with rendering API. */

    vertexBase[0] = /* mapped ptr to vertex buffer */;
    triangleBase[0] = /* mapped ptr to triangle buffer */;

    geometryBuffer[0].vertexData = vertexBase[0];
    geometryBuffer[0].triangleData = triangleBase[0];
    geometryBuffer[0].vertexIndex = 0;

    for (int32 i = 1; i < fontCount; i++)
    {
        vertexBase[i] = vertexBase[i - 1] + vertexCount[i - 1];
        triangleBase[i] = triangleBase[i - 1] + triangleCount[i - 1];

        geometryBuffer[i].vertexData = vertexBase[i];
        geometryBuffer[i].triangleData = triangleBase[i];
        geometryBuffer[i].vertexIndex = vertexBase[i] - vertexBase[0];
    }

    Slug::BuildSlugEx(compiledText, nullptr, fontCount, fontDesc,
            Point2D{0.0F, 0.0F}, geometryBuffer);

    // Calculate actual vertex and triangle counts.
    for (int32 i = 0; i < fontCount; i++)
    {
        vertexCount[i] = geometryBuffer[i].vertexData - vertexBase[i];
        triangleCount[i] = geometryBuffer[i].triangleData - triangleBase[i];
    }
}
```

4.7. Text Colors

The color of the text is controlled by the `textColor` field of the `LayoutData` structure, which is itself another structure of the type `ColorData`. Text may be rendered as a solid color or as a gradient, as specified by the `gradientFlag` field of the `ColorData` structure. If the gradient is enabled, then the text color varies smoothly between two colors attained at two given y coordinates in em space. This information is used by the `BuildSlug()` function to calculate a color for each vertex. If a glyph effect is enabled (see Section 4.12 below), then the color of the effect is separately controlled by another `ColorData` structure named `effectColor`.

The color output by the Slug shaders is linear RGB color. This color must be converted to sRGB for proper display, and this is typically done by enabling hardware sRGB conversion for the render target. The blending function should use a source factor of source alpha and a destination factor of inverse source alpha.

4.8. Color Glyph Layers

A font may contain color layer data for some or all of its glyphs, and this information is typically present in fonts that include emoji glyphs. Multicolor glyphs are rendered as a stack of separate layers using the same shaders that render ordinary monochrome glyphs, and each layer is rendered with a solid color. The color and gradient information in the `LayoutData` structure have no effect on the colors of the individual layers.

Rendering with multiple color layers is enabled by default, but it can be disabled by setting the `kLayoutLayerDisable` flag in the `layoutFlags` field of the `LayoutData` structure. When an attempt is made to render a multicolor glyph while color layers are disabled, the corresponding monochrome glyph is rendered instead, if it is available in the font.

4.9. Optical Weight

The Slug shaders have a special option that can be used to increase the *optical weight*, or apparent heaviness, of glyphs and icons when they are rendered at small sizes. This is useful for counteracting the lightening that occurs when a font has been shrunk to a size at which few individual pixels are fully covered by a typical glyph, and it works best for dark-colored glyphs on a light background. The optical weight option boosts the coverage value by adding a short calculation to the shader. It is enabled by setting the `kRenderOpticalWeight` bit in the `renderFlags` field of the `LayoutData` structure.

4.10. Adaptive Supersampling

The Slug shaders have an advanced option that enables *adaptive supersampling*, a feature that improves the appearance of text at very small sizes. This feature is particularly useful for cases when the camera transform is changing and text is being rendered at a small size or, in a 3D scene, at a distance far from the camera. In most cases, adaptive supersampling should be turned off because it won't make a noticeable difference in appearance at normal font sizes.

Adaptive supersampling uses the screen-space derivatives at each pixel to determine a sample count between 1 and 4 independently for each of the x and y directions. Smaller glyphs, and thus higher derivatives, cause the sample count to be greater, and this produces much higher-quality per-pixel filtering. Adaptive supersampling is enabled by setting the `kRenderSupersampling` bit in the `renderFlags` field of the `LayoutData` structure.

4.11. Clipping

A line of text can be clipped by vertical planes on the left and right by the `BuildSlug()` function or `BuildMultiLineText()` function. Clipping is enabled by setting the `kLayoutClippingPlanes` bit in the `layoutFlags` field of the `LayoutData` structure. The object-space positions of the clip planes are specified by the `clipLeft` and `clipRight` fields of the `LayoutData` structure.

The `CountSlug()` function is not able to take clipping into consideration, so it always returns the maximum numbers of vertices and triangles that could be generated without clipping. This allows an application to allocate enough space one time for all cases and simply call `BuildSlug()` or `BuildMultiLineText()` by itself whenever the clipping configuration changes (e.g., for horizontally scrolling text).

4.12. Effects

Slug can render a hard drop shadow effect or a geometric outline effect for every glyph. These are enabled by specifying either `kGlyphEffectShadow` or `kGlyphEffectOutline` in the `effectType` field of the `LayoutData` structure. The shadow effect is always available, but the outline effect requires that expanded outline contours be generated when a font is converted to the Slug format. (See Chapter 7 for details.) When either of these effects is enabled, the `CountSlug()` and `CountMultiLineText()` functions account for the additional geometry that would be generated, and the `BuildSlug()` and `BuildMultiLineText()` functions generate extra vertices and triangles for the effects. The geometry corresponding to an effect directly precedes the ordinary geometry for each glyph in the output buffers.

The effect geometry can be offset from the text geometry with the `effectOffset` field of the `LayoutData` structure. This offset is ordinarily used only for the shadow effect, but it can be applied to the outline effect as well.

The fact that an effect is enabled never changes the positions that are calculated for each glyph. In the case of the outline effect, it may be desirable to add some positive tracking to prevent adjacent glyphs from colliding when thick outlines are being used.

4.13. Icons and Pictures

Using the same algorithm that allows glyphs to be rendered directly from curve data on the GPU, Slug is also able to render arbitrary vector-based icons. An icon can be monochrome, or it can be made up of multiple color layers. Like glyphs, icons are defined by a set of closed contours composed of quadratic Bézier curves. The source data can come from an album file, or it can be supplied directly to the Slug library by calling the `ImportIconData()` or `ImportMultiColorIconData()` function. Those functions fill out an `IconData` structure and generate the curve and band texture data needed to render an icon.

An icon is rendered by calling the `BuildIcon()` function to generate a list of vertices and triangles. Depending on the geometry type requested, the number of vertices can range from 3 to 8, and the number of triangles can range from 1 to 6. Once the vertex and triangle data is available, an icon is rendered with the same shaders that are used to render glyphs.

Unlike glyphs, an icon composed of multiple color layers is rendered with a special shader that computes the output of all layers at once. This type of shader is selected by the `GetShaderIndices()` function by including the `kRenderMulticolor` flag in the `renderFlags` parameter. (This flag is valid only for icons.) Rendering multiple color layers in this manner can be slower than rendering the layers separately, but it allows an icon to be properly blended with a background using partial transparency.

A picture is similar to an icon, but it is generally used to render more complicated graphics. Pictures are created by specifying the `-pict` switch when a vector graphics file is transformed into an album by the `slugicon` tool. (See Chapter 8.) Pictures are always rendered as a set of monochrome components, and the color of each component is stored in the vertex data generated by the `BuildPicture()` function. An icon that never needs to be partially transparent can always be imported as a picture. The picture equivalent of an icon generates more vertices and triangles, but it almost always renders with better performance.

4.14. Bounding Polygons

Slug normally renders each glyph as a single quad that coincides with the glyph's bounding box. When glyphs are rendered at large sizes, there are often significant areas of empty space inside the bounding box, so Slug provides an optimization that renders with a tighter polygon having 3–6 vertices and 1–4 triangles. The difference is shown in Figure 4.3. Due to various factors, this optimization may not yield a speed improvement at small font sizes, so care should be taken to measure the performance difference and use the bounding polygons only when they provide a clear advantage.

Bounding polygons are enabled by changing the `geometryType` field of the `LayoutData` structure to `kGeometryPolygons`. This option affects the numbers of vertices and triangles returned by the `CountSlug()` and `CountMultiLineText()` functions, but it does not affect which shader is used to render glyphs. The same optimization is available for icons and pictures, except in this case, the `kGeometry-Polygons` geometry type is passed directly to the `CountIcon()`, `CountPicture()`, `BuildIcon()`, and `BuildPicture()` functions.

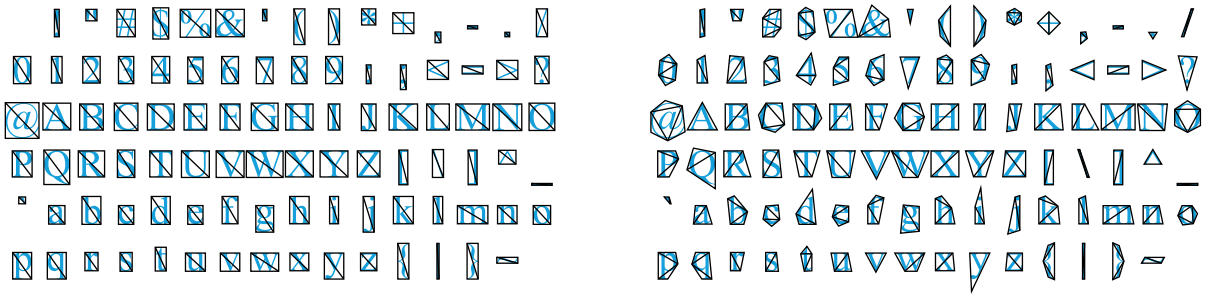

Figure 4.3. To reduce the number of pixels rendered for most glyphs, simple quads on the left are replaced with glyph-specific polygons having 3–6 vertices on the right.

For bounding polygons to be available for a font, the font must be imported by the `slugfont` tool with bounding polygon generation enabled. By default, the maximum number of vertices for a bounding polygon is only four, and this means that each glyph can be bounded by a triangle or a quadrilateral. A maximum of five or six vertices can be specified in order to allow tighter polygons, but this requires significantly more computation during the import process. Depending on the size at which glyphs are rendered, polygons with a larger number of vertices may not yield a speed improvement despite having a smaller area due to the fact that pixels belonging to 2×2 quads along interior edges are double shaded by the GPU. In general, polygons having a larger number of vertices are more effective for larger font sizes. A factor that multiplies the cost of interior edges (based on their lengths) can be adjusted to either encourage or discourage higher numbers of vertices during the import process. Setting this edge cost factor to zero causes the polygon with the smallest area to be chosen regardless of the lengths of interior edges. This would be appropriate for fonts that are always rendered at very large sizes. Higher edge cost factors make it more difficult for polygons with larger numbers of vertices to be selected, which would be appropriate for fonts that could be rendered at smaller sizes.

4.15. Optimization

Because Slug renders glyphs and icons directly from mathematical curves, such as those contained in a font, the fragment shader has to perform substantially more computation than a font rendering solution based on any kind of prerendered texture images. The previous section discusses the use of tight bounding polygons to reduce the number of pixels that need to be processed. This section discusses additional ways to minimize the amount of computation by enabling special optimizations in Slug and by making wise font selections that don't waste rendering time.

Slug has the option of generating only a single triangle per glyph or icon so that *rectangle primitives* can be used when they are exposed by the rendering API. (In Vulkan and OpenGL, rectangle primitives are exposed through extensions named `VK_NV_fill_rectangle` and `GL_NV_fill_rectangle`.) Rectangle primitives are enabled by changing the `geometryType` field of the `LayoutData` structure to `kGeometryRectangles`. This option is useful when glyph or icon bounding boxes known to always be aligned to the screen x and y axes because rectangle primitives simply fill the screen-space rectangle enclosing the three vertices of a triangle. The advantage to using rectangle primitives is that there is no longer an

internal edge where fragment shaders are executed twice per pixel in the 2×2 quads intersecting both triangles. Be aware that if rectangle primitives are rendered with user clipping planes enabled, then the GPU throws out the entire primitive whenever any one of its vertices is clipped.

When rendering pictures, there may be many components that are made up of only straight lines. Slug can take advantage of this fact to select a faster path in the shaders for these specific components as they are encountered during the process of rendering an entire picture. To enable this optimization, the `kRenderLinearCurves` bit should be set in the `renderFlags` parameter passed to the `GetShaderIndices()` function.

The greatest performance gains can often be achieved by simply selecting a font that requires less computation in the Slug shader. The most important factor is the number of Bézier curves making up each glyph because performance is directly related to the number of curves that have to be considered inside the shader. (Slug goes to great lengths to ensure that only a small subset of a glyph's full set of curves are examined at each pixel.) A font having the fewest control points possible while still possessing the desired style characteristics should be selected for best performance.

Due to the way in which the Slug rendering algorithm works, a straight horizontal or vertical line in a glyph has half the shading cost of any other Bézier curve. If the situation allows it, fonts containing such straight lines should be preferred over fonts whose glyphs have strokes that are near but not exactly horizontal or vertical.

Glyphs belonging to simple fonts like Arial sometimes have extra curves in them called *ink traps*. These are placed inside tight corners by font designers to create extra space where ink may spread out when the font is used in print, but they serve no purpose in on-screen rendering. Fonts containing ink traps render a little bit slower than similar fonts without ink traps, so they should be avoided when possible.

5

Programming Reference

This chapter provides reference documentation for the functions and data structures exposed in the Slug library's application programming interface (API), arranged in alphabetical order.

The following API functions perform the primary text rendering services provided by the Slug library.

- `MeasureSlug()`
- `CountSlug()`
- `BuildSlug()`
- `LayoutSlug()`
- `AssembleSlug()`
- `BreakMultiLineText()`
- `CountMultiLineText()`
- `BuildMultiLineText()`
- `LayoutMultiLineText()`
- `GetFontCount()`
- `GetFontHeader()`
- `GetFontKeyData()`
- `SetDefaultLayoutData()`
- `UpdateLayoutData()`
- `CalculateGlyphCount()`
- `BuildTruncatableSlug()`
- `LocateSlug()`
- `TestSlug()`

The following API functions perform the primary icon and picture rendering services provided by the Slug library.

- `CountIcon()`
- `BuildIcon()`
- `CountPicture()`
- `BuildPicture()`
- `GetAlbumHeader()`
- `ExtractAlbumTextures()`
- `ImportIconData()`
- `ImportMulticolorIconData()`

The following API functions are used to create fills and strokes at run time.

- `CountFill()`
- `CountStroke()`
- `CreateFill()`
- `CreateStroke()`
- `SetDefaultFillData()`
- `SetDefaultStrokeData()`

The following API functions are used to create textures and shaders used for rendering.

- `GetCurveTextureStorageSize()`
- `GetBandTextureStorageSize()`
- `ExtractCurveTexture()`
- `ExtractBandTexture()`
- `GetShaderIndices()`
- `GetVertexShaderSourceCode()`
- `GetFragmentShaderSourceCode()`

AlbumHeader structure

The `AlbumHeader` structure contains general information about an album.

Fields

Field	Description
`int32 iconCount`	The total number of icons in the album.
`int32 iconDataOffset`	The offset to the table of `IconData` structures.
`int32 pictureCount`	The total number of pictures in the album.
`int32 pictureDataOffset`	The offset to the table of `PictureData` structures.
`int32 meshVertexOffset`	The offset to the mesh vertex data (used for strokes).
`int32 meshTriangleOffset`	The offset to the mesh triangle data (used for strokes).

Description

The `AlbumHeader` structure contains information about the rendering characteristics of an album. Most of the fields are used internally by the Slug library functions that accept an album header. A pointer to an `AlbumHeader` structure can be obtained from the raw `.slug` file data by calling the `GetAlbumHeader()` function.

AssembleSlug() function

The `AssembleSlug()` function generates the vertices and triangles for a set of glyphs using specific positions, transforms, and colors supplied by the application.

Prototype

```
void AssembleSlug(const FontHeader *fontHeader,
                  const LayoutData *layoutData,
                  int32 glyphCount,
                  const int32 *glyphIndexBuffer,
                  const Point2D *positionBuffer,
                  const Matrix2D *matrixBuffer,
                  const Color4U *colorBuffer,
                  GeometryBuffer *geometryBuffer,
                  Box2D *textBox = nullptr);
```

Parameters

Parameter	Description
`fontHeader`	A pointer to the `FontHeader` structure retrieved with the `GetFontHeader()` function for a particular `.slug` file.
`layoutData`	A pointer to a `LayoutData` structure containing the text layout state that is applied.
`glyphCount`	The number of glyphs for which vertex and triangle data is generated.
`glyphIndexBuffer`	A pointer to an array of integers specifying the glyph indices. This parameter cannot be `nullptr`, and the array it points to must be at least as large as the `glyphCount` parameter.
`positionBuffer`	A pointer to an array of `Point2D` structures specifying the glyph positions. This parameter cannot be `nullptr`, and the array it points to must be at least as large as the `glyphCount` parameter.
`matrixBuffer`	A pointer to an array of `Matrix2D` structures specifying the glyph transforms. This parameter cannot be `nullptr`, and the array it points to must be at least as large as the `glyphCount` parameter.
`colorBuffer`	A pointer to an array of `Color4U` structures specifying the glyph colors. This parameter can be `nullptr`, in which case colors are determined by the state contained in the `LayoutData` structure. If this parameter is not `nullptr`, then the array it points to must be at least as large as the `glyphCount` parameter.

`geometryBuffer`	A pointer to the `GeometryBuffer` structure containing information about where the output vertex and triangle data is stored. This parameter can be `nullptr`, in which case no vertex and triangle data is generated.
`textBox`	A pointer to a `Box2D` structure that receives the bounding box of the entire set of glyphs. This parameter can be `nullptr`, in which case the bounding box is not returned.

Description

The `AssembleSlug()` function generates all of the vertex data and triangle data needed to render a set of glyphs using specific positions, transforms, and colors supplied by the application. This data is written in a format that is meant to be consumed directly by the GPU.

The per-glyph information provided to the `AssembleSlug()` function is typically generated by either the `LayoutSlug()` function or the `LayoutMultiLineText()` function. The positions, transforms, and colors can optionally be modified by the application prior to calling the `AssembleSlug()` function in order to create a custom text layout.

The `geometryBuffer` parameter points to a `GeometryBuffer` structure containing the addresses of the storage into which vertex and triangle data are written. These addresses are typically in memory that is visible to the GPU. Upon return from the `AssembleSlug()` function, the `GeometryBuffer` structure is updated so that the `vertexData` and `triangleData` fields point to the next element past the end of the data that was written. The `vertexIndex` field is advanced to one greater than the largest vertex index written. This updated information allows for multiple sets of glyphs having the same shaders to be built in the same vertex buffer and drawn with a single rendering command.

The actual numbers of vertices and triangles generated by the `AssembleSlug()` function should be determined by examining the pointers in the `GeometryBuffer` structure upon return and subtracting the original values of those pointers. The resulting differences can be less than the maximum values returned by the `CountSlug()` function. The code in Listing 4.4 demonstrates how the final vertex and triangle counts should be calculated.

If the `textBox` parameter is not `nullptr`, then the bounding box of the entire set of glyphs is written to the location it points to. In the case that no vertices were generated (e.g., the text string consists only of spaces), the maximum extent of the box in both the x and y directions will be less than the minimum extent, and this condition should be interpreted as an empty box.

AssembleSlugEx() function

The `AssembleSlugEx()` function generates the vertices and triangles for a set of glyphs using specific positions, transforms, and colors supplied by the application.

Prototype

```
void AssembleSlugEx(int32 fontCount,
                    const FontDesc *fontDesc,
                    const LayoutData *layoutData,
                    int32 glyphCount,
                    const uint8 *fontIndexBuffer,
                    const int32 *glyphIndexBuffer,
                    const Point2D *positionBuffer,
                    const Matrix2D *matrixBuffer,
                    const Color4U *colorBuffer,
                    GeometryBuffer *geometryBuffer,
                    Box2D *textBox = nullptr);
```

Parameters

Parameter	Description
`fontCount`	The total number of fonts that may be utilized. This must be at least 1.
`fontDesc`	A pointer to an array of `FontDesc` structures describing the fonts that may be utilized. The number of elements in this array must be equal to the value of the `fontCount` parameter.
`layoutData`	A pointer to a `LayoutData` structure containing the text layout state that is applied.
`glyphCount`	The number of glyphs for which vertex and triangle data is generated.
`fontIndexBuffer`	A pointer to an array of integers specifying the font indices. This parameter can be `nullptr`, in which case font index 0 is used for all glyphs. If this parameter is not `nullptr`, then the array it points to must be at least as large as the `glyphCount` parameter.
`glyphIndexBuffer`	A pointer to an array of integers specifying the glyph indices. This parameter cannot be `nullptr`, and the array it points to must be at least as large as the `glyphCount` parameter.

`positionBuffer`	A pointer to an array of `Point2D` structures specifying the glyph positions. This parameter cannot be `nullptr`, and the array it points to must be at least as large as the `glyphCount` parameter.
`matrixBuffer`	A pointer to an array of `Matrix2D` structures specifying the glyph transforms. This parameter cannot be `nullptr`, and the array it points to must be at least as large as the `glyphCount` parameter.
`colorBuffer`	A pointer to an array of `Color4U` structures specifying the glyph colors. This parameter can be `nullptr`, in which case colors are determined by the state contained in the `LayoutData` structure. If this parameter is not `nullptr`, then the array it points to must be at least as large as the `glyphCount` parameter.
`geometryBuffer`	A pointer to an array of `GeometryBuffer` structures containing information about where the output vertex and triangle data is stored. The number of elements in this array must be equal to the value of the `fontCount` parameter. This parameter can be `nullptr`, in which case no vertex and triangle data is generated.
`textBox`	A pointer to a `Box2D` structure that receives the bounding box of the entire set of glyphs. This parameter can be `nullptr`, in which case the bounding box is not returned.

Description

The `AssembleSlugEx()` function is an extended version of the `AssembleSlug()` function capable of handling multiple fonts through the mapping mechanism described in Section 4.6. A call to the `AssembleSlug()` function is internally forwarded to the `AssembleSlugEx()` function with the `fontCount` parameter set to 1, the `fontDesc` parameter set to the address of a single `FontDesc` structure containing the font header with default scale and offset, and the `fontIndexBuffer` parameter set to `nullptr`.

The `fontCount` and `fontDesc` parameters specify the master font list containing the full set of fonts that can be used with the text string. The `fontIndexBuffer` parameter specifies the font index within the master font list for each glyph, which would normally be generated by the `LayoutSlugEx()` function.

Aside from the first two parameters, the remaining parameters passed to the `AssembleSlugEx()` function have the same meanings as the parameters with the same names passed to the `AssembleSlug()` function. The additional `fontIndexBuffer` parameter optionally points to an array that specifies the index of the font used by each glyph.

BreakMultiLineText() function

The `BreakMultiLineText()` function determines the locations at which text should be broken into multiple lines.

Prototype

```
int32 BreakMultiLineText(const CompiledText *compiledText,
                         const FontHeader *fontHeader,
                         float maxSpan,
                         int32 softBreakCount,
                         const uint32 *softBreakArray,
                         int32 hardBreakCount,
                         const uint32 *hardBreakArray,
                         int32 trimCount,
                         const uint32 *trimArray,
                         int32 maxLineCount,
                         LineData *lineDataArray,
                         const LineData *previousLine = nullptr);
```

Parameters

Parameter	Description
`compiledText`	A pointer to a `CompiledText` object returned by a preceding call to the `CompileString()` function.
`fontHeader`	A pointer to the `FontHeader` structure retrieved with the `GetFontHeader()` function for a particular `.slug` file.
`maxSpan`	The maximum physical horizontal span of the text.
`softBreakCount`	The number of soft break characters specified by the `softBreakArray` parameter.
`softBreakArray`	A pointer to an array of soft break characters with `softBreakCount` entries. The values in this array are Unicode characters, and they must be sorted in ascending order. This parameter can `nullptr` only if the `softBreakCount` parameter is 0.
`hardBreakCount`	The number of hard break characters specified by the `hardBreakArray` parameter.
`hardBreakArray`	A pointer to an array of hard break characters with `hardBreakCount` entries. The values in this array are Unicode characters, and they must be sorted in

	ascending order. This parameter can `nullptr` only if the `hardBreakCount` parameter is 0.
`trimCount`	The number of trim characters specified by the `trimArray` parameter.
`trimArray`	A pointer to an array of trim characters with `trimCount` entries. The values in this array are Unicode characters, and they must be sorted in ascending order. This parameter can `nullptr` only if the `trimCount` parameter is 0.
`maxLineCount`	The maximum number of lines for which data is returned through the `lineDataArray` parameter.
`lineDataArray`	A pointer to an array of `LineData` structures to which information about each line of text is written. The array must be large enough to hold the maximum number of lines specified by the `maxLineCount` parameter.
`previousLine`	A pointer to a `LineData` structure containing information about the previous line in the text. This parameter can be `nullptr` to indicate that there is no previous line. The `LineData` structure specified by this parameter should be one that was generated by a preceding call to the `BreakMultiLineText()` function.

Description

The `BreakMultiLineText()` function determines the locations at which text should be broken into multiple lines such that each line fits within the specific horizontal span given by the `maxSpan` parameter. These locations are determined under the constraint that the string can be broken only where certain conditions are satisfied.

The `compiledText` parameter should be a pointer to a `CompiledText` structure returned by a preceding call to the `CompileString()` function. The pointer passed to the `fontHeader` parameter must be the same that was passed to the `fontHeader` parameter of the `CompileString()` function.

If the `softBreakCount` parameter is not zero, then the `softBreakArray` parameter must point to an array of Unicode characters with `softBreakCount` entries. These values represent the set of characters after which a line break is allowed and typically include characters such as a space (U+0020), hyphen (U+002D), and slash (U+002F). Advanced uses might also include characters such as an em dash (U+2014) or any of the sized spaces beginning at U+2000.

If the `kLayoutSoftHyphen` flag is set in the `layoutFlags` field of the `LayoutData` structure specified by the `layoutData` parameter, then a line break may occur after a soft hyphen character with Unicode value U+00AD. The soft hyphen character must also be included in the array of soft break characters specified by the `softBreakArray` parameter. Soft hyphens are typically placed inside words where it is acceptable to break lines, and a soft hyphen is rendered only if it is the final character on a line. Any soft hyphen character that does not correspond to an actual line break is not displayed and does not contribute to the physical span of a line.

If the `hardBreakCount` parameter is not zero, then the `hardBreakArray` parameter must point to an array of Unicode characters with `hardBreakCount` entries. These values represent the set of characters after which a line break is mandatory and typically include characters such as a newline (U+000A). A hard break character usually indicates that the current paragraph ends at the break and a new paragraph begins with the next character. If paragraph attributes are enabled, then this causes paragraph spacing to be applied before the new paragraph and indentation to be applied to the new paragraph's first line. To specify that a particular hard break character should not begin a new paragraph, its character code can be combined with the value `kBreakSameParagraph` with logical OR.

To accommodate text that may contain a mixture of one-character and two-character line breaks, a hard break character can be combined with the value `kBreakCombineNext` to indicate that it could be part of a single break. When this flag is present and the hard break character is immediately followed by another hard break character having a different character code, then the two breaks are combined into one. This is useful for allowing CR (carriage return) and LF (line feed) to each cause a line break when occurring in isolation, but recognizing the sequence CR-LF as a single line break. In this case, the `kBreakCombineNext` flag should be applied to the CR character in the `hardBreakArray` parameter.

The following flags can be combined with hard break characters using logical OR. Each of these inserts a flag bit into the most significant byte of the character code, but they should all be disregarded when sorting the array of hard break characters into ascending order. Only the lower 24 bits of the character code are considered when matching characters in the text.

Value	Description
`kBreakSameParagraph`	The hard break begins a new line, but it does not begin a new paragraph.
`kBreakCombineNext`	If the hard break is followed by another hard break having a different character code, then two breaks are combined into one.

The `maxLineCount` parameter specifies the maximum number of lines for which data can be written to the array of `LineData` structures specified by the `lineDataArray` parameter. The value returned by the `BreakMultiLineText()` function is the actual number of lines for which data was written. The returned number of lines can be zero in the case that the maximum physical horizontal span is insufficient to contain the first character in the string.

For each line of text, a `LineData` structure is written in the array specified by the `lineDataArray` parameter. Within this structure, the byte length of the substring of characters that fits within the maximum span, after being broken at an allowable point, is written to the `fullTextLength` field, and the physical horizontal span of that substring is written to the `fullLineSpan` field. A possibly shorter substring length and physical span that excludes trimmed characters at the end of the line are written to the `trimTextLength` and `trimLineSpan` fields. The lengths written to the `fullTextLength` and `trimTextLength` fields correspond to the sizes of substrings beginning at the address specified by the `text` parameter. If the line of text would constitute the last line in a paragraph, because the line ends with a hard break character that does not have the `kBreakSameParagraph` flag set, then the `lineFlags` field of

the `LineData` structure contains the `kLineParagraphLast` flag. Otherwise, the `lineFlags` field is set to zero.

The set of excluded characters that is trimmed at the end of each line is specified by the `trimCount` and `trimArray` parameters. If `trimCount` is not zero, then the `trimArray` parameter must point to an array of Unicode characters having the number of entries specified by `trimCount`. Values specified in this array typically include spaces and other characters that do not generate any geometry.

For any line, if a null terminator or hard break character is encountered before the maximum span is reached, then the text is always broken at that point (before a null terminator, but after a hard break character). Otherwise, the text is broken after the last soft break character that was encountered before the maximum span was reached. If no soft break character was encountered, then the text is broken after the last character that fits within the maximum span plus any contiguous run of immediately following characters that are included in the trim array.

If the `kLayoutWrapDisable` flag is set in the `layoutFlags` field of the `LayoutData` structure specified by the `layoutData` parameter, then lines of text are allowed to overflow the maximum span. In this case, lines can be broken only after hard break characters, and the value of the `maxSpan`, `softBreakCount`, and `softBreakArray` parameters have no effect (but the `softBreakArray` parameter must still be a valid pointer).

If the `previousLine` parameter is not `nullptr`, then it contains information about the last line of text that was broken by a preceding call to the `BreakMultiLineText()` function. When this information is supplied, the starting position within the string given by the `text` parameter is advanced by the value of the `fullTextLength` field of the `LineData` structure, and the `lineFlags` field of the `LineData` structure determines whether the first line of the text starting at that advanced position is the first line of a new paragraph. This mechanism can be used to accumulate data for multiple lines through multiple calls to the `BreakMultiLineText()` function. (The `previousLine` parameter is normally `nullptr` for the first such call.)

Any characters in the original text string designated as control characters by the Unicode standard may specify soft or hard break locations, but are otherwise ignored. These characters never contribute any spacing in the slug layout, even if the original font defines nonzero advance widths for them.

BreakMultiLineTextEx() function

The `BreakMultiLineTextEx()` function determines the locations at which text should be broken into multiple lines.

Prototype

```
int32 BreakMultiLineTextEx(const CompiledText *compiledText,
                           int32 fontCount,
                           const FontDesc *fontDesc,
                           float maxSpan,
                           int32 softBreakCount,
                           const uint32 *softBreakArray,
                           int32 hardBreakCount,
                           const uint32 *hardBreakArray,
                           int32 trimCount,
                           const uint32 *trimArray,
                           int32 maxLineCount,
                           LineData *lineDataArray,
                           const LineData *previousLine = nullptr);
```

Parameters

Parameter	Description
`compiledText`	A pointer to a `CompiledText` object returned by a preceding call to the `CompileStringEx()` function.
`fontCount`	The total number of fonts that may be utilized. This must be at least 1.
`fontDesc`	A pointer to an array of `FontDesc` structures describing the fonts that may be utilized. The number of elements in this array must be equal to the value of the `fontCount` parameter.
`maxSpan`	The maximum physical horizontal span of the text.
`softBreakCount`	The number of soft break characters specified by the `softBreakArray` parameter.
`softBreakArray`	A pointer to an array of soft break characters with `softBreakCount` entries. The values in this array are Unicode characters, and they must be sorted in ascending order. This parameter can `nullptr` only if the `softBreakCount` parameter is 0.
`hardBreakCount`	The number of hard break characters specified by the `hardBreakArray` parameter.

`hardBreakArray`	A pointer to an array of hard break characters with `hardBreakCount` entries. The values in this array are Unicode characters, and they must be sorted in ascending order. This parameter can `nullptr` only if the `hardBreakCount` parameter is 0.
`trimCount`	The number of trim characters specified by the `trimArray` parameter.
`trimArray`	A pointer to an array of trim characters with `trimCount` entries. The values in this array are Unicode characters, and they must be sorted in ascending order. This parameter can `nullptr` only if the `trimCount` parameter is 0.
`maxLineCount`	The maximum number of lines for which data is returned through the `lineDataArray` parameter.
`lineDataArray`	A pointer to an array of `LineData` structures to which information about each line of text is written. The array must be large enough to hold the maximum number of lines specified by the `maxLineCount` parameter.
`previousLine`	A pointer to a `LineData` structure containing information about the previous line in the text. This parameter can be `nullptr` to indicate that there is no previous line. The `LineData` structure specified by this parameter should be one that was generated by a preceding call to the `BreakMultiLineText()` function.

Description

The `BreakMultiLineTextEx()` function is an extended version of the `BreakMultiLineText()` function capable of handling multiple fonts through the mapping mechanism described in Section 4.6. A call to the `BreakMultiLineText()` function is internally forwarded to the `BreakMultiLineTextEx()` function with the `fontCount` parameter set to 1, the `fontDesc` parameter set to the address of a single `FontDesc` structure containing the font header with default scale and offset, and the `fontMap` parameter set to `nullptr`.

The `compiledText` parameter should be a pointer to a `CompiledText` structure returned by a preceding call to the `CompileStringEx()` function. The value of the `fontCount` parameter and the entries of the array specified by the `fontDesc` parameter must be exactly the same values that were passed to the `fontCount` and `fontDesc` parameters of the `CompileStringEx()` function.

After the first three parameters, the remaining parameters passed to the `BreakMultiLineTextEx()` function have the same meanings as the parameters with the same names passed to the `BreakMultiLineText()` function.

BreakSlug() function

The `BreakSlug()` function calculates the partial length of a line of text that fits within a specified physical horizontal span and optionally breaks the line at an allowable location.

Prototype

```
void BreakSlug(const CompiledText *compiledText,
               const GlyphRange *glyphRange,
               const FontHeader *fontHeader,
               float maxSpan,
               int32 softBreakCount,
               const uint32 *softBreakArray,
               int32 hardBreakCount,
               const uint32 *hardBreakArray,
               int32 trimCount,
               const uint32 *trimArray,
               LineData *lineData);
```

Parameters

Parameter	Description
`compiledText`	A pointer to a `CompiledText` object returned by a preceding call to the `CompileString()` function.
`glyphRange`	A pointer to a `GlyphRange` structure specifying the range of glyphs to process. This parameter can be `nullptr`, in which case all of the glyphs stored in the `CompiledText` object are processed.
`fontHeader`	A pointer to the `FontHeader` structure retrieved with the `GetFontHeader()` function for a particular `.slug` file.
`maxSpan`	The maximum physical horizontal span of the text.
`softBreakCount`	The number of soft break characters specified by the `softBreakArray` parameter.
`softBreakArray`	A pointer to an array of soft break characters with `softBreakCount` entries. The values in this array are Unicode characters, and they must be sorted in ascending order. This parameter can `nullptr` only if the `softBreakCount` parameter is 0.
`hardBreakCount`	The number of hard break characters specified by the `hardBreakArray` parameter.

`hardBreakArray`	A pointer to an array of hard break characters with `hardBreakCount` entries. The values in this array are Unicode characters, and they must be sorted in ascending order. This parameter can `nullptr` only if the `hardBreakCount` parameter is 0.
`trimCount`	The number of trim characters specified by the `trimArray` parameter.
`trimArray`	A pointer to an array of trim characters with `trimCount` entries. The values in this array are Unicode characters, and they must be sorted in ascending order. This parameter can `nullptr` only if the `trimCount` parameter is 0.
`lineData`	A pointer to a `LineData` structure to which information about the line of text is written.

Description

The `BreakSlug()` function determines how many characters of a text string can fit within a specific horizontal span under the constraint that the string can be broken only at specific locations. The `maxSpan` parameter specifies the maximum physical horizontal span of the text.

The `compiledText` parameter should be a pointer to a `CompiledText` structure returned by a preceding call to the `CompileString()` function. A pointer to a `GlyphRange` structure may be passed to the `glyphRange` parameter to specify that only a subset of glyphs are to be processed. The pointer passed to the `fontHeader` parameter must be the same that was passed to the `fontHeader` parameter of the `CompileString()` function.

If the `softBreakCount` parameter is not zero, then the `softBreakArray` parameter must point to an array of Unicode characters with `softBreakCount` entries. These values represent the set of characters after which a line break is allowed and typically include characters such as a space (U+0020), hyphen (U+002D), and slash (U+002F). Advanced uses might also include characters such as an em dash (U+2014) or any of the sized spaces beginning at U+2000.

If the `hardBreakCount` parameter is not zero, then the `hardBreakArray` parameter must point to an array of Unicode characters with `hardBreakCount` entries. These values represent the set of characters after which a line break is mandatory and typically include characters such as a newline (U+000A). A hard break character usually indicates that the current paragraph ends at the break and a new paragraph begins with the next character. If paragraph attributes are enabled, then this causes paragraph spacing to be applied before the new paragraph and indentation to be applied to the new paragraph's first line. To specify that a particular hard break character should not begin a new paragraph, its character code can be combined with the value `kBreakSameParagraph` with logical OR.

To accommodate text that may contain a mixture of one-character and two-character line breaks, a hard break character can be combined with the value `kBreakCombineNext` to indicate that it could be part of a single break. When this flag is present and the hard break character is immediately followed by another hard break character having a different character code, then the two breaks are combined into one. This is useful for allowing CR (carriage return) and LF (line feed) to each cause a line break when occurring

in isolation, but recognizing the sequence CR-LF as a single line break. In this case, the `kBreakCombineNext` flag should be applied to the CR character in the `hardBreakArray` parameter.

The following flags can be combined with hard break characters using logical OR. Each of these inserts a flag bit into the most significant byte of the character code, but they should all be disregarded when sorting the array of hard break characters into ascending order. Only the lower 24 bits of the character code are considered when matching characters in the text.

Value	Description
`kBreakSameParagraph`	The hard break begins a new line, but it does not begin a new paragraph.
`kBreakCombineNext`	If the hard break is followed by another hard break having a different character code, then two breaks are combined into one.

The total number of glyphs that actually fit within the maximum span after being broken at an allowable point, plus any glyphs corresponding to trimmed characters or hard break characters at the end of the line, is written to the `glyphCount` field of the `LineData` structure specified by the `lineData` parameter. This number does not include any null terminator that may occur at the end of the line. The byte length of the substring of characters corresponding to the value in the `glyphCount` field is written to the `fullTextLength` field.

The set of excluded characters that can be trimmed at the end of the line is specified by the `trimCount` and `trimArray` parameters. If `trimCount` is not zero, then the `trimArray` parameter must point to an array of Unicode characters having the number of entries specified by `trimCount`. Values specified in this array typically include spaces and other characters that do not generate any geometry.

The values written to the `firstGlyph` and `lastGlyph` fields of the `LineData` structure specified by the `lineData` parameter correspond to the range of untrimmed glyphs preceding any hard break characters or a null terminator. Because some glyphs may be excluded from this range, it can contain fewer glyphs than the number written to the `glyphCount` field. The range can also be empty if the line contains no untrimmed glyphs. The horizontal span of the text corresponding to this range of glyphs is written to the `lineSpan` field, and the byte length of this portion of the text in the original string is written to the `trimTextLength` field.

The value written to the `spaceJustify` field of the `LineData` structure is the amount of extra horizontal advance width that must be added to each space character in order to fully justify the line of text. It is set equal to the difference between the `maxSpan` parameter and the span written to the `lineSpan` field divided by the number of untrimmed space characters in the line.

If the line of text would constitute the last line in a paragraph, because the line ends with a null terminator or a hard break character that does not have the `kBreakSameParagraph` flag set, then the `lineFlags` field of the `LineData` structure contains the `kLineParagraphLast` flag. Otherwise, the `lineFlags` field is set to zero. For the last line in a paragraph, the value written to the spaceJustify field is always zero.

If a null terminator or hard break character is encountered before the maximum span is reached, then the text is always broken at that point (before a null terminator, but after a hard break character). Otherwise, the text is broken after the last soft break character that was encountered before the maximum span was reached. If no soft break character was encountered, then the text is broken after the last character that fits within the maximum span plus any contiguous run of immediately following characters that are included in the trim array.

If the `kLayoutWrapDisable` flag is set in the `layoutFlags` field of the `LayoutData` structure specified by the `layoutData` parameter, then lines of text are allowed to overflow the maximum span. In this case, lines can be broken only after hard break characters, and the value of the `maxSpan`, `softBreakCount`, and `softBreakArray` parameters have no effect (but the `softBreakArray` parameter must still be a valid pointer).

Any characters in the original text string designated as control characters by the Unicode standard may specify soft or hard break locations, but are otherwise ignored. These characters never contribute any spacing in the slug layout, even if the original font defines nonzero advance widths for them.

BreakSlugEx() function

The `BreakSlugEx()` function calculates the partial length of a line of text that fits within a specified physical horizontal span and optionally breaks the line at an allowable location.

Prototype

```
void BreakSlugEx(const CompiledText *compiledText,
                 const GlyphRange *glyphRange,
                 int32 fontCount,
                 const FontDesc *fontDesc,
                 float maxSpan,
                 int32 softBreakCount,
                 const uint32 *softBreakArray,
                 int32 hardBreakCount,
                 const uint32 *hardBreakArray,
                 int32 trimCount,
                 const uint32 *trimArray,
                 LineData *lineData);
```

Parameters

Parameter	Description
`compiledText`	A pointer to a `CompiledText` object returned by a preceding call to the `CompileStringEx()` function.
`glyphRange`	A pointer to a `GlyphRange` structure specifying the range of glyphs to process. This parameter can be `nullptr`, in which case all of the glyphs stored in the `CompiledText` object are processed.
`fontCount`	The total number of fonts that may be utilized. This must be at least 1.
`fontDesc`	A pointer to an array of `FontDesc` structures describing the fonts that may be utilized. The number of elements in this array must be equal to the value of the `fontCount` parameter.
`maxSpan`	The maximum physical horizontal span of the text.
`softBreakCount`	The number of soft break characters specified by the `softBreakArray` parameter.

softBreakArray	A pointer to an array of soft break characters with softBreakCount entries. The values in this array are Unicode characters, and they must be sorted in ascending order. This parameter can nullptr only if the softBreakCount parameter is 0.
hardBreakCount	The number of hard break characters specified by the hardBreakArray parameter.
hardBreakArray	A pointer to an array of hard break characters with hardBreakCount entries. The values in this array are Unicode characters, and they must be sorted in ascending order. This parameter can nullptr only if the hardBreakCount parameter is 0.
trimCount	The number of trim characters specified by the trimArray parameter.
trimArray	A pointer to an array of trim characters with trimCount entries. The values in this array are Unicode characters, and they must be sorted in ascending order. This parameter can nullptr only if the trimCount parameter is 0.
lineData	A pointer to a LineData structure to which information about the line of text is written.

Description

The BreakSlugEx() function is an extended version of the BreakSlug() function capable of handling multiple fonts through the mapping mechanism described in Section 4.6. A call to the BreakSlug() function is internally forwarded to the BreakSlugEx() function with the fontCount parameter set to 1, the fontDesc parameter set to the address of a single FontDesc structure containing the font header with default scale and offset, and the fontMap parameter set to nullptr.

The compiledText parameter should be a pointer to a CompiledText structure returned by a preceding call to the CompileStringEx() function. A pointer to a GlyphRange structure may be passed to the glyphRange parameter to specify that only a subset of glyphs are to be processed. The value of the fontCount parameter and the entries of the array specified by the fontDesc parameter must be exactly the same values that were passed to the fontCount and fontDesc parameters of the CompileStringEx() function.

After the first four parameters, the remaining parameters passed to the BreakSlugEx() function have the same meanings as the parameters with the same names passed to the BreakSlug() function.

BuildMultiLineText() function

The `BuildMultiLineText()` function generates the vertices and triangles for multiple lines of text.

Prototype

```
void BuildMultiLineText(const CompiledText *compiledText,
                        const FontHeader *fontHeader,
                        int32 lineIndex,
                        int32 lineCount,
                        const LineData *lineDataArray,
                        const Point2D& position,
                        float maxSpan,
                        GeometryBuffer *geometryBuffer,
                        PlaceholderBuffer *placeholderBuffer = nullptr,
                        Box2D *textBox = nullptr);
```

Parameters

Parameter	Description
`compiledText`	A pointer to a `CompiledText` object returned by a preceding call to the `CompileString()` function.
`fontHeader`	A pointer to the `FontHeader` structure retrieved with the `GetFontHeader()` function for a particular `.slug` file.
`lineIndex`	The zero-based index of the first line of text to build.
`lineCount`	The number of lines of text to build.
`lineDataArray`	A pointer to an array of `LineData` structures containing information about each line of text. The lines of text to be built correspond to elements indexed `lineIndex` through `lineIndex + lineCount - 1` in this array.
`position`	The *x* and *y* coordinates of the first glyph at the baseline of the first line of text.
`maxSpan`	The maximum physical horizontal span of the text.
`geometryBuffer`	A pointer to a `GeometryBuffer` structure containing information about where the output vertex and triangle data is stored. This parameter can be `nullptr`, in which case no vertex and triangle data is generated.

`placeholderBuffer`	A pointer to a `PlaceholderBuffer` structure containing information about where the output placeholder data is stored. This parameter can be `nullptr`, in which case no placeholder data is generated.
`textBox`	A pointer to a `Box2D` structure that receives the bounding box containing all lines of text. This parameter can be `nullptr`, in which case the bounding box is not returned.

Description

The `BuildMultiLineText()` function generates all of the vertex data and triangle data needed to render multiple lines of text. This data is written in a format that is meant to be consumed directly by the GPU.

The `compiledText` parameter should be a pointer to a `CompiledText` structure returned by a preceding call to the `CompileString()` function. The pointer passed to the `fontHeader` parameter must be the same that was passed to the `fontHeader` parameter of the `CompileString()` function.

Before the `BuildMultiLineText()` function can be called, the `BreakMultiLineText()` and `CountMultiLineText()` functions must be called for the same compiled string to determine the locations where lines break and the maximum amount of storage that the `BuildMultiLineText()` function will need to write its data. The compiled string must be exactly the same for all three functions to ensure that the correct amount of storage can be allocated and that the data generated by the `BuildMultiLineText()` function stays within the calculated limits.

The `lineIndex` parameter specifies the zero-based index of the first line of text to build, and the `lineCount` parameter specifies the number of lines to build. The `lineDataArray` parameter must point to an array of `LineData` structures containing at least `lineIndex + lineCount` elements. These would normally have been generated by a previous call to the `BreakMultiLineText()` function.

The `position` parameter specifies the *x* and *y* coordinates of the left side of the first glyph at the baseline of the first line of text. This is often (0, 0) when the transformation matrix applied externally by the application includes an object-space position.

The `maxSpan` parameter specifies the maximum physical horizontal span for all lines of text, and it's value should match the value previously passed to the `BreakMultiLineText()` function to generate the array of `LineData` structures. If the text alignment is `kAlignmentRight` or `kAlignmentCenter`, as specified by the `textAlignment` field of the `LayoutData` structure, then the `maxSpan` parameter is used to determine the proper horizontal position at which each line of text is rendered. If embedded format directives are enabled, then the alignment can be changed within a line of text, but the new alignment does not take effect until the next line is started.

The `geometryBuffer` parameter points to a `GeometryBuffer` structure containing the addresses of the storage into which vertex and triangle data are written. These addresses are typically in memory that is visible to the GPU. Upon return from the `BuildMultiLineText()` function, the `GeometryBuffer` structure is updated so that the `vertexData` and `triangleData` fields point to the next element past the end of the data that was written. The `vertexIndex` field is advanced to one greater than the largest vertex

index written. This updated information allows for multiple strings of text having the same shaders to be built in the same vertex buffer and drawn with a single rendering command.

If placeholders are being used, the `placeholderBuffer` parameter points to a `PlaceholderBuffer` structure containing the address of the storage into which placeholder information is written. Upon return from the `BuildMultiLineText()` function, the `PlaceholderBuffer` structure is updated so that the `placeholderData` field points to the next element past the data that was written in the same manner that pointers are updated in the `GeometryBuffer` structure.

The actual numbers of vertices and triangles generated by the `BuildMultiLineText()` function should be determined by examining the pointers in the `GeometryBuffer` structure upon return and subtracting the original values of those pointers. Likewise, the actual number of placeholders generated by the `BuildMultiLineText()` function should be determined by examining the pointer in the `PlaceholderBuffer` structure and subtracting the original value. The resulting differences can be less than the maximum values returned by the `CountMultiLineText()` function. The code in Listing 4.5 demonstrates how the final vertex and triangle counts should be calculated.

If the `textBox` parameter is not `nullptr`, then the bounding box containing all lines of text is written to the location it points to. In the case that no vertices were generated (e.g., the text string consists only of spaces), the maximum extent of the box in both the x and y directions will be less than the minimum extent, and this condition should be interpreted as an empty box.

Any characters in the original text string designated as control characters by the Unicode standard do not generate any output. These characters never contribute any spacing in the slug layout, even if the original font defines nonzero advance widths for them, and they never cause any vertices or triangles to be generated.

The vertex positions generated by the `BuildMultiLineText()` function have coordinates in slug space, where the x axis points to the right and the y axis points downward. (Note that the y axis in slug space points in the opposite direction of the y axis in em space.) The triangles generated by the `BuildMultiLineText()` function are wound counterclockwise in slug space.

When a new line is started, it is placed at a distance below the previous line given by the product of the font size and leading, as specified by the `fontSize` and `textLeading` fields of the `LayoutData` structure. If paragraph attributes are enabled and the new line is the first line in a new paragraph, then the leading is increased by the `paragraphSpacing` field of the `LayoutData` structure. If embedded format directives are enabled, the leading and paragraph spacing values can be changed within a line of text, but the new line spacing takes effect when the next line or paragraph is started.

BuildMultiLineTextEx() function

The `BuildMultiLineTextEx()` function generates the vertices and triangles for multiple lines of text.

Prototype

```
void BuildMultiLineTextEx(const CompiledText *compiledText,
                          int32 fontCount,
                          const FontDesc *fontDesc,
                          int32 lineIndex,
                          int32 lineCount,
                          const LineData *lineDataArray,
                          const Point2D& position,
                          float maxSpan,
                          GeometryBuffer *geometryBuffer,
                          PlaceholderBuffer *placeholderBuffer = nullptr,
                          Box2D *textBox = nullptr);
```

Parameters

Parameter	Description
`compiledText`	A pointer to a `CompiledText` object returned by a preceding call to the `CompileStringEx()` function.
`fontCount`	The total number of fonts that may be utilized. This must be at least 1.
`fontDesc`	A pointer to an array of `FontDesc` structures describing the fonts that may be utilized. The number of elements in this array must be equal to the value of the `fontCount` parameter.
`lineIndex`	The zero-based index of the first line of text to build.
`lineCount`	The number of lines of text to build.
`lineDataArray`	A pointer to an array of `LineData` structures containing information about each line of text. The lines of text to be built correspond to elements indexed `lineIndex` through `lineIndex + lineCount - 1` in this array.
`position`	The *x* and *y* coordinates of the first glyph at the baseline of the first line of text.
`maxSpan`	The maximum physical horizontal span of the text.

`geometryBuffer`	A pointer to an array of `GeometryBuffer` structures containing information about where the output vertex and triangle data is stored for each font. The number of elements in this array must be equal to the value of the `fontCount` parameter. This parameter can be `nullptr`, in which case no vertex and triangle data is generated.
`placeholderBuffer`	A pointer to a `PlaceholderBuffer` structure containing information about where the output placeholder data is stored. This parameter can be `nullptr`, in which case no placeholder data is generated.
`textBox`	A pointer to a `Box2D` structure that receives the bounding box containing all lines of text. This parameter can be `nullptr`, in which case the bounding box is not returned.

Description

The `BuildMultiLineTextEx()` function is an extended version of the `BuildMultiLineText()` function capable of handling multiple fonts through the mapping mechanism described in Section 4.6. A call to the `BuildMultiLineText()` function is internally forwarded to the `BuildMultiLineTextEx()` function with the `fontCount` parameter set to 1, the `fontDesc` parameter set to the address of a single `FontDesc` structure containing the font header with default scale and offset, and the `fontMap` parameter set to `nullptr`.

The `compiledText` parameter should be a pointer to a `CompiledText` structure returned by a preceding call to the `CompileStringEx()` function. The value of the `fontCount` parameter and the entries of the array specified by the `fontDesc` parameter must be exactly the same values that were passed to the `fontCount` and `fontDesc` parameters of the `CompileStringEx()` function.

After the first three parameters, the remaining parameters passed to the `BuildMultiLineTextEx()` function have the same meanings as the parameters with the same names passed to the `BuildMultiLineText()` function with one exception. The `geometryBuffer` parameter must now point to an array of `GeometryBuffer` structures having one entry per font. Each `GeometryBuffer` structure specifies the location where vertex and triangle data is written for the corresponding font index. Keep in mind that it is possible for no geometry to be generated for some fonts if they are not used in the portion of the string processed by the `BuildMultiLineTextEx()` function.

BuildIcon() function

The `BuildIcon()` function generates the vertices and triangles for a single icon.

Prototype

```
void BuildIcon(const IconData *iconData,
               const Point2D& iconPosition,
               const Matrix2D& iconMatrix,
               const ColorData *colorData,
               uint32 renderFlags,
               GeometryType geometryType,
               GeometryBuffer *geometryBuffer);
```

Parameters

Parameter	Description
`iconData`	A pointer to the `IconData` structure corresponding to the icon.
`iconPosition`	The *x* and *y* coordinates corresponding to the origin in the icon's local coordinate system.
`iconMatrix`	The 2×2 transformation matrix to apply to the icon's local coordinate system.
`colorData`	A pointer to a `ColorData` structure containing the solid color or gradient applied to the icon.
`renderFlags`	Flags specifying various render options. These are the same options described for the `renderFlags` parameter of the `GetShaderIndices()` function.
`geometryType`	The type of geometry used to render the icon. See the description for information about the various types.
`geometryBuffer`	A pointer to the `GeometryBuffer` structure containing information about where the output vertex and triangle data is stored.

Description

The `BuildIcon()` function generates the vertex data and triangle data needed to render a single icon. This data is written in a format that is meant to be consumed directly by the GPU.

The `iconData` parameter must point to an `IconData` structure returned by the `GetIconData()` function or generated by either the `ImportIconData()` or `ImportMulticolorIconData()` function specifically for the icon being rendered.

The `iconPosition` and `iconMatrix` parameters specify a translation and 2×2 matrix that are applied to the icon's vertex coordinates. The position is added after vertices have been transformed by the matrix. The matrix must be invertible, but it is not required to be orthogonal. The local coordinate system for an icon is the same as that for a glyph. The *x* axis points to the right, and the *y* axis points up.

The following values are the geometry types that can be specified for the `geometryType` parameter.

Value	Description
`kGeometryQuads`	The icon is rendered with a single quad composed of 4 vertices and 2 triangles.
`kGeometryPolygons`	The icon is rendered with a tight bounding polygon having between 3 and 8 vertices and between 1 and 6 triangles.
`kGeometryRectangles`	The icon is rendered with 3 vertices making up exactly one triangle with the expectation that the window-aligned bounding rectangle will be filled. In this case, no data is written to the `triangleData` array specified in the `GeometryBuffer` structure, so it can be `nullptr`. This type can be used only when rectangle primitives are available, such as provided by the `VK_NV_fill_rectangle` and `GL_NV_fill_rectangle` extensions.

The `geometryBuffer` parameter points to a `GeometryBuffer` structure containing the addresses of the storage into which vertex and triangle data are written. These addresses are typically in memory that is visible to the GPU, and they must be large enough to hold the maximum numbers of vertices and triangles that could be generated for the specified value of the `geometryType` parameter. Upon return from the `BuildIcon()` function, the `GeometryBuffer` structure is updated so that the `vertexData` and `triangleData` fields point to the next element past the end of the data that was written. The `vertexIndex` field is advanced to one greater than the largest vertex index written. This updated information allows for multiple icons having the same shaders to be built in the same vertex buffer and drawn with a single rendering command.

If the `geometryType` parameter is `kGeometryQuads`, then the `triangleData` field of the `GeometryBuffer` structure may be set to `nullptr`. In this case, no triangle data is generated, and four vertices are generated in an order that allows the icon to be rendered as a triangle strip.

BuildPicture() function

The `BuildPicture()` function generates the vertices and triangles for an entire picture.

Prototype

```
void BuildPicture(const AlbumHeader *albumHeader,
                  int32 pictureIndex,
                  const Point2D& picturePosition,
                  const Vector2D& pictureScale,
                  GeometryType geometryType,
                  GeometryBuffer *geometryBuffer);
```

Parameters

Parameter	Description
`albumHeader`	A pointer to the `AlbumHeader` structure retrieved with the `GetAlbumHeader()` function for a particular `.slug` file.
`pictureIndex`	The index of the picture within the album. This index must be between 0 and one less than the count given by the `pictureCount` field of the `AlbumHeader` structure, inclusive.
`picturePosition`	The *x* and *y* coordinates corresponding to the origin in the picture's local coordinate system.
`pictureScale`	The *x* and *y* scale to apply to the picture's vertex coordinates.
`geometryType`	The type of geometry used to render the picture. See the description for information about the various types.
`geometryBuffer`	A pointer to the `GeometryBuffer` structure containing information about where the output vertex and triangle data is stored.

Description

The `BuildPicture()` function generates the vertex data and triangle data needed to render an entire picture. This data is written in a format that is meant to be consumed directly by the GPU.

The `picturePosition` and `pictureScale` parameters specify a translation and scale that are applied to the picture's vertex coordinates. The local coordinate system for a picture is the same as that for a glyph or icon. The *x* axis points to the right, and the *y* axis points up.

The following values are the geometry types that can be specified for the `geometryType` parameter.

Value	Description
`kGeometryQuads`	The components of the picture are each rendered with a single quad composed of 4 vertices and 2 triangles.
`kGeometryPolygons`	The components of the picture are each rendered with a tight bounding polygon having between 3 and 8 vertices and between 1 and 6 triangles.
`kGeometryRectangles`	The components of the picture are each rendered with 3 vertices making up exactly one triangle with the expectation that the window-aligned bounding rectangle will be filled. This type can be used only when rectangle primitives are available, such as provided by the `VK_NV_fill_rectangle` and `GL_NV_fill_rectangle` extensions.

The `geometryBuffer` parameter points to a `GeometryBuffer` structure containing the addresses of the storage into which vertex and triangle data are written. These addresses are typically in memory that is visible to the GPU, and they must be large enough to hold the maximum numbers of vertices and triangles that could be generated for the specified value of the `geometryType` parameter. Upon return from the `BuildPicture()` function, the `GeometryBuffer` structure is updated so that the `vertexData` and `triangleData` fields point to the next element past the end of the data that was written. The `vertexIndex` field is advanced to one greater than the largest vertex index written. This updated information allows for multiple pictures having the same shaders to be built in the same vertex buffer and drawn with a single rendering command.

BuildSlug() function

The `BuildSlug()` function generates the vertices and triangles for a single line of text, or "slug".

Prototype

```
void BuildSlug(const CompiledText *compiledText,
               const GlyphRange *glyphRange,
               const FontHeader *fontHeader,
               const Point2D& position,
               GeometryBuffer *geometryBuffer,
               PlaceholderBuffer *placeholderBuffer = nullptr,
               Box2D *textBox = nullptr,
               Point2D *exitPosition = nullptr);
```

Parameters

Parameter	Description
`compiledText`	A pointer to a `CompiledText` object returned by a preceding call to the `CompileString()` function.
`glyphRange`	A pointer to a `GlyphRange` structure specifying the range of glyphs to process. This parameter can be `nullptr`, in which case all of the glyphs stored in the `CompiledText` object are processed.
`fontHeader`	A pointer to the `FontHeader` structure retrieved with the `GetFontHeader()` function for a particular `.slug` file.
`position`	The *x* and *y* coordinates of the first glyph at the baseline.
`geometryBuffer`	A pointer to the `GeometryBuffer` structure containing information about where the output vertex and triangle data is stored. This parameter can be `nullptr`, in which case no vertex and triangle data is generated.
`placeholderBuffer`	A pointer to a `PlaceholderBuffer` structure containing information about where the output placeholder data is stored. This parameter can be `nullptr`, in which case no placeholder data is generated.
`textBox`	A pointer to a `Box2D` structure that receives the bounding box of the entire line of text. This parameter can be `nullptr`, in which case the bounding box is not returned.

`exitPosition`	A pointer to a `Point2D` structure that receives the *x* and *y* coordinates of the new drawing position after it has been advanced past the final glyph. This parameter can be `nullptr`, in which case the updated position is not returned.

Description

The `BuildSlug()` function generates all of the vertex data and triangle data needed to render a single line of text, or "slug". This data is written in a format that is meant to be consumed directly by the GPU.

The `compiledText` parameter should be a pointer to a `CompiledText` structure returned by a preceding call to the `CompileString()` function. A pointer to a `GlyphRange` structure may be passed to the `glyphRange` parameter to specify that only a subset of glyphs are to be processed. The pointer passed to the `fontHeader` parameter must be the same that was passed to the `fontHeader` parameter of the `CompileString()` function.

The `glyphRange` parameter optionally points to a `GlyphRange` structure specifying the numbers of the first and last glyphs to be built.

Before the `BuildSlug()` function can be called, the `CountSlug()` function must be called for the same compiled string to determine the maximum amount of storage that the `BuildSlug()` function will need to write its data. The compiled string must be exactly the same for both functions to ensure that the correct amount of storage can be allocated and that the data generated by the `BuildSlug()` function stays within the calculated limits.

The `position` parameter specifies the *x* and *y* coordinates of the left side of the first glyph at the baseline. This is often (0, 0) when the transformation matrix applied externally by the application includes an object-space position.

The `geometryBuffer` parameter points to a `GeometryBuffer` structure containing the addresses of the storage into which vertex and triangle data are written. These addresses are typically in memory that is visible to the GPU. Upon return from the `BuildSlug()` function, the `GeometryBuffer` structure is updated so that the `vertexData` and `triangleData` fields point to the next element past the end of the data that was written. The `vertexIndex` field is advanced to one greater than the largest vertex index written. This updated information allows for multiple lines of text having the same shaders to be built in the same vertex buffer and drawn with a single rendering command.

If placeholders are being used, the `placeholderBuffer` parameter points to a `PlaceholderBuffer` structure containing the address of the storage into which placeholder information is written. Upon return from the `BuildSlug()` function, the `PlaceholderBuffer` structure is updated so that the `placeholderData` field points to the next element past the data that was written in the same manner that pointers are updated in the `GeometryBuffer` structure.

The actual numbers of vertices and triangles generated by the `BuildSlug()` function should be determined by examining the pointers in the `GeometryBuffer` structure upon return and subtracting the original values of those pointers. Likewise, the actual number of placeholders generated by the

`BuildSlug()` function should be determined by examining the pointer in the `PlaceholderBuffer` structure and subtracting the original value. The resulting differences can be less than the maximum values returned by the `CountSlug()` function. The code in Listing 4.4 demonstrates how the final vertex and triangle counts should be calculated.

If the `textBox` parameter is not `nullptr`, then the bounding box of the entire slug is written to the location it points to. In the case that no vertices were generated (e.g., the text string consists only of spaces), the maximum extent of the box in both the x and y directions will be less than the minimum extent, and this condition should be interpreted as an empty box.

If the `exitPosition` parameter is not `nullptr`, then the final drawing position is written to it. The final drawing position corresponds to the position after the advance width for the final glyph has been applied along with any tracking that may be in effect.

Any characters in the original text string designated as control characters by the Unicode standard do not generate any output. These characters never contribute any spacing in the slug layout, even if the original font defines nonzero advance widths for them, and they never cause any vertices or triangles to be generated.

The vertex positions generated by the `BuildSlug()` function have coordinates in slug space, where the x axis points to the right and the y axis points downward. (Note that the y axis in slug space points in the opposite direction of the y axis in em space.) The triangles generated by the `BuildSlug()` function are wound counterclockwise in slug space.

BuildSlugEx() function

The `BuildSlugEx()` function generates the vertices and triangles for a single line of text, or "slug".

Prototype

```
void BuildSlugEx(const CompiledText *compiledText,
                 const GlyphRange *glyphRange,
                 int32 fontCount,
                 const FontDesc *fontDesc,
                 const Point2D& position,
                 GeometryBuffer *geometryBuffer,
                 PlaceholderBuffer *placeholderBuffer = nullptr,
                 Box2D *textBox = nullptr,
                 Point2D *exitPosition = nullptr);
```

Parameters

Parameter	Description
`compiledText`	A pointer to a `CompiledText` object returned by a preceding call to the `CompileStringEx()` function.
`glyphRange`	A pointer to a `GlyphRange` structure specifying the range of glyphs to process. This parameter can be `nullptr`, in which case all of the glyphs stored in the `CompiledText` object are processed.
`fontCount`	The total number of fonts that may be utilized. This must be at least 1.
`fontDesc`	A pointer to an array of `FontDesc` structures describing the fonts that may be utilized. The number of elements in this array must be equal to the value of the `fontCount` parameter.
`position`	The *x* and *y* coordinates of the first glyph at the baseline.
`geometryBuffer`	A pointer to an array of `GeometryBuffer` structures containing information about where the output vertex and triangle data is stored for each font. The number of elements in this array must be equal to the value of the `fontCount` parameter. This parameter can be `nullptr`, in which case no vertex and triangle data is generated.
`placeholderBuffer`	A pointer to a `PlaceholderBuffer` structure containing information about where the output placeholder data is stored. This parameter can be `nullptr`, in which case no placeholder data is generated.

`textBox`	A pointer to a `Box2D` structure that receives the bounding box of the entire line of text. This parameter can be `nullptr`, in which case the bounding box is not returned.
`exitPosition`	A pointer to a `Point2D` structure that receives the *x* and *y* coordinates of the new drawing position after it has been advanced past the final glyph. This parameter can be `nullptr`, in which case the updated position is not returned.

Description

The `BuildSlugEx()` function is an extended version of the `BuildSlug()` function capable of handling multiple fonts through the mapping mechanism described in Section 4.6. A call to the `BuildSlug()` function is internally forwarded to the `BuildSlugEx()` function with the `fontCount` parameter set to 1, the `fontDesc` parameter set to the address of a single `FontDesc` structure containing the font header with default scale and offset, and the `fontMap` parameter set to `nullptr`.

The `compiledText` parameter should be a pointer to a `CompiledText` structure returned by a preceding call to the `CompileStringEx()` function. A pointer to a `GlyphRange` structure may be passed to the `glyphRange` parameter to specify that only a subset of glyphs are to be processed. The value of the `fontCount` parameter and the entries of the array specified by the `fontDesc` parameter must be exactly the same values that were passed to the `fontCount` and `fontDesc` parameters of the `CompileStringEx()` function.

After the first four parameters, the remaining parameters passed to the `BuildSlugEx()` function have the same meanings as the parameters with the same names passed to the `BuildSlug()` function with one exception. The `geometryBuffer` parameter must now point to an array of `GeometryBuffer` structures having one entry per font. Each `GeometryBuffer` structure specifies the location where vertex and triangle data is written for the corresponding font index. Keep in mind that it is possible for no geometry to be generated for some fonts if they are not used in the portion of the string processed by the `BuildSlugEx()` function.

BuildTruncatableSlug() function

The `BuildTruncatableSlug()` function builds a line of text, or "slug", that must fit into a maximum physical span or be truncated.

Prototype

```
bool BuildTruncatableSlug(const FontHeader *fontHeader,
                          const LayoutData *layoutData,
                          const char *text,
                          int32 maxLength,
                          const char *suffix,
                          uint32 buildFlags,
                          const Point2D& position,
                          float maxSpan,
                          int32 trimCount,
                          const uint32 *trimArray,
                          GeometryBuffer *geometryBuffer,
                          PlaceholderBuffer *placeholderBuffer = nullptr,
                          Box2D *textBox = nullptr,
                          Point2D *beginPosition = nullptr,
                          Point2D *endPosition = nullptr,
                          CompiledText *compiledText = nullptr);
```

Parameters

Parameter	Description
`fontHeader`	A pointer to the `FontHeader` structure retrieved with the `GetFontHeader()` function for a particular `.slug` file.
`layoutData`	A pointer to a `LayoutData` structure containing the initial text layout state that is applied.
`text`	A pointer to a text string. The characters must be encoded as UTF-8. If the `maxLength` parameter is –1, then this string must be null terminated.
`maxLength`	The maximum number of bytes to be processed in the string. If this is set to –1, then the string must be null terminated, and the entire string is processed.
`suffix`	A pointer to a text string that is appended to the primary text string in the case that it has to be truncated. The characters must be encoded as UTF-8, and this string must be null terminated.

`buildFlags`	Various flags that specify building options. See the description for information about the individual bits.
`position`	The *x* and *y* coordinates of the first glyph at the baseline.
`maxSpan`	The maximum physical horizontal span of the text.
`trimCount`	The number of trim characters specified by the `trimArray` parameter.
`trimArray`	A pointer to an array of trim characters with `trimCount` entries. The values in this array are Unicode characters, and they must be sorted in ascending order. This parameter can `nullptr` only if the `trimCount` parameter is 0.
`geometryBuffer`	A pointer to the `GeometryBuffer` structure containing information about where the output vertex and triangle data is stored. This parameter can be `nullptr`, in which case no vertex and triangle data is generated.
`placeholderBuffer`	A pointer to a `PlaceholderBuffer` structure containing information about where the output placeholder data is stored. This parameter can be `nullptr`, in which case no placeholder data is generated.
`textBox`	A pointer to a `Box2D` structure that receives the bounding box of the entire line of text. This parameter can be `nullptr`, in which case the bounding box is not returned.
`beginPosition`	A pointer to a `Point2D` structure that receives the *x* and *y* coordinates of the initial drawing position after it has been adjusted for alignment. This parameter can be `nullptr`, in which case the beginning position is not returned.
`endPosition`	A pointer to a `Point2D` structure that receives the *x* and *y* coordinates of the new drawing position after it has been advanced past the final glyph. This parameter can be `nullptr`, in which case the ending position is not returned.
`compiledText`	A pointer to a `CompiledText` structure that will be used for temporary storage. If this is `nullptr`, then internal storage shared among all callers is used. See the `CompiledText` structure for information about reentrancy.

Description

The `BuildTruncatableSlug()` function is a high-level function that builds a line of text, or "slug", that is required to fit within a maximum physical span. If the text string does not fit within the span, then it is truncated after the largest number of characters possible while still leaving enough space for a specified suffix string to be appended to it.

Before the `BuildTruncatableSlug()` function is called, the `CountSlug()` function must be called twice to determine the maximum possible number of vertices and triangles that could be built. The first call

must specify the same values for the `text` and `maxLength` parameters that are passed to the `BuildTruncatableSlug()` function. The second call to the `CountSlug()` function must specify the suffix string as its `text` parameter and must specify a `maxLength` parameter of –1. The values returned from these two calls should be added and used to allocate space for the vertices and triangles generated by the `BuildTruncatableSlug()` function.

The `fontHeader`, `layoutData`, `text`, `maxLength`, `position`, `geometryBuffer`, and `exitPosition` parameters have the same meanings as they do for the `BuildSlug()` function.

The `maxSpan` parameter specifies the maximum physical horizontal span in which the text string must fit. If the entire string specified by the `text` and `maxLength` parameters fits inside this span, then the entire string is built, the `suffix` parameter is ignored, and the return value is `true`. Otherwise, the physical span needed to build the entire suffix string is subtracted from the value of the `maxSpan` parameter, and only the number of characters from the text string fitting into this smaller span are built. The entire suffix string is then built and appended to the output generated for the truncated text string. In the case, the return value is `false`.

When the suffix string is built, the initial layout state is reset to the values specified by the `layoutData` parameter. Any changes to the layout state made by format directives in the primary text string do not affect the suffix string.

The following values can be combined (through logical OR) for the `buildFlags` parameter.

Value	Description
`kBuildTruncatableAlignment`	Alignment is applied to the text within the maximum span as specified by the `textAlignment` field of the `LayoutData` structure.
`kBuildTruncatableForceSuffix`	The suffix text is always appended to the main text even if the main text does not need to be truncated.

BuildTruncatableSlugEx() function

The `BuildTruncatableSlugEx()` function builds a line of text, or "slug", that must fit into a maximum physical span or be truncated.

Prototype

```
bool BuildTruncatableSlugEx(int32 fontCount,
                            const FontDesc *fontDesc,
                            const FontMap *fontMap,
                            const LayoutData *layoutData,
                            const char *text,
                            int32 maxLength,
                            const char *suffix,
                            uint32 buildFlags,
                            const Point2D& position,
                            float maxSpan,
                            int32 trimCount,
                            const uint32 *trimArray,
                            GeometryBuffer *geometryBuffer,
                            PlaceholderBuffer *placeholderBuffer = nullptr,
                            Box2D *textBox = nullptr,
                            Point2D *beginPosition = nullptr,
                            Point2D *endPosition = nullptr,
                            CompiledText *compiledText = nullptr);
```

Parameters

Parameter	Description
`fontCount`	The total number of fonts that may be utilized. This must be at least 1.
`fontDesc`	A pointer to an array of `FontDesc` structures describing the fonts that may be utilized. The number of elements in this array must be equal to the value of the `fontCount` parameter.
`fontMap`	A pointer to a `FontMap` structure defining the relationships among font types and the source fonts specified in the font header array. This can be `nullptr`, in which case only the font with index 0 is used.
`layoutData`	A pointer to a `LayoutData` structure containing the initial text layout state that is applied.
`text`	A pointer to a text string. The characters must be encoded as UTF-8. If the `maxLength` parameter is –1, then this string must be null terminated.

`maxLength`	The maximum number of bytes to be processed in the string. If this is set to −1, then the string must be null terminated, and the entire string is processed.
`suffix`	A pointer to a text string that is appended to the primary text string in the case that it has to be truncated. The characters must be encoded as UTF-8, and this string must be null terminated.
`buildFlags`	Various flags that specify building options. See the description for information about the individual bits.
`position`	The *x* and *y* coordinates of the first glyph at the baseline.
`maxSpan`	The maximum physical horizontal span of the text.
`trimCount`	The number of trim characters specified by the `trimArray` parameter.
`trimArray`	A pointer to an array of trim characters with `trimCount` entries. The values in this array are Unicode characters, and they must be sorted in ascending order. This parameter can `nullptr` only if the `trimCount` parameter is 0.
`geometryBuffer`	A pointer to the `GeometryBuffer` structure containing information about where the output vertex and triangle data is stored. This parameter can be `nullptr`, in which case no vertex and triangle data is generated.
`placeholderBuffer`	A pointer to a `PlaceholderBuffer` structure containing information about where the output placeholder data is stored. This parameter can be `nullptr`, in which case no placeholder data is generated.
`textBox`	A pointer to a `Box2D` structure that receives the bounding box of the entire line of text. This parameter can be `nullptr`, in which case the bounding box is not returned.
`beginPosition`	A pointer to a `Point2D` structure that receives the *x* and *y* coordinates of the initial drawing position after it has been adjusted for alignment. This parameter can be `nullptr`, in which case the beginning position is not returned.
`endPosition`	A pointer to a `Point2D` structure that receives the *x* and *y* coordinates of the new drawing position after it has been advanced past the final glyph. This parameter can be `nullptr`, in which case the ending position is not returned.
`compiledText`	A pointer to a `CompiledText` structure that will be used for temporary storage. If this is `nullptr`, then internal storage shared among all callers is used. See the `CompiledText` structure for information about reentrancy.

Description

The `BuildTruncatableSlugEx()` function is an extended version of the `BuildTruncatableSlug()` function capable of handling multiple fonts through the mapping mechanism described in Section 4.6. A call to the `BuildTruncatableSlug()` function is internally forwarded to the `BuildTruncatable-SlugEx()` function with the `fontCount` parameter set to 1, the `fontDesc` parameter set to the address of a single `FontDesc` structure containing the font header with default scale and offset, and the `fontMap` parameter set to `nullptr`.

After the first three parameters, the remaining parameters passed to the `BuildTruncatableSlugEx()` function have the same meanings as the parameters with the same names passed to the `Build-TruncatableSlug()` function with one exception. The `geometryBuffer` parameter must now point to an array of `GeometryBuffer` structures having one entry per font. Each `GeometryBuffer` structure specifies the location where vertex and triangle data is written for the corresponding font index. Keep in mind that it is possible for no geometry to be generated for some fonts if they are not used in the portion of the string processed by the `BuildTruncatableSlugEx()` function.

CalculateGlyphCount() function

The `CalculateGlyphCount()` function calculates the partial lengths of a line of compiled text that fit within specified physical horizontal spans.

Prototype

```
void CalculateGlyphCount(const CompiledText *compiledText,
                         const GlyphRange *glyphRange,
                         const FontHeader *fontHeader,
                         int32 spanCount,
                         const float *spanArray,
                         int32 trimCount,
                         const uint32 *trimArray,
                         int32 *glyphCountArray,
                         float *glyphSpanArray = nullptr);
```

Parameters

Parameter	Description
`compiledText`	A pointer to a `CompiledText` object returned by a preceding call to the `CompileString()` function.
`glyphRange`	A pointer to a `GlyphRange` structure specifying the range of glyphs to process. This parameter can be `nullptr`, in which case all of the glyphs stored in the `CompiledText` object are processed.
`fontHeader`	A pointer to the `FontHeader` structure retrieved with the `GetFontHeader()` function for a particular `.slug` file.
`spanCount`	The number of spans stored in the array specified by the `spanArray` parameter. This must be at least one.
`spanArray`	A pointer to an array of horizontal spans containing `spanCount` entries. The values in this array must be sorted in increasing order.
`trimCount`	The number of trim characters specified by the `trimArray` parameter.
`trimArray`	A pointer to an array of trim characters with `trimCount` entries. The values in this array are Unicode characters, and they must be sorted in ascending order. This parameter can `nullptr` only if the `trimCount` parameter is 0.
`glyphCountArray`	A pointer to an array having `spanCount` entries to which the glyph counts are written. This parameter cannot be `nullptr`.

`glyphSpanArray`	A pointer to an array having `spanCount` entries to which the actual spans are written. This parameter can be `nullptr`, in which case the spans not returned.

Description

The `CalculateGlyphCount()` function determines how many characters of a text string can fit within a given set of horizontal spans. The `spanCount` parameter specifies the number of horizontal spans to consider, and the `spanArray` parameter points to the array of span values.

The `compiledText` parameter should be a pointer to a `CompiledText` structure returned by a preceding call to the `CompileString()` function. A pointer to a `GlyphRange` structure may be passed to the `glyphRange` parameter to specify that only a subset of glyphs are to be processed. The pointer passed to the `fontHeader` parameter must be the same that was passed to the `fontHeader` parameter of the `CompileString()` function.

The number of glyphs that fit within each horizontal span is returned in the array specified by the `glyphCountArray` parameter. If the `glyphSpanArray` parameter is not `nullptr`, then it receives the actual width of the glyphs fitting into each span, which is no greater than the corresponding span supplied by the `spanArray` parameter.

If the `trimCount` parameter is not zero, then the `trimArray` parameter must point to an array of Unicode characters having the number of entries specified by `trimCount`. Values specified in this array typically include spaces and other characters that do not generate any geometry. Glyphs corresponding to these characters are not counted when they occur at the end of the sequence of glyphs that fit within each span, and they do not contribute to the actual spans returned through the `glyphSpanArray` parameter.

Any characters in the original text string designated as control characters by the Unicode standard do not contribute to the text length. These characters never contribute any spacing in the slug layout, even if the original font defines nonzero advance widths for them.

CalculateGlyphCountEx() function

The `CalculateGlyphCountEx()` function calculates the partial lengths of a line of compiled text that fit within specified physical horizontal spans.

Prototype

```
void CalculateGlyphCountEx(const CompiledText *compiledText,
                           const GlyphRange *glyphRange,
                           int32 fontCount,
                           const FontDesc *fontDesc,
                           int32 spanCount,
                           const float *spanArray,
                           int32 trimCount,
                           const uint32 *trimArray,
                           int32 *glyphCountArray,
                           float *glyphSpanArray = nullptr);
```

Parameters

Parameter	Description
`compiledText`	A pointer to a `CompiledText` object returned by a preceding call to the `CompileStringEx()` function.
`glyphRange`	A pointer to a `GlyphRange` structure specifying the range of glyphs to process. This parameter can be `nullptr`, in which case all of the glyphs stored in the `CompiledText` object are processed.
`fontCount`	The total number of fonts that may be utilized. This must be at least 1.
`fontDesc`	A pointer to an array of `FontDesc` structures describing the fonts that may be utilized. The number of elements in this array must be equal to the value of the `fontCount` parameter.
`spanCount`	The number of spans stored in the array specified by the `spanArray` parameter. This must be at least one.
`spanArray`	A pointer to an array of horizontal spans containing `spanCount` entries. The values in this array must be sorted in increasing order.
`trimCount`	The number of trim characters specified by the `trimArray` parameter.

`trimArray`	A pointer to an array of trim characters with `trimCount` entries. The values in this array are Unicode characters, and they must be sorted in ascending order. This parameter can `nullptr` only if the `trimCount` parameter is 0.
`glyphCountArray`	A pointer to an array having `spanCount` entries to which the glyph counts are written. This parameter cannot be `nullptr`.
`glyphSpanArray`	A pointer to an array having `spanCount` entries to which the actual spans are written. This parameter can be `nullptr`, in which case the spans not returned.

Description

The `CalculateGlyphCountEx()` function is an extended version of the `CalculateGlyphCount()` function capable of handling multiple fonts through the mapping mechanism described in Section 4.6. A call to the `CalculateGlyphCount()` function is internally forwarded to the `CalculateGlyphCountEx()` function with the `fontCount` parameter set to 1, the `fontDesc` parameter set to the address of a single `FontDesc` structure containing the font header with default scale and offset, and the `fontMap` parameter set to `nullptr`.

The `compiledText` parameter should be a pointer to a `CompiledText` structure returned by a preceding call to the `CompileStringEx()` function. A pointer to a `GlyphRange` structure may be passed to the `glyphRange` parameter to specify that only a subset of glyphs are to be processed. The value of the `fontCount` parameter and the entries of the array specified by the `fontDesc` parameter must be exactly the same values that were passed to the `fontCount` and `fontDesc` parameters of the `CompileStringEx()` function.

After the first four parameters, the remaining parameters passed to the `CalculateGlyphCountEx()` function have the same meanings as the parameters with the same names passed to the `CalculateGlyphCount()` function.

ColorData structure

The `ColorData` structure contains information about colors and gradients.

Fields

Field	Description
`Color4U color[2]`	The text colors. For each color, the red, green, and blue components are specified in the sRGB color space, and the alpha component is linear. If gradients are not used, then the first color is assigned to all glyph vertices, and the second color is ignored. If the `gradientFlag` field is `true`, then a gradient is applied using both colors. The alpha value should be 255 for fully opaque text.
`float gradient[2]`	The *y* coordinates at which the gradient color equals the corresponding entry in the color field. Each *y* coordinate is the em-space distance above the baseline. Negative values are allowed. The two *y* coordinates must not be equal. This field is used only if the `gradientFlag` field is `true`.
`bool gradientFlag`	A flag indicating whether the color gradient is used. If this is `false`, then text is rendered using only `color[0]` as a solid color. If this is `true`, then text is rendered with a gradient determined by `color[0]` and `color[1]` at the *y* coordinates given by `gradient[0]` and `gradient[1]`.

Description

The `ColorData` structure contains information about colors and gradients applied to a line of text. This structure is used in the `LayoutData` structure for the text color and effect color.

If the `gradientFlag` field is `false`, then the second entry in the `color` array and both entries in the `gradient` array are ignored. In this case, only the first entry in the `color` array is used, and it represents the solid color with which text is rendered.

If the `gradientFlag` field is `true`, then the two entries of the `color` array correspond to colors attained at the em-space *y* coordinates specified by the corresponding two entries of the `gradient` array.

CompiledCharacter structure

The `CompiledCharacter` structure holds information about one character in a compiled text string.

Fields

Field	Description
`int32 stringLocation`	The byte offset at which this character begins in the original string.
`uint32 unicodeValue`	The full 32-bit Unicode value of this character. This is zero for the null terminator.
`uint8 unicodeLength`	The number of bytes occupied in the original string by the UTF-8 encoding of this character.
`uint8 unicodeFlags`	Flags corresponding to various Unicode properties for this character.
`uint8 slugFlags`	Flags used internally for this character.

Description

When a text string is processed by the `CompileString()` function, an array of `CompiledCharacter` structures is stored in the returned `CompiledText` structure. Each entry of that array corresponds to a single Unicode character in the original string and contains the information described above.

The array of compiled characters in the `CompiledText` structure always ends with a special null terminator entry for which the `unicodeValue` field has the value zero. The value of the `stringLocation` field for the null terminator is the length of the original text string excluding the terminator.

The `unicodeFlags` field contains the flags that would be returned by the `GetUnicodeCharacterFlags()` function for the value in the `unicodeValue` field.

CompiledGlyph structure

The `CompiledGlyph` structure holds information about one glyph in a compiled text string.

Fields

Field	Description
`uint32` `glyphIndex`	The low 24 bits contain the index of the glyph within the font to which it belongs. The high 8 bits contain flags used internally. The null terminator has the value `kTerminatorGlyph`.
`int32` `characterNumber`	The index of the first compiled character to which this glyph corresponds.
`uint8` `characterCount`	The number of compiled characters to which this glyph corresponds.
`uint8` `fontIndex`	The index of the font that was chosen for this glyph.
`uint8` `layoutIndex`	The index of the `LayoutData` used by this glyph.
`uint8` `runIndex`	The index of the `RunData` used by this glyph.

Description

When a text string is processed by the `CompileString()` function, an array of `CompiledGlyph` structures is stored in the returned `CompiledText` structure. Each entry of that array corresponds to a single glyph in the final output and contains the information described above. Glyphs are generated after font selection, sequence replacement, and alternate substitution have been applied, and thus the final number of glyphs may be different than the original number of characters in the text string.

The array of `CompiledGlyph` structures in the `CompiledText` structure always ends with a special null terminator entry for which the `glyphIndex` field has the value `kTerminatorGlyph`.

CompiledText and CompiledStorage structures

The `CompiledText` structure stores information about the characters in a text string and the glyphs generated when that string was processed. The `CompiledStorage` structure is an extension of the `CompiledText` structure that includes the maximum storage space for the compiled information.

Fields

Field	Description
`int32 characterCount`	The number of compiled characters in the `compiledCharacter` array, excluding the null terminator.
`int32 glyphCount`	The number of compiled glyphs in the `compiledGlyph` array, excluding the null terminator.
`int32 layoutCount`	The number of different `LayoutData` structures in the `layoutData` array.
`int32 runCount`	The number of different `RunData` structures in the `runData` array.
`CompiledCharacter compiledCharacter[]`	An array of `CompiledCharacter` structures corresponding to the text string. The number of entries is the value of the `characterCount` field plus one more containing the null terminator.
`CompiledGlyph compiledGlyph[]`	An array of `CompiledGlyph` structures corresponding to the glyphs generated by the characters in the text string. The number of entries is the value of the `glyphCount` field plus one more containing the null terminator.
`LayoutData layoutData[]`	An array of `LayoutData` structures used by the text string. The number of entries is given by the `layoutCount` field. The `layoutIndex` field of the `CompiledGlyph` structure indexes into this array.
`RunData runData[]`	An array of `RunData` structures used by the text string. The number of entries is given by the `runCount` field. The `runIndex` field of the `CompiledGlyph` structure indexes into this array.

Description

The `CompiledText` structure serves as a header describing the contents of storage space for information generated by the `CompileString()` function. The `CompiledStorage` structure is composed of a `CompiledText` structure followed by the maximum storage space allowed for a single string of text. The compiled text includes information about the Unicode characters in the original string, the glyphs that were generated for those characters, the various layout states used by the text string, and each directional

run that occurred in the text string. Once this information has been compiled, it can be consumed by many of the Slug library functions.

When the library is used in a single-threaded context, there is normally no need for the application to allocate its own `CompiledStorage` structures because the library contains one that can be used internally. A pointer to the internal `CompiledText` structure is returned by the `CompileString()` function when the application does not specify its own storage in the final parameter.

If library functions are called from multiple threads, then the application must ensure that a different `CompiledStorage` structure is used by each thread so that the library is safely reentrant. The storage allocated by the application is passed as the final parameter of the `CompileString()` function. Due to the somewhat large size of `CompiledStorage` structures, they should not be allocated on the stack, but only on the heap or as a static part of the program binary.

CompileString() function

The `CompileString()` function processes a text string and compiles a list of glyphs.

Prototype

```
const CompiledText *CompileString(const FontHeader *fontHeader,
                                  const LayoutData *layoutData,
                                  const char *text,
                                  int32 maxLength = -1,
                                  LayoutData *exitLayoutData = nullptr,
                                  CompiledStorage *compiledStorage = nullptr);
```

Parameters

Parameter	Description
`fontHeader`	A pointer to the `FontHeader` structure retrieved with the `GetFontHeader()` function for a particular `.slug` file.
`layoutData`	A pointer to a `LayoutData` structure containing the initial text layout state that is applied.
`text`	A pointer to a text string. The characters must be encoded as UTF-8. If the `maxLength` parameter is –1, then this string must be null terminated.
`maxLength`	The maximum number of bytes to be processed in the string. If this is set to –1, then the string must be null terminated, and the entire string is processed.
`exitLayoutData`	A pointer to a `LayoutData` structure to which the updated text layout state is written. It is safe to specify the same pointer for both the `layoutData` and `exitLayoutData` parameters. This parameter can be `nullptr`, in which case the updated state is not returned.
`compiledStorage`	A pointer to a `CompiledStorage` structure that will be used for temporary storage. If this is `nullptr`, then internal storage shared among all callers is used. See the `CompiledStorage` structure for information about reentrancy.

Description

The `CompileString()` function processes the UTF-8 encoded text string specified by the `text` and `maxLength` parameters.

The `text` parameter should point to a string of characters encoded as UTF-8. The `maxLength` parameter specifies the maximum number of bytes of this string to be processed, which is not necessarily the

number of characters. If a null terminator is encountered before the number of bytes specified by `maxLength` has been processed, then processing stops at the null terminator. If `maxLength` is set to −1, then the string must be null terminated, and the entire string is processed.

If the `exitLayoutData` parameter is not `nullptr`, then the updated layout state is written to the location it points to. It is safe to specify the same pointer for both the `layoutData` and `exitLayoutData` parameters. The `kLayoutFormatDirectives` bit must be set in the `layoutFlags` field of the `LayoutData` structure specified by the `layoutData` parameter for any changes to be made to the layout state by embedded format directives as the text is processed.

If the `compiledStorage` parameter is not `nullptr`, then the return value is a pointer to the `CompiledText` base object of the specified storage. Otherwise, if the `compiledStorage` parameter is `nullptr`, then the return value is a pointer to the library's internal shared storage.

CompileStringEx() function

The `CompileStringEx()` function processes a text string and compiles a list of glyphs.

Prototype

```
const CompiledText *CompileStringEx(int32 fontCount,
                                    const FontDesc *fontDesc,
                                    const FontMap *fontMap,
                                    const LayoutData *layoutData,
                                    const char *text,
                                    int32 maxLength = -1,
                                    LayoutData *exitLayoutData = nullptr,
                                    CompiledStorage *compiledStorage = nullptr);
```

Parameters

Parameter	Description
`fontCount`	The total number of fonts that may be utilized. This must be at least 1.
`fontDesc`	A pointer to an array of `FontDesc` structures describing the fonts that may be utilized. The number of elements in this array must be equal to the value of the `fontCount` parameter.
`fontMap`	A pointer to a `FontMap` structure defining the relationships among font types and the source fonts specified in the font header array. This can be `nullptr`, in which case only the font with index 0 is used.
`layoutData`	A pointer to a `LayoutData` structure containing the initial text layout state that is applied.
`text`	A pointer to a text string. The characters must be encoded as UTF-8. If the `maxLength` parameter is –1, then this string must be null terminated.
`maxLength`	The maximum number of bytes to be processed in the string. If this is set to –1, then the string must be null terminated, and the entire string is processed.
`exitLayoutData`	A pointer to a `LayoutData` structure to which the updated text layout state is written. It is safe to specify the same pointer for both the `layoutData` and `exitLayoutData` parameters. This parameter can be `nullptr`, in which case the updated state is not returned.

`compiledStorage`	A pointer to a `CompiledStorage` structure that will be used for temporary storage. If this is `nullptr`, then internal storage shared among all callers is used. See the `CompiledStorage` structure for information about reentrancy.

Description

The `CompileStringEx()` function is an extended version of the `CompileString()` function capable of handling multiple fonts through the mapping mechanism described in Section 4.6. A call to the `CompileString()` function is internally forwarded to the `CompileStringEx()` function with the `fontCount` parameter set to 1, the `fontDesc` parameter set to the address of a single `FontDesc` structure containing the font header with default scale and offset, and the `fontMap` parameter set to `nullptr`.

The `fontCount` and `fontDesc` parameters specify the master font list containing the full set of fonts that can be used with the text string. The `fontMap` parameter specifies the mapping from generic font types to the actual fonts as described for the `FontMap` structure.

After the first three parameters, the remaining parameters passed to the `CompileStringEx()` function have the same meanings as the parameters with the same names passed to the `CompileString()` function.

CountFill() function

The `CountStroke()` function calculates the numbers of vertices and triangles that will be generated by the `CreateStroke()` function for a single filled path and determines how much space will be used in the curve and band textures.

Prototype

```
void CountFill(const FillData *fillData,
               int32 curveCount,
               const QuadraticBezier2D *curveArray,
               const Integer2D& curveTextureSize,
               Integer2D *curveWriteLocation,
               const Integer2D& bandTextureSize,
               Integer2D *bandWriteLocation,
               int32 *vertexCount,
               int32 *triangleCount,
               const CreateData *createData = nullptr,
               FillWorkspace *workspace = nullptr);
```

Parameters

Parameter	Description
`fillData`	A pointer to a `FillData` structure containing the fill state that is applied.
`curveCount`	The number of Bézier curves in the filled path. This must be no greater than `kMaxFillCurveCount`, which is defined to be 8192.
`curveArray`	A pointer to an array of quadratic Bézier curves containing as many entries as specified by the `curveCount` parameter. This array must specify a closed loop, and the first control point of each curve must be exactly equal to the last control point of the preceding curve.
`curveTextureSize`	The *x* and *y* dimensions of the curve texture, in texels, to which control point data will be written by the `CreateFill()` function. If the *y* size is greater than one, then the *x* size must be 4096.
`curveWriteLocation`	On entry, the initial texture coordinates at which control point data will be written in the curve texture. On exit, the new coordinates following the data that will be written.

`bandTextureSize`	The *x* and *y* dimensions of the band texture, in texels, to which band data will be written by the `CreateFill()` function. If the *y* size is greater than one, then the *x* size must be 4096.
`bandWriteLocation`	On entry, the initial texture coordinates at which band data will be written in the band texture. On exit, the new coordinates following the data that will be written.
`vertexCount`	A pointer to the location that receives the number of vertices that will be generated by the `CreateFill()` function.
`triangleCount`	A pointer to the location that receives the number of triangles that will be generated by the `CreateFill()` function.
`createData`	A pointer to a `CreateData` structure specifying optimization parameters for the filled path. If this is `nullptr`, then the default parameters are applied.
`workspace`	A pointer to a `FillWorkspace` structure that will be used for temporary storage. If this is `nullptr`, then an internal workspace shared among all callers is used. See the `FillWorkspace` structure for information about reentrancy.

Description

The `CountFill()` function calculates the numbers of vertices and triangles that will be generated and the amount of space in the curve and band textures that will be used by the `CreateFill()` function for a specific fill data, create data, and set of Bézier curves. The `CountFill()` function must be called before vertex and triangle data can be generated so that the appropriate amount of storage space can be allocated.

The appearance of the interior of the filled path is determined by the data supplied in the `FillData` structure specified by the `fillData` parameter. This structure can specify a solid fill color or a gradient. The `SetDefaultFillData()` function should be used to initialize the `FillData` structure to its default values before setting individual fields.

The path itself is defined by an array of quadratic Bézier curves given by the `curveCount` and `curveArray` parameters. The set of curves must form a closed loop, and the first control point of each curve must be exactly equal to the last control point of the preceding curve. The last curve in the array is considered to be the predecessor of the first curve in the array, and thus the first control point of the first curve must be exactly equal to the last control point of the last curve.

The `curveTextureSize` and `bandTextureSize` parameters specify the *x* and *y* dimensions of the curve texture and band texture. The `curveWriteLocation` and `bandWriteLocation` parameters specify the coordinates in those textures at which new data will be written by the `CreateFill()` function. These should be set to (0, 0) initially and should simply retain their output values for any subsequent calls to the `CountFill()` or `CountStroke()` functions. When the `CountFill()` function returns, the write

locations are updated to reflect the amount of space that will be required. The actual texture storage passed to the `CreateFill()` function must have a height at least one greater than the *y* coordinate of the final write location for each of the curve and band textures.

The numbers of vertices and triangles are stored in the locations pointed to by the `vertexCount` and `triangleCount` parameters. The vertex count and triangle count are always returned, and the corresponding parameters cannot be `nullptr`. This information should be used to allocate vertex buffers of the proper size before calling the `CreateFill()` function to fill them with data.

The `createData` parameter points to a `CreateData` structure containing optimization settings for the internal data used for rendering. If this parameter is `nullptr`, then the default values listed in the description of the `CreateData` structure are applied.

CountIcon() function

The `CountIcon()` function calculates the maximum numbers of vertices and triangles that will be generated by the `BuildIcon()` function for a single icon.

Prototype

```
int32 CountIcon(const IconData *iconData,
                GeometryType geometryType,
                int32 *vertexCount,
                int32 *triangleCount);
```

Parameters

Parameter	Description
`iconData`	A pointer to the `IconData` structure corresponding to the icon.
`geometryType`	The type of geometry that will be used to render the icon. See the description for information about the various types.
`vertexCount`	A pointer to the location that receives the total number of vertices that will be generated by the `BuildIcon()` function.
`triangleCount`	A pointer to the location that receives the total number of triangles that will be generated by the `BuildIcon()` function.

Description

The `CountIcon()` function calculates the maximum numbers of vertices and triangles that will be generated by the `BuildIcon()` function for a specific icon. The `CountIcon()` function must be called before vertex and triangle data can be generated so that the appropriate amount of storage space can be allocated.

The `iconData` parameter must point to an `IconData` structure returned by the `GetIconData()` function or generated by either the `ImportIconData()` or `ImportMulticolorIconData()` function specifically for the icon being rendered.

The maximum number of vertices and maximum number of triangles are stored in the locations pointed to by the `vertexCount` and `triangleCount` parameters. The vertex count and triangle count are always returned, and the corresponding parameters cannot be `nullptr`. This information should be used to allocate vertex buffers of the proper size before calling the `BuildIcon()` function to fill them with data.

The following values are the geometry types that can be specified for the `geometryType` parameter.

Value	Description
`kGeometryQuads`	The icon is rendered with a single quad composed of 4 vertices and 2 triangles.
`kGeometryPolygons`	The icon is rendered with a tight bounding polygon having between 3 and 8 vertices and between 1 and 6 triangles.
`kGeometryRectangles`	The icon is rendered with 3 vertices making up exactly one triangle with the expectation that the window-aligned bounding rectangle will be filled. This type can be used only when rectangle primitives are available, such as provided by the `VK_NV_fill_rectangle` and `GL_NV_fill_rectangle` extensions.

CountMultiLineText() function

The `CountMultiLineText()` function calculates the maximum numbers of vertices and triangles that will be generated by the `BuildMultiLineText()` function for multiple lines of text.

Prototype

```
int32 CountMultiLineText(const CompiledText *compiledText,
                         const FontHeader *fontHeader,
                         int32 lineIndex,
                         int32 lineCount,
                         const LineData *lineDataArray,
                         int32 *vertexCount,
                         int32 *triangleCount,
                         int32 *placeholderCount = nullptr);
```

Parameters

Parameter	Description
`compiledText`	A pointer to a `CompiledText` object returned by a preceding call to the `CompileString()` function.
`fontHeader`	A pointer to the `FontHeader` structure retrieved with the `GetFontHeader()` function for a particular `.slug` file.
`lineIndex`	The zero-based index of the first line of text to count.
`lineCount`	The number of lines of text to count.
`lineDataArray`	A pointer to an array of `LineData` structures containing information about each line of text. The lines of text to be built correspond to elements indexed `lineIndex` through `lineIndex + lineCount - 1` in this array.
`vertexCount`	A pointer to the location that receives the maximum number of vertices that will be generated by the `BuildMultiLineText()` function.
`triangleCount`	A pointer to the location that receives the maximum number of triangles that will be generated by the `BuildMultiLineText()` function.
`placeholderCount`	A pointer to the location that receives the number of placeholders that will be generated by the `BuildMultiLineText()` function. This parameter can be `nullptr`, in which case the placeholder count is not returned.

Description

The `CountMultiLineText()` function calculates the maximum numbers of vertices and triangles that will be generated by the `BuildMultiLineText()` function for a specific font, layout state, and text string. The `CountMultiLineText()` function must be called before vertex and triangle data can be generated so that the appropriate amount of storage space can be allocated.

The `compiledText` parameter should be a pointer to a `CompiledText` structure returned by a preceding call to the `CompileString()` function. The pointer passed to the `fontHeader` parameter must be the same that was passed to the `fontHeader` parameter of the `CompileString()` function.

The `lineIndex` parameter specifies the zero-based index of the first line of text to count, and the `lineCount` parameter specifies the number of lines to count. The `lineDataArray` parameter must point to an array of `LineData` structures containing at least `lineIndex + lineCount` elements. These would normally have been generated by a previous call to the `BreakMultiLineText()` function.

If the `lineIndex` parameter is not zero and embedded format directives are enabled, then it is the caller's responsibility to ensure that the `LayoutData` structure specified by the `layoutData` parameter has been updated to match the correct state for the first line of text. This can be accomplished by calling the `UpdateLayoutData()` function with a `maxLength` parameter given by the `fullTextLength` field of `lineDataArray[lineIndex - 1]`. This layout state will normally be passed to the `BuildMultiLineText()` function as well, so the `UpdateLayoutData()` function would typically be called once to generate a layout state that is passed to both functions.

The maximum number of vertices and maximum number of triangles are stored in the locations pointed to by the `vertexCount` and `triangleCount` parameters. The vertex count and triangle count are always returned, and the corresponding parameters cannot be `nullptr`. This information should be used to allocate vertex buffers of the proper size before calling the `BuildMultiLineText()` function to fill them with data. The `BuildMultiLineText()` function may end up generating fewer than the maximum numbers of vertices and triangles returned by the `CountMultiLineText()` function depending on various factors.

> **Note:** If the value returned in the `vertexCount` parameter is greater than 65535, then the text is too large to be rendered with the 16-bit vertex indices stored in the `Triangle` structure. In this case, the 32-bit vertex indices provided by the `Triangle32` structure must be used instead.

If the `placeholderCount` parameter is not `nullptr`, then the number of placeholders is stored in the location that it points to. This information should be used to allocate an array of `PlaceholderData` structures of the proper size before calling the `BuildSlug()` function with a `placeholderBuffer` parameter that is not `nullptr`.

The value returned by the `CountMultiLineText()` function is the number of individual glyphs generated for the text string, which can be different from the number of characters. The glyph count includes glyphs that do not have any geometry, such as the glyph corresponding to the space character. Underline and strikethrough decorations do not affect the glyph count.

Any characters in the original text string designated as control characters by the Unicode standard do not generate any output. These characters never contribute to the vertex and triangle counts, and they never cause any geometry to be generated by the `BuildMultiLineText()` function.

CountMultiLineTextEx() function

The `CountMultiLineTextEx()` function calculates the maximum numbers of vertices and triangles that will be generated by the `BuildMultiLineTextEx()` function for multiple lines of text.

Prototype

```
int32 CountMultiLineTextEx(const CompiledText *compiledText,
                           int32 fontCount,
                           const FontDesc *fontDesc,
                           int32 lineIndex,
                           int32 lineCount,
                           const LineData *lineDataArray,
                           int32 *vertexCount,
                           int32 *triangleCount,
                           int32 *placeholderCount = nullptr);
```

Parameters

Parameter	Description
`compiledText`	A pointer to a `CompiledText` object returned by a preceding call to the `CompileStringEx()` function.
`fontCount`	The total number of fonts that may be utilized. This must be at least 1.
`fontDesc`	A pointer to an array of `FontDesc` structures describing the fonts that may be utilized. The number of elements in this array must be equal to the value of the `fontCount` parameter.
`lineIndex`	The zero-based index of the first line of text to count.
`lineCount`	The number of lines of text to count.
`lineDataArray`	A pointer to an array of `LineData` structures containing information about each line of text. The lines of text to be built correspond to elements indexed `lineIndex` through `lineIndex + lineCount - 1` in this array.
`vertexCount`	A pointer to an array of values that receives the maximum number of vertices that will be generated by the `BuildMultiLineTextEx()` function for each font. The number of elements in this array must be equal to the value of the `fontCount` parameter.

`triangleCount`	A pointer to an array of values that receives the maximum number of triangles that will be generated by the `BuildMultiLineTextEx()` function for each font. The number of elements in this array must be equal to the value of the `fontCount` parameter.
`placeholderCount`	A pointer to the location that receives the number of placeholders that will be generated by the `BuildMultiLineTextEx()` function. This parameter can be `nullptr`, in which case the placeholder count is not returned.

Description

The `CountMultiLineTextEx()` function is an extended version of the `CountMultiLineText()` function capable of handling multiple fonts through the mapping mechanism described in Section 4.6. A call to the `CountMultiLineText()` function is internally forwarded to the `CountMultiLineTextEx()` function with the `fontCount` parameter set to 1, the `fontDesc` parameter set to the address of a single `FontDesc` structure containing the font header with default scale and offset, and the `fontMap` parameter set to `nullptr`.

The `compiledText` parameter should be a pointer to a `CompiledText` structure returned by a preceding call to the `CompileStringEx()` function. The value of the `fontCount` parameter and the entries of the array specified by the `fontDesc` parameter must be exactly the same values that were passed to the `fontCount` and `fontDesc` parameters of the `CompileStringEx()` function.

After the first three parameters, the remaining parameters passed to the `CountMultiLineTextEx()` function have the same meanings as the parameters with the same names passed to the `CountMultiLineText()` function with two exceptions. The `vertexCount` and `triangleCount` parameters must now each point to an array of values having one entry per font. Each entry in these arrays receives the vertex count or triangle count for the corresponding font index. Keep in mind that it is possible for no geometry to be generated for some fonts if they are not used in the portion of the string processed by the `CountMultiLineTextEx()` function.

CountPicture() function

The `CountPicture()` function calculates the maximum numbers of vertices and triangles that will be generated by the `BuildPicture()` function for an entire picture.

Prototype

```
int32 CountPicture(const AlbumHeader *albumHeader,
                   int32 pictureIndex,
                   GeometryType geometryType,
                   int32 *vertexCount,
                   int32 *triangleCount);
```

Parameters

Parameter	Description
`albumHeader`	A pointer to the `AlbumHeader` structure retrieved with the `GetAlbumHeader()` function for a particular `.slug` file.
`pictureIndex`	The index of the picture within the album. This index must be between 0 and one less than the count given by the `pictureCount` field of the `AlbumHeader` structure, inclusive.
`geometryType`	The type of geometry that will be used to render the picture. See the description for information about the various types.
`vertexCount`	A pointer to the location that receives the total number of vertices that will be generated by the `BuildPicture()` function.
`triangleCount`	A pointer to the location that receives the total number of triangles that will be generated by the `BuildPicture()` function.

Description

The `CountPicture()` function calculates the maximum numbers of vertices and triangles that will be generated by the `BuildPicture()` function for a specific icon. The `CountPicture()` function must be called before vertex and triangle data can be generated so that the appropriate amount of storage space can be allocated.

The maximum number of vertices and maximum number of triangles are stored in the locations pointed to by the `vertexCount` and `triangleCount` parameters. The vertex count and triangle count are always returned, and the corresponding parameters cannot be `nullptr`. This information should be used to

allocate vertex buffers of the proper size before calling the `BuildPicture()` function to fill them with data.

> **Note:** If the value returned in the `vertexCount` parameter is greater than 65535, then the picture is too large to be rendered with the 16-bit vertex indices stored in the `Triangle` structure. In this case, the 32-bit vertex indices provided by the `Triangle32` structure must be used instead.

The following values are the geometry types that can be specified for the `geometryType` parameter.

Value	Description
`kGeometryQuads`	The components of the picture are each rendered with a single quad composed of 4 vertices and 2 triangles.
`kGeometryPolygons`	The components of the picture are each rendered with a tight bounding polygon having between 3 and 8 vertices and between 1 and 6 triangles.
`kGeometryRectangles`	The components of the picture are each rendered with 3 vertices making up exactly one triangle with the expectation that the window-aligned bounding rectangle will be filled. This type can be used only when rectangle primitives are available, such as provided by the `VK_NV_fill_rectangle` and `GL_NV_fill_rectangle` extensions.

CountSlug() function

The `CountSlug()` function calculates the maximum numbers of vertices and triangles that will be generated by the `BuildSlug()` function for a single line of text, or "slug".

Prototype

```
int32 CountSlug(const CompiledText *compiledText,
                const GlyphRange *glyphRange,
                const FontHeader *fontHeader,
                int32 *vertexCount,
                int32 *triangleCount,
                int32 *placeholderCount = nullptr);
```

Parameters

Parameter	Description
`compiledText`	A pointer to a `CompiledText` object returned by a preceding call to the `CompileString()` function.
`glyphRange`	A pointer to a `GlyphRange` structure specifying the range of glyphs to process. This parameter can be `nullptr`, in which case all of the glyphs stored in the `CompiledText` object are processed.
`fontHeader`	A pointer to the `FontHeader` structure retrieved with the `GetFontHeader()` function for a particular `.slug` file.
`vertexCount`	A pointer to the location that receives the maximum number of vertices that will be generated by the `BuildSlug()` function.
`triangleCount`	A pointer to the location that receives the maximum number of triangles that will be generated by the `BuildSlug()` function.
`placeholderCount`	A pointer to the location that receives the number of placeholders that will be generated by the `BuildSlug()` function. This parameter can be `nullptr`, in which case the placeholder count is not returned.

Description

The `CountSlug()` function calculates the maximum numbers of vertices and triangles that will be generated by the `BuildSlug()` function for a specific font, layout state, and text string. The `CountSlug()` function must be called before vertex and triangle data can be generated so that the appropriate amount of storage space can be allocated.

The `compiledText` parameter should be a pointer to a `CompiledText` structure returned by a preceding call to the `CompileString()` function. A pointer to a `GlyphRange` structure may be passed to the `glyphRange` parameter to specify that only a subset of glyphs are to be processed. The pointer passed to the `fontHeader` parameter must be the same that was passed to the `fontHeader` parameter of the `CompileString()` function.

The maximum number of vertices and maximum number of triangles are stored in the locations pointed to by the `vertexCount` and `triangleCount` parameters. The vertex count and triangle count are always returned, and the corresponding parameters cannot be `nullptr`. This information should be used to allocate vertex buffers of the proper size before calling the `BuildSlug()` function to fill them with data. The `BuildSlug()` function may end up generating fewer than the maximum numbers of vertices and triangles returned by the `CountSlug()` function depending on various factors.

> **Note:** If the value returned in the `vertexCount` parameter is greater than 65535, then the text is too large to be rendered with the 16-bit vertex indices stored in the `Triangle` structure. In this case, the 32-bit vertex indices provided by the `Triangle32` structure must be used instead.

If the `placeholderCount` parameter is not `nullptr`, then the number of placeholders is stored in the location that it points to. This information should be used to allocate an array of `PlaceholderData` structures of the proper size before calling the `BuildSlug()` function with a `placeholderBuffer` parameter that is not `nullptr`.

The value returned by the `CountSlug()` function is the number of individual glyphs generated for the text string, which can be different from the number of characters. The glyph count includes glyphs that do not have any geometry, such as the glyph corresponding to the space character. Underline and strikethrough decorations do not affect the glyph count.

Any characters in the original text string designated as control characters by the Unicode standard do not generate any output. These characters never contribute to the vertex and triangle counts, and they never cause any geometry to be generated by the `BuildSlug()` function.

CountSlugEx() function

The `CountSlugEx()` function calculates the maximum numbers of vertices and triangles that will be generated by the `BuildSlugEx()` function for a single line of text, or "slug".

Prototype

```
int32 CountSlugEx(const CompiledText *compiledText,
                  const GlyphRange *glyphRange,
                  int32 fontCount,
                  const FontDesc *fontDesc,
                  int32 *vertexCount,
                  int32 *triangleCount,
                  int32 *placeholderCount = nullptr);
```

Parameters

Parameter	Description
`compiledText`	A pointer to a `CompiledText` object returned by a preceding call to the `CompileStringEx()` function.
`glyphRange`	A pointer to a `GlyphRange` structure specifying the range of glyphs to process. This parameter can be `nullptr`, in which case all of the glyphs stored in the `CompiledText` object are processed.
`fontCount`	The total number of fonts that may be utilized. This must be at least 1.
`fontDesc`	A pointer to an array of `FontDesc` structures describing the fonts that may be utilized. The number of elements in this array must be equal to the value of the `fontCount` parameter.
`vertexCount`	A pointer to an array of values that receives the maximum number of vertices that will be generated by the `BuildSlugEx()` function for each font. The number of elements in this array must be equal to the value of the `fontCount` parameter.
`triangleCount`	A pointer to an array of values that receives the maximum number of triangles that will be generated by the `BuildSlugEx()` function for each font. The number of elements in this array must be equal to the value of the `fontCount` parameter.

`placeholderCount`	A pointer to the location that receives the number of placeholders that will be generated by the `BuildSlugEx()` function. This parameter can be `nullptr`, in which case the placeholder count is not returned.

Description

The `CountSlugEx()` function is an extended version of the `CountSlug()` function capable of handling multiple fonts through the mapping mechanism described in Section 4.6. A call to the `CountSlug()` function is internally forwarded to the `CountSlugEx()` function with the `fontCount` parameter set to 1, the `fontDesc` parameter set to the address of a single `FontDesc` structure containing the font header with default scale and offset, and the `fontMap` parameter set to `nullptr`.

The `compiledText` parameter should be a pointer to a `CompiledText` structure returned by a preceding call to the `CompileStringEx()` function. A pointer to a `GlyphRange` structure may be passed to the `glyphRange` parameter to specify that only a subset of glyphs are to be processed. The value of the `fontCount` parameter and the entries of the array specified by the `fontDesc` parameter must be exactly the same values that were passed to the `fontCount` and `fontDesc` parameters of the `CompileStringEx()` function.

After the first four parameters, the remaining parameters passed to the `CountSlugEx()` function have the same meanings as the parameters with the same names passed to the `CountSlug()` function with two exceptions. The `vertexCount` and `triangleCount` parameters must now each point to an array of values having one entry per font. Each entry in these arrays receives the vertex count or triangle count for the corresponding font index. Keep in mind that it is possible for no geometry to be generated for some fonts if they are not used in the portion of the string processed by the `CountSlugEx()` function.

CountStroke() function

The `CountStroke()` function calculates the numbers of vertices and triangles that will be generated by the `CreateStroke()` function for a single stroked path and determines how much space will be used in the curve texture.

Prototype

```
void CountStroke(const StrokeData *strokeData,
                 uint32 strokeFlags,
                 int32 curveCount,
                 const QuadraticBezier2D *curveArray,
                 const Integer2D& curveTextureSize,
                 Integer2D *curveWriteLocation,
                 int32 *vertexCount,
                 int32 *triangleCount,
                 StrokeWorkspace *workspace = nullptr);
```

Parameters

Parameter	Description
`strokeData`	A pointer to a `StrokeData` structure containing the stroke state that is applied.
`strokeFlags`	Flags specifying properties of the stroked path. See the description for information about the particular values.
`curveCount`	The number of Bézier curves in the stroke. This must be no greater than `kMaxStrokeCurveCount`, which is defined to be 8192.
`curveArray`	A pointer to an array of quadratic Bézier curves containing as many entries as specified by the `curveCount` parameter. For all but the first curve in the array, the first control point of each curve must be exactly equal to the last control point of the preceding curve.
`curveTextureSize`	The *x* and *y* dimensions of the curve texture, in texels, to which control point data will be written by the `CreateStroke()` function. If the *y* size is greater than one, then the *x* size must be 4096.
`curveWriteLocation`	On entry, the initial texture coordinates at which control point data will be written in the curve texture. On exit, the new coordinates following the data that will be written.

`vertexCount`	A pointer to the location that receives the maximum number of vertices that will be generated by the `CreateStroke()` function.
`triangleCount`	A pointer to the location that receives the maximum number of triangles that will be generated by the `CreateStroke()` function.
`workspace`	A pointer to a `StrokeWorkspace` structure that will be used for temporary storage. If this is `nullptr`, then an internal workspace shared among all callers is used. See the `StrokeWorkspace` structure for information about reentrancy.

Description

The `CountStroke()` function calculates the numbers of vertices and triangles that will be generated and the amount of space in the curve texture that will be used by the `CreateStroke()` function for a specific stroke data, stroke flags, and set of Bézier curves. The `CountStroke()` function must be called before vertex and triangle data can be generated so that the appropriate amount of storage space can be allocated.

The appearance of the stroked path is determined by the data supplied in the `StrokeData` structure specified by the `strokeData` parameter and the value of the `strokeFlags` parameter. The `StrokeData` structure contains information about the stroke width, the stroke color, the cap style, the join style, and dashing.

The path itself is defined by an array of quadratic Bézier curves given by the `curveCount` and `curveArray` parameters. The set of curves must form a closed loop, and the first control point of each curve must be exactly equal to the last control point of the preceding curve. The last curve in the array is considered to be the predecessor of the first curve in the array, and thus the first control point of the first curve must be exactly equal to the last control point of the last curve.

The `curveTextureSize` parameter specifies the x and y dimensions of the curve texture. The `curveWriteLocation` parameter specifies the coordinates in the curve texture at which new data will be written by the `CreateStroke()` function. This should be set to (0, 0) initially and should simply retain its output values for any subsequent calls to the `CountStroke()` or `CountFill()` functions. When the `CountStroke()` function returns, the write location is updated to reflect the amount of space that will be required. The actual texture storage passed to the `CreateStroke()` function must have a height at least one greater than the y coordinate of the final write location for the curve texture.

The numbers of vertices and triangles are stored in the locations pointed to by the `vertexCount` and `triangleCount` parameters. The vertex count and triangle count are always returned, and the corresponding parameters cannot be `nullptr`. This information should be used to allocate vertex buffers of the proper size before calling the `CreateStroke()` function to fill them with data.

CreateData structure

The `CreateData` structure specifies optimization parameters used when creating a filled path with the `CreateFill()` function.

Fields

Field	Description
`int32` `maxBandCount`	The maximum number of horizontal and vertical bands created for a filled path. This must be at least one and no more than `kMaxFillBandCount`, which is defined to be 32.
`int32` `maxVertexCount`	The maximum number of vertices that may be used to form the bounding polygon for a filled path. To indicate that an optimal bounding polygon should be created, this must be in the range 4–6. To indicate that a filled path should always be rendered with a simple bounding box, this should be set to zero.
`float` `interiorEdgeFactor`	The cost factor used when considering the lengths of interior edges while searching for the optimal bounding polygon. Ignored if the `maxVertexCount` field is zero.

Description

The `CreateData` structure specifies optimization parameters used when creating a filled path with the `CreateFill()` function.

The default `CreateData` structure, used when `nullptr` is passed to the `CreateFill()` function, contains the following values.

Field	Default Value
`maxBandCount`	16
`maxVertexCount`	0
`interiorEdgeFactor`	1.0

CreateFill() function

The `CreateFill()` function generates the vertices and triangles for a single filled path.

Prototype

```
void CreateFill(const FillData *strokeData,
                int32 curveCount,
                const QuadraticBezier2D *curveArray,
                TextureBuffer *curveTextureBuffer,
                TextureBuffer *bandTextureBuffer,
                GeometryBuffer *geometryBuffer,
                const CreateData *createData = nullptr,
                FillWorkspace *workspace = nullptr);
```

Parameters

Parameter	Description
`fillData`	A pointer to a `FillData` structure containing the fill state that is applied.
`curveCount`	The number of Bézier curves in the filled path. This must be no greater than `kMaxFillCurveCount`, which is defined to be 8192.
`curveArray`	A pointer to an array of quadratic Bézier curves containing as many entries as specified by the `curveCount` parameter. This array must specify a closed loop, and the first control point of each curve must be exactly equal to the last control point of the preceding curve.
`curveTextureBuffer`	A pointer to a `TextureBuffer` structure describing the curve texture map.
`bandTextureBuffer`	A pointer to a `TextureBuffer` structure describing the band texture map.
`geometryBuffer`	A pointer to the `GeometryBuffer` structure containing information about where the output vertex and triangle data is stored.
`createData`	A pointer to a `CreateData` structure specifying optimization parameters for the filled path. If this is `nullptr`, then the default parameters are applied.
`workspace`	A pointer to a `FillWorkspace` structure that will be used for temporary storage. If this is `nullptr`, then an internal workspace shared among all callers is used. See the `FillWorkspace` structure for information about reentrancy.

Description

The `CreateFill()` function generates all of the internal data needed by Slug to render an arbitrary filled path. This function generates vertices, triangles, control points, and band data that are stored in vertex buffers and texture maps supplied by the application. This data is written in a format that is meant to be consumed directly by the GPU, and it can immediately be used for rendering. Multiple filled and stroked paths can be accumulated in the same output buffers and rendered as a single unit.

Before the `CreateFill()` function can be called, the `CountFill()` function must be called to determine the amount of storage that the `CreateFill()` function will need to write its data. The fill data, create data, and path must be exactly the same for both functions to ensure that the correct amount of storage can be allocated and that the data generated by the `CreateFill()` function stays within the calculated limits. For both functions, the `fillData` and `createData` parameters must point to `FillData` and `CreateData` structures containing identical information, the `curveCount` parameters must be equal, and the `curveArray` parameters must point to identical paths.

The appearance of the interior of the filled path is determined by the data supplied in the `FillData` structure specified by the `fillData` parameter. This structure can specify a solid fill color or a gradient. The `SetDefaultFillData()` function should be used to initialize the `FillData` structure to its default values before setting individual fields.

The path itself is defined by an array of quadratic Bézier curves given by the `curveCount` and `curveArray` parameters. The set of curves must form a closed loop, and the first control point of each curve must be exactly equal to the last control point of the preceding curve. The last curve in the array is considered to be the predecessor of the first curve in the array, and thus the first control point of the first curve must be exactly equal to the last control point of the last curve.

The `curveTextureBuffer` and `bandTextureBuffer` parameters point to `TextureBuffer` structures containing information about the texture maps and the locations where new curve and band data is written. The fields of the `TextureBuffer` structures must be initialized with exactly the same values that were used in previous corresponding calls to the `CountFill()` function. Upon return from the `CreateFill()` function, the write locations for each texture map are updated so they point to the next place that new data can be written. As with the vertex data, this allows for multiple paths to be stored in the same texture maps.

The `geometryBuffer` parameter points to a `GeometryBuffer` structure containing the addresses of the storage into which vertex and triangle data is written. These addresses are typically in memory that is visible to the GPU. Upon return from the `CreateFill()` function, the `GeometryBuffer` structure is updated so that the `vertexData` and `triangleData` fields point to the next element past the end of the data that was written. The `vertexIndex` field is advanced to one greater than the largest vertex index written. This updated information allows for multiple filled and/or stroked paths to be built in the same vertex buffer and drawn with a single rendering command.

The `createData` parameter points to a `CreateData` structure containing optimization settings for the internal data used for rendering. If this parameter is `nullptr`, then the default values listed in the description of the `CreateData` structure are applied.

CreateStroke() function

The `CreateStroke()` function generates the vertices and triangles for a single stroked path.

Prototype

```
void CreateStroke(const StrokeData *strokeData,
                  uint32 strokeFlags,
                  int32 curveCount,
                  const QuadraticBezier2D *curveArray,
                  TextureBuffer *textureBuffer,
                  GeometryBuffer *geometryBuffer,
                  StrokeWorkspace *workspace = nullptr);
```

Parameters

Parameter	Description
`strokeData`	A pointer to a `StrokeData` structure containing the stroke state that is applied.
`strokeFlags`	Flags specifying properties of the stroked path. See the description for information about the particular values.
`curveCount`	The number of Bézier curves in the stroke. This must be no greater than `kMaxStrokeCurveCount`, which is defined to be 8192.
`curveArray`	A pointer to an array of quadratic Bézier curves containing as many entries as specified by the `curveCount` parameter. For all but the first curve in the array, the first control point of each curve must be exactly equal to the last control point of the preceding curve.
`textureBuffer`	A pointer to a `TextureBuffer` structure describing the curve texture map.
`geometryBuffer`	A pointer to the `GeometryBuffer` structure containing information about where the output vertex and triangle data is stored.
`workspace`	A pointer to a `StrokeWorkspace` structure that will be used for temporary storage. If this is `nullptr`, then an internal workspace shared among all callers is used. See the `StrokeWorkspace` structure for information about reentrancy.

Description

The `CreateStroke()` function generates all of the internal data needed by Slug to render an arbitrary stroked path. This function generates vertices, triangles, and control points that are stored in vertex buffers and texture maps supplied by the application. This data is written in a format that is meant to be

consumed directly by the GPU, and it can immediately be used for rendering. Multiple filled and stroked paths can be accumulated in the same output buffers and rendered as a single unit.

Before the `CreateStroke()` function can be called, the `CountStroke()` function must be called to determine the amount of storage that the `CreateStroke()` function will need to write its data. The stroke data, stroke flags, and path must be exactly the same for both functions to ensure that the correct amount of storage can be allocated and that the data generated by the `CreateStroke()` function stays within the calculated limits. For both functions, the `strokeData` parameters must point to `StrokeData` structures containing identical information, the `strokeFlags` parameters must be equal, the `curveCount` parameters must be equal, and the `curveArray` parameters must point to identical paths.

The appearance of the stroked path is determined by the data supplied in the `StrokeData` structure specified by the `strokeData` parameter and the value of the `strokeFlags` parameter. The `StrokeData` structure contains information about the stroke width, the stroke color, the cap style, the join style, and dashing.

The path itself is defined by an array of quadratic Bézier curves given by the `curveCount` and `curveArray` parameters. The set of curves may be open or closed, but in all cases must be continuous. The first control point of each curve must be exactly equal to the last control point of the preceding curve. If the path is closed, meaning that the last control point of the last curve is equal to the first control point of the first curve, then the first and last curves are joined only if the `kStrokeClosed` flag is specified in the `strokeFlags` parameter. If the `kStrokeClosed` flag is not specified, then caps are applied in the same way they would be for an open path.

The following values can be combined (through logical OR) in the `strokeFlags` parameter.

Value	Description
`kStrokeClosed`	If the strokes form a closed path, then the beginning and end of the path are joined as if the first and last curves occurred consecutively.
`kStrokeContours`	The strokes may contain multiple closed contours. Whenever a closed subpath is detected, a discontinuity is created, and a new path begins.

The `curveTextureBuffer` parameter points to a `TextureBuffer` structure containing information about the texture map and the location where new curve data is written. (Strokes do not generate band data.) Upon return from the `CreateStroke()` function, the write location for the texture map is updated so it points to the next place that new data can be written. As with the vertex data, this allows for multiple paths to be stored in the same texture map.

The `geometryBuffer` parameter points to a `GeometryBuffer` structure containing the addresses of the storage into which vertex and triangle data is written. These addresses are typically in memory that is visible to the GPU. Upon return from the `CreateStroke()` function, the `GeometryBuffer` structure is updated so that the `vertexData` and `triangleData` fields point to the next element past the end of the data that was written. The `vertexIndex` field is advanced to one greater than the largest vertex index

written. This updated information allows for multiple filled and/or stroked paths to be built in the same vertex buffer and drawn with a single rendering command.

ExtendedGlyphData structure

The `ExtendedGlyphData` structure contains extended information about a specific glyph.

Fields

Field	Description
`Vector2D glyphOffset`	An em-space offset that is applied to the glyph. This is used when a glyph has contours identical to those of another glyph but at a different position.
`uint32 caretData`	The high 8 bits contain the number of caret positions for the glyph, and the low 24 bits contain the offset into the font's extended data table where the positions are stored.

Description

Some glyphs may have extended information stored in an `ExtendedGlyphData` structure. If a glyph has extended data, then the `extendedData` field of the `GlyphData` structure associated with the glyph is nonzero. The information in the `ExtendedGlyphData` structure is used internally.

ExtractBandTexture() function

The `ExtractBandTexture()` function decompresses the band texture stored in a `.slug` file.

Prototype

```
void ExtractBandTexture(const SlugFileHeader *fileHeader,
                        void *bandTexture);
```

Parameters

Parameter	Description
`fileHeader`	A pointer to the `SlugFileHeader` structure beginning at the first byte of the contents of a particular `.slug` file.
`bandTexture`	A pointer to the location where the decompressed band texture is stored.

Description

The `ExtractBandTexture()` function decompresses the band texture and stores it in memory at the location specified by the `bandTexture` parameter. This memory storage must be allocated by the caller. The size of the storage is obtained by calling the `GetBandTextureStorageSize()` function.

Once the curve and band texture data has been extracted, it can be passed to the rendering API to be used by the Slug shaders. The caller is responsible for releasing the texture storage when it is no longer needed.

ExtractCurveTexture() function

The `ExtractCurveTexture()` function decompresses the curve texture stored in a `.slug` file.

Prototype

```
void ExtractCurveTexture(const SlugFileHeader *fileHeader,
                         void *curveTexture);
```

Parameters

Parameter	Description
`fileHeader`	A pointer to the `SlugFileHeader` structure beginning at the first byte of the contents of a particular `.slug` file.
`curveTexture`	A pointer to the location where the decompressed curve texture is stored.

Description

The `ExtractCurveTexture()` function decompresses the curve texture and stores it in memory at the location specified by the `curveTexture` parameter. This memory storage must be allocated by the caller. The size of the storage is obtained by calling the `GetCurveTextureStorageSize()` function.

Once the curve and band texture data has been extracted, it can be passed to the rendering API to be used by the Slug shaders. The caller is responsible for releasing the texture storage when it is no longer needed.

ExtractFontTextures() function

The `ExtractFontTextures()` function decompresses the curve texture and band texture data for a font.

Prototype

```
void ExtractFontTextures(const FontHeader *fontHeader,
                         void *curveTexture,
                         void *bandTexture);
```

Parameters

Parameter	Description
`fontHeader`	A pointer to the `FontHeader` structure retrieved with the `GetFontHeader()` function for a particular `.slug` file.
`curveTexture`	A pointer to the location where the decompressed curve texture is stored. If this is `nullptr`, then the curve texture is not extracted.
`bandTexture`	A pointer to the location where the decompressed band texture is stored. If this is `nullptr`, then the band texture is not extracted.

Description

The `ExtractFontTextures()` function decompresses the texture data needed to render glyphs belonging to a specific font and stores them in memory at the locations specified by the `curveTexture` and `bandTexture` parameters. This memory storage must be allocated by the caller. The size of the storage is determined by using the dimensions of the textures given by the `FontHeader` structure. The `curveTextureSize` and `bandTextureSize` fields of the `FontHeader` structure give the texture widths and heights in texels. Each texel of the curve texture is 8 bytes in size, and each texel of the band texture is 4 bytes in size. The code in Listing 4.1 demonstrates how to calculate the storage sizes.

Once the texture data has been extracted, it can be passed to the rendering API to be used by the Slug shader. The caller is responsible for releasing the texture storage when it is no longer needed.

FillData structure

The `FillData` structure controls the options that determine the appearance of a filled path.

Fields

Field	Description
`Color4U fillColor`	The color of the fill.
`GradientType gradientType`	The type of gradient applied to the fill. See the description for information about possible values.
`Bivector3D gradientLine`	The scaled line corresponding to the direction and extent of a linear gradient, used when the `gradientType` field is `kGradientLinear`.
`Point3D gradientCircle`	The center and radius of the circle corresponding to a radial gradient, used when the `gradientType` field is `kGradientRadial`.
`Color4U gradientColor[2]`	The two colors used by the gradient. When the filled path is rendered, these are multiplied by the value of the `fillColor` field.

Description

The `FillData` structure controls the options that determine the appearance of a filled path. The fields of the `FillData` structure should be initialized to their default values by calling the `SetDefaultFillData()` function.

The following values are the gradient types that can be specified in the `gradientType` field.

Value	Description
`kGradientNone`	No gradient is applied.
`kGradientLinear`	The gradient is linear with respect to a scaled line.
`kGradientRadial`	The gradient is radial with respect to a center and radius.

The `gradientLine` and `gradientCircle` fields occupy the same storage, so values should be written to only one of these.

FillWorkspace structure

The `FillWorkspace` structure is used internally for temporary storage by the library functions that generate geometry and texture data for solid fills.

Description

The `FillWorkspace` structure serves as temporary storage space while data is being processed by the Slug library functions that generate geometry and texture data for solid fills. When the library is used in a single-threaded context, there is no need to allocate and specify `FillWorkspace` structures because the library can use its own internal storage. However, if these library functions are called from multiple threads, then the application must ensure that a different `FillWorkspace` structure is specified for each thread so that the library is safely reentrant.

`FillWorkspace` structures allocated by the application are passed to Slug library functions that need them as the last parameter. Due to the large size of the `FillWorkspace` structures, they should not be allocated on the stack, but only on the heap or as a static part of the program binary.

FontBoundingBoxData structure

The `FontBoundingBoxData` structure contains information about the bounding box limits for a font.

Fields

Field	Description
`Box2D baseBoundingBox`	The font-wide union of all em-space bounding boxes for base glyphs, which excludes combining marks.
`Box2D markBoundingBox`	The font-wide union of all em-space bounding boxes for glyphs that are combining marks. If a font does not contain any combining marks, then all coordinates for this box are zero.

Description

When the `GetFontKeyData()` function is called with a key parameter of `kFontKeyBoundingBox`, the return value is a pointer to a `FontBoundingBoxData` structure.

For these bounding boxes, whether a glyph is considered to be a combining mark depends only on whether the original font contained attachment data for the glyph and may not reflect whether the corresponding character has the `kCharacterCombiningMark` property.

FontDecorationData structure

The `FontDecorationData` structure contains information about the underline and strikethrough decorations for a font.

Fields

Field	Description
`float` `decorationSize`	The thickness of the decoration stroke, in em units.
`float` `decorationPosition`	The em-space *y* position of the bottom edge of the decoration stroke.
`uint16` `dataLocation[2]`	The coordinates in the font's band texture at which data for the decoration geometry begins.

Description

When the `GetFontKeyData()` function is called with a key parameter of `kFontKeyUnderline` or `kFontKeyStrikethrough`, the return value is a pointer to a `FontDecorationData` structure.

FontDesc structure

The `FontDesc` structure contains information about a single font referenced by a font map.

Fields

Field	Description
`const FontHeader *fontHeader`	A pointer to the `FontHeader` structure associated with the font, retrieved with the `GetFontHeader()` function.
`float fontScale`	A scale value applied to the font. This scales all glyphs by a fixed amount, effectively multiplying the overall font size. The default value is 1.0 applies no scale.
`float fontOffset`	An offset value applied to the font, in em units. This shifts all glyphs vertically by a fixed distance, where positive values offset upward. The size of the em is the product of the font size and the value of the `fontScale` field. The default value of 0.0 applies no offset.

Description

A `FontDesc` structure contains information about a single font referenced by a font map.

FontHeader structure

The `FontHeader` structure contains general information about a font.

Fields

Field	Description
`int32` `fontKeyDataCount`	The number of entries in the key data table.
`int32` `fontKeyDataOffset`	The offset to the key data table.
`int32` `pageCount`	The total number of 256-entry pages covered by the range of code points included in the font. Not every page must contain glyph mappings.
`int32` `pageIndexOffset`	The offset to the page index table. The page index table contains `pageCount` signed 16-bit entries. An entry of –1 in the table indicates that a page contains no glyph mappings.
`int32` `glyphIndexOffset`	The offset to the glyph index table. There are 256 entries for each page that contains glyph mappings, and each entry is a 32-bit integer.
`int32` `glyphCount`	The total number of unique glyphs in the font.
`int32` `glyphDataOffset[2]`	The offsets to the tables of `GlyphData` structures. The first offset is always valid, and it corresponds to the table of `GlyphData` structures for ordinary glyphs. The second offset is valid only when variants of each glyph are available for special effects such as outlining. If there are no variants, then the second offset is zero.
`int32` `contourDataOffset`	The offset to the contour data table. The location of the contour data for a particular glyph within this table is given by the `contourData` member of the `GraphicData` structure. This offset is zero if contour data is not available.

`int32` `decomposeDataOffset`	The offset to the decompose data table. The location of the decompose data for a particular glyph within this table is given by the `decomposeData` member of the `GlyphData` structure. This offset is zero if no glyphs in the font have decompose data.
`int32` `colorLayerDataOffset`	The offset to the color layer data table. The location of the color layer data for a particular glyph within this table is given by the `colorLayerData` member of the `GlyphData` structure. This offset is zero if color layer data is not available.
`int32` `baseAnchorDataOffset`	The offset to the combining base anchor data table. The location of the base anchor data for a particular glyph within this table is given by the `baseAnchorData` member of the `GlyphData` structure. This offset is zero if base anchor data is not available.
`int32` `markAttachDataOffset`	The offset to the combining mark attach data table. The location of the mark attach data for a particular glyph within this table is given by the `markAttachData` member of the `GlyphData` structure. This offset is zero if mark attach data is not available.
`int32` `kernDataOffset[2]`	The offset to the kerning data table. The first entry is for horizontal layout, and the second entry is for vertical layout. The location of the kerning data for a particular glyph within this table is given by the `kernData` member of the `GlyphData` structure. This offset is zero if kerning data is not available.
`int32` `sequenceDataOffset`	The offset to the sequence data table. The location of the sequence data for a particular glyph within this table is given by the `sequenceData` member of the `GlyphData` structure. This offset is zero if sequence data is not available.
`int32` `alternateDataOffset`	The offset to the alternate data table. The location of the alternate data for a particular glyph within this table is given by the `alternateData` member of the `GlyphData` structure. This offset is zero if alternate data is not available.
`int32` `caretPositionDataOffset`	The offset to the caret data table. The location of the caret data for a particular glyph within this table is given by the `caretData` member of the `GlyphData` structure. This offset is zero if no glyphs in the font have caret position data.

Description

The `FontHeader` structure contains information about the rendering characteristics and layout capabilities of a font. Most of the fields are used internally by the Slug library functions that accept a font header or an array of `FontDesc` structures. A pointer to a `FontHeader` structure can be obtained from the raw `.slug` file data by calling the `GetFontHeader()` function.

FontHeightData structure

The `FontHeightData` structure contains information about the cap height and ex height for a font.

Fields

Field	Description
`float` `fontCapHeight`	The cap height for the font, in em units. This represents the typical distance from the baseline to the tops of the capital roman letters.
`float` `fontExHeight`	The ex height for the font, in em units. This represents the typical distance from the baseline to the tops of lowercase roman letters, disregarding ascenders.

Description

When the `GetFontKeyData()` function is called with a key parameter of `kFontKeyHeight`, the return value is a pointer to a `FontHeightData` structure.

The values in the `FontHeightData` structure are taken from the `'OS/2'` table in the original font. In the unlikely case that the original font did not contain an `'OS/2'` table, the cap height is equal to the top of the bounding box for the uppercase letter H, and the ex height is equal to the top of the bounding box for the lowercase letter x.

FontMap structure

The `FontMap` structure contains information that maps each member of a set of font types to an array of source font indices.

Fields

Field	Description
`int32 fontTypeCount`	The number of font types. This must be at least 1.
`int32 fontSourceCount`	The maximum number of source fonts per font type. This must be at least 1.
`const uint32 *fontTypeArray`	A pointer to an array of application-defined font type codes. The number of elements in this array must be equal to the value of the `fontTypeCount` field.
`const uint8 *fontIndexTable`	A pointer to a two-dimensional array of font indices. The number of elements in this array must be equal to the product of the values of the `fontTypeCount` and `fontSourceCount` fields.

Description

A `FontMap` structure is passed to each of the library functions that handles multiple fonts for a single text string. The mechanism through which type codes are mapped to a set of source fonts is described in Section 4.6.

The `fontTypeCount` and `fontTypeArray` fields define the type codes that determine the current font style within a text string. The initial type code is specified by the `fontType` field of the `LayoutData` structure. The type code can be changed inside a string by the `font()` format directive. If the current type code is ever one that does not appear in the array supplied by the `FontMap` structure, then it is as if the first type code in the array was the current type code.

The `fontSourceCount` field specifies the maximum number of source fonts that can be searched for each font type. The product of the type count and the source count determines the number of entries in the table specified by the `fontIndexTable` field. For each type code, the table must contain an array of `fontSourceCount` consecutive entries in the font index table. The same array for the next type code immediately follows without any padding.

Each entry supplies the index of a font in the array of `FontDesc` structures passed to a library function. If a glyph for a particular Unicode character is not found in the first font referenced by the index table, then the search continues in the second font and so on until the number of sources is exhausted. If an index greater than or equal to the total number of fonts (as specified to a library function by the `fontCount` parameter) is encountered, then the search stops. This allows different numbers of sources

to be specified for different type codes. It is typical to place a value of 255 in the array to signal that there are no more entries, but this only has to be done if the number of entries is less than `fontSourceCount`. If no glyph corresponding to a particular Unicode character is found in any of the source fonts supplied for the current font type, then the missing glyph belonging to the first font referenced in the index array for the current font type is used.

FontMetricsData structure

The `FontMetricsData` structure contains information about the ascent, descent, and line gap for a font.

Fields

Field	Description
`float` `metricAscent`	The font designer's suggested ascent for the font, in em units. This is a positive distance above the baseline representing the vertical space occupied by the font.
`float` `metricDescent`	The font designer's suggested descent for the font, in em units. This is a negative distance below the baseline representing the vertical space occupied by the font.
`float` `metricLineGap`	The font designer's suggested line gap for the font, in em units. This is a positive distance added to the difference between the ascent and descent to calculate a suggested leading.

Description

When the `GetFontKeyData()` function is called with a key parameter of `kFontKeyMetrics` or `kFontKeyTypoMetrics`, the return value is a pointer to a `FontMetricsData` structure.

When the `kFontKeyMetrics` key is specified, the data in the `FontMetricsData` structure reflects the values in the `'hhea'` table in the original font.

When the `kFontKeyTypoMetrics` key is specified, the data in the `FontMetricsData` structure reflects the values in the `'OS/2'` table in the original font. In the unlikely case that the original font did not contain an `'OS/2'` table, the metrics data for the `kFontKeyTypoMetrics` key is identical to the metrics data for the `kFontKeyMetrics` key.

FontOutlineData structure

The `FontOutlineData` structure contains information about the outline effect for a font.

Fields

Field	Description
`float outlineEffectSize`	The size by which glyphs are expanded for the outline effect, in em units.
`float outlineMiterLimit`	The miter limit used for the outline effect where curves meet.
`OutlineJoinType outlineJoinType`	The join type used when the miter limit is exceeded in the outline effect.

Description

When the `GetFontKeyData()` function is called with a key parameter of `kFontKeyOutline`, the return value is a pointer to a `FontOutlineData` structure.

FontPolygonData structure

The `FontPolygonData` structure contains information about the glyph bounding polygons for a font.

Fields

Field	Description
`int32` `polygonVertexCount`	The maximum number of vertices that the bounding polygon for any glyph can have.
`float` `polygonEdgeFactor`	The cost multiplier applied to the interior edges of a polygon's triangulation.

Description

When the `GetFontKeyData()` function is called with a key parameter of `kFontKeyPolygon`, the return value is a pointer to a `FontPolygonData` structure.

FontScriptData structure

The `FontScriptData` structure contains information about the transformed-based subscripts or superscripts for a font.

Fields

Field	Description
`Vector2D scriptScale`	The scale for script glyphs.
`Vector2D scriptOffset`	The offset for script glyphs.

Description

When the `GetFontKeyData()` function is called with a key parameter of `kFontKeySubscript` or `kFontKeySuperscript`, the return value is a pointer to a `FontScriptData` structure.

GeometryBuffer structure

The `GeometryBuffer` structure contains pointers to the storage locations where vertices and triangles are written.

Fields

Field	Description
`volatile Vertex *vertexData`	A pointer to the location where vertex data is written.
`volatile Triangle16 *triangleData`	A pointer to the location where triangle data with 16-bit indices is written.
`volatile Triangle32 *triangleData32`	A pointer to the location where triangle data with 32-bit indices is written.
`uint32 vertexIndex`	The index of the first vertex written. The indices stored in the triangle data begin with this value. This should be set to zero for the first object (text, icon, fill, or stroke) in any group that will occupy the same geometry buffer.
`IndexType indexType`	The type of the vertex indices, either `kIndex16` or `kIndex32`.

Description

The `GeometryBuffer` structure contains information that tells several functions where to write the vertex and triangle data that they generate. The `vertexData` and `triangleData` (or `triangleData32`) fields typically point to GPU-visible memory that could be write-combined, so the pointers are declared `volatile` to ensure that write-only instructions are generated and that write order is preserved by the compiler for best performance.

The `triangleData` and `triangleData32` fields occupy the same storage location (as a union), and only one of them should be set. The `indexType` field indicates whether triangle indices are 16-bit or 32-bit unsigned integers. By default, `indexType` field is initialized to `kIndex16`.

When a build function returns, the fields of the `GeometryBuffer` structure have been updated to point to the end of the data that was written, and the same `GeometryBuffer` structure can be passed to additional function calls to append more data to the same vertex buffer.

GetAlbumHeader() function

The `GetAlbumHeader()` function returns a pointer to the `AlbumHeader` structure stored in a `.slug` file.

Prototype

```
const AlbumHeader *GetAlbumHeader(const SlugFileHeader *fileHeader);
```

Parameters

Parameter	Description
`fileHeader`	A pointer to the `SlugFileHeader` structure beginning at the first byte of the contents of a particular `.slug` file.

Description

After a `.slug` file is loaded into memory, a pointer to its contents should be cast to a pointer to a `SlugFileHeader` structure. If the file contains an album, then the `GetAlbumHeader()` function should be called to retrieve the `AlbumHeader` structure contained in the file. The returned pointer is then passed to other Slug functions that deal with albums.

Note that many data structures inside a `.slug` file are aligned to 64-byte boundaries to promote good cache performance. In particular, the `IconData` structures are 32-byte aligned and are exactly 96 bytes in size. To benefit from this, the contents of a `.slug` file must be loaded into a 64-byte aligned region of memory.

GetBandTextureStorageSize() function

The `GetBandTextureStorageSize()` function returns the decompressed size of the band texture stored in a `.slug` file.

Prototype

```
uint32 GetBandTextureStorageSize(const SlugFileHeader *fileHeader);
```

Parameters

Parameter	Description
`fileHeader`	A pointer to the `SlugFileHeader` structure beginning at the first byte of the contents of a particular `.slug` file.

Description

The `GetBandTextureStorageSize()` function returns the storage size, in bytes, needed to hold the decompressed curve texture stored in the file whose contents are specified by the `fileHeader` parameter. The curve texture data is decompressed by calling the `ExtractBandTexture()` function.

GetCompactCompiledStorageSize() function

The `GetCompactCompiledStorageSize()` function returns the minimum storage size needed to hold a compiled string of text.

Prototype

```
uint32 GetCompactCompiledStorageSize(const CompiledText *compiledText);
```

Parameters

Parameter	Description
`compiledText`	A pointer to a `CompiledText` structure containing a compiled string of text.

Description

The `GetCompactCompiledStorageSize()` function returns the minimum storage size, in bytes, needed to hold the compiled string of text stored in the object pointed to by the `compiledText` parameter. An application calls the `GetCompactCompiledStorageSize()` function to determine how many bytes would be written to memory by the `MakeCompactCompiledText()` function.

GetCurveTextureStorageSize() function

The `GetCurveTextureStorageSize()` function returns the decompressed size of the curve texture stored in a `.slug` file.

Prototype

```
uint32 GetCurveTextureStorageSize(const SlugFileHeader *fileHeader);
```

Parameters

Parameter	Description
`fileHeader`	A pointer to the `SlugFileHeader` structure beginning at the first byte of the contents of a particular `.slug` file.

Description

The `GetCurveTextureStorageSize()` function returns the storage size, in bytes, needed to hold the decompressed curve texture stored in the file whose contents are specified by the `fileHeader` parameter. The curve texture data is decompressed by calling the `ExtractCurveTexture()` function.

GetFontHeader() function

The `GetFontHeader()` function returns a pointer to a `FontHeader` structure stored in a `.slug` file.

Prototype

```
const FontHeader *GetFontHeader(const SlugFileHeader *fileHeader,
                                int32 index = 0);
```

Parameters

Parameter	Description
`fileHeader`	A pointer to the `SlugFileHeader` structure beginning at the first byte of the contents of a particular `.slug` file.
`index`	The index of the font within the file. This must be less than the count specified in the `resourceCount` field of the `SlugFileHeader` structure.

Description

After a `.slug` file is loaded into memory, a pointer to its contents should be cast to a pointer to a `SlugFileHeader` structure. If the file contains fonts, then the `GetFontHeader()` function should be called to retrieve the `FontHeader` structure for a specific font contained in the file. Most `.slug` files contain only a single font, so the `index` parameter can usually be omitted. The returned pointer to a `FontHeader` structure is then passed to other Slug functions that deal with fonts.

Note that many data structures inside a `.slug` file are aligned to 64-byte boundaries to promote good cache performance. In particular, the `GlyphData` structures are 64-byte aligned and are exactly 128 bytes in size. To benefit from this, the contents of a `.slug` file must be loaded into a 64-byte aligned region of memory.

GetFontKeyData() function

The `GetFontKeyData()` function returns a pointer to the data structure associated with a particular key value.

Prototype

```
const void *GetFontKeyData(const FontHeader *fontHeader,
                           FontKeyType key);
```

Parameters

Parameter	Description
`fontHeader`	A pointer to the `FontHeader` structure retrieved with the `GetFontHeader()` function for a particular `.slug` file.
`key`	The key value corresponding to the type of data to be returned.

Description

A font may contain various short chunks of data that each contain information about a particular aspect of the font. The `GetFontKeyData()` function finds the data associated with a given key value and returns a pointer to a corresponding data structure. The `key` parameter can be one of the values in the following table. For a particular key value, the `GetFontKeyData()` function returns a pointer to the data structure in the last column, which must be cast from the pointer to `void` that is returned.

The only types of data that are guaranteed to exist are the metrics and height data associated with the `kFontKeyMetrics`, `kFontKeyTypoMetrics`, and `kFontKeyHeight` key values. Data associated with other key values may not be present, and this includes the name of the font. When the requested type of data is not available, the return value is `nullptr`.

Key	Description	Data Structure
`kFontKeyName`	The name of the font.	Null-terminated string encoded in UTF-8.
`kFontKeySubname`	The subname of the font, normally referring to a typeface such as “Regular”, “Bold”, etc.	Null-terminated string encoded in UTF-8.
`kFontKeyMetrics`	Data for the ascent, descent, and line gap. These are the values specified in the `'hhea'` table in the original font.	`FontMetricsData`

`kFontKeyTypoMetrics`	Data for the ascent, descent, and line gap. These are the values specified in the `'OS/2'` table in the original font, if it existed. Otherwise, they are the same values returned for the `kFontKeyMetrics` key.	`FontMetricsData`
`kFontKeyHeight`	Data for the cap height and ex height.	`FontHeightData`
`kFontKeyBoundingBox`	Data for the bounding box limits.	`FontBoundingBoxData`
`kFontKeySubscript`	Data for the subscript transform.	`FontScriptData`
`kFontKeySuperscript`	Data for the superscript transform.	`FontScriptData`
`kFontKeyUnderline`	Data for the underline position and size.	`FontDecorationData`
`kFontKeyStrikethrough`	Data for the strikethrough position and size.	`FontDecorationData`
`kFontKeyPolygon`	Data for the glyph bounding polygons	`FontPolygonData`
`kFontKeyOutline`	Data for the outline effect.	`FontOutlineData`

GetFragmentShaderSourceCode() function

The `GetFragmentShaderSourceCode()` function returns the source code for a glyph fragment shader.

Prototype

```
int32 GetFragmentShaderSourceCode(uint32 fragmentIndex,
                                  const char **fragmentCode,
                                  uint32 shaderFlags = kFragmentShaderDefault);
```

Parameters

Parameter	Description
`fragmentIndex`	The index of the fragment shader returned by the `GetShaderIndices()` function.
`fragmentCode`	A pointer to the location that receives an array of pointers to the components of the fragment shader source code. The size of the array must be at least `kMaxFragmentStringCount`.
`shaderFlags`	Flags that determine what components of the shader are returned. See the description for information about the individual bits. The value `kFragmentShaderDefault` should be specified unless the shader is being incorporated into an external material system.

Description

The `GetFragmentShaderSourceCode()` function constructs an array of pointers to strings that make up the source code for a glyph fragment shader. The exact components stored in the array are determined by the rendering options corresponding to the value of the `fragmentIndex` parameter and the flags specified by the `shaderFlags` parameter. Pointers to one or more strings are stored in the array specified by the `fragmentCode` parameter. The return value is the number of strings that were stored in the array.

The following values can be combined (through logical OR) in the `shaderFlags` parameter.

Value	Description
`kFragmentShaderProlog`	A prolog component containing type definitions for compatibility across different shading languages is included in the returned string array.
`kFragmentShaderMain`	A component containing a `main()` function and declarations for interpolants and texture maps is included in the returned string array.

`kFragmentShaderDefault`	All components required for a standalone fragment shader are included in the returned string array.

The `shaderFlags` parameter is intended to be used to explicitly omit various shader components when the Slug shader is to be incorporated into an external material system. The value `kFragmentShaderDefault` should always be specified for standalone fragment shaders, and it has the effect of including all components.

If the `shaderFlags` parameter is not `kFragmentShaderDefault`, then an external material system must ensure that certain declarations have been made so that the returned shader code is valid. In particular, if the `kFragmentShaderProlog` flag is omitted, then the application must ensure that the type and function identifiers listed in the following table are available. (These identifiers have different names in GLSL.)

Identifier	Description
`int2`	A signed integer with two components.
`int4`	A signed integer with four components.
`uint4`	An unsigned integer with four components.
`float2`	A floating-point value with two components.
`float3`	A floating-point value with three components.
`float4`	A floating-point value with four components.
`lerp(x, y, t)`	A function that returns linear interpolation, `x * (1 - t) + y * t`.
`asint(x)`	A function that reinterprets the bits of *x* as a signed integer.
`asuint(x)`	A function that reinterprets the bits of *x* as an unsigned integer.
`asfloat(x)`	A function that reinterprets the bits of *x* as a floating-point value.
`saturate(x)`	A function that clamps *x* to the range [0,1].

In a GLSL shader under OpenGL ES, the application must also make the following declarations if the `kFragmentShaderProlog` flag is omitted. (These should not be declared for desktop OpenGL.)

```
precision highp float;
precision highp int;
precision highp sampler2D;
precision highp usampler2D;
```

If the `kFragmentShaderMain` flag is omitted, then the application is responsible for declaring the four interpolants output by the vertex shader as well as the two texture maps used by the fragment shader, each described in the following table. In this case, the `kVertexShaderMain` flag should generally be omitted when calling the `GetVertexShaderSourceCode()` function as well.

Resource	Description
`color`	The vertex color with four floating-point components. In HLSL and PSSL, this has the user-defined semantic `U_COLOR`.
`texcoord`	The vertex texture coordinates with two floating-point components. In HLSL and PSSL, this has the user-defined semantic `U_TEXCOORD`.
`banding`	The vertex banding data with four floating-point components. This must be declared as being flat shaded (no interpolation). In HLSL and PSSL, this has the user-defined semantic `U_BANDING`.
`glyph`	The vertex glyph data with four signed integer components. This must be declared as being flat shaded (no interpolation). In HLSL and PSSL, this has the user-defined semantic `U_GLYPH`.
`curveTexture`	The 2D texture map having four floating-point components containing curve data.
`bandTexture`	The 2D texture map having two unsigned integer components containing band data.

Furthermore, if the `kFragmentShaderMain` flag is omitted, then additional shader code supplied by the application must call the `SlugRender()` function with the parameters shown below. (In GLSL, the `curveData` and `bandData` parameters have the types `sampler2D` and `usampler2D`, respectively.) The return value is a linear RGBA color with the glyph coverage stored in the alpha component.

```
float4 SlugRender(Texture2D curveData,        // Curve texture map.
                  Texture2D<uint4> bandData,  // Band texture map.
                  float2 pixelPosition,       // Vertex texcoord.
                  float4 vertexColor,         // Vertex color.
                  float4 bandTransform,       // Vertex banding.
                  int4 glyphData);            // Vertex glyph.
```

GetGlyphContourCurveCount() function

The `GetGlyphContourCurveCount()` function returns the number of Bézier curves composing a specific glyph.

Prototype

```
int32 GetGlyphContourCurveCount(const FontHeader *fontHeader,
                                int32 glyphIndex);
```

Parameters

Parameter	Description
`fontHeader`	A pointer to the `FontHeader` structure retrieved with the `GetFontHeader()` function for a particular `.slug` file.
`glyphIndex`	The index of the glyph within the font.

Description

The `GetGlyphContourCurveCount()` function returns the number of quadratic Bézier curves composing the glyph with index specified by the `glyphIndex` parameter in the font specific by the `fontHeader` parameter. This function should be called to determine how much space needs to be allocated for the curve data returned by the `GetGlyphContourData()` function.

If a font does not contain contour data, then the `GetGlyphContourCurveCount()` function always returns zero. A font contains contour data only if it was converted to the Slug format with the `-contours` option specified on the command line of the `slugfont` tool.

The glyph index corresponding to a specific Unicode character value can be retrieved with the `GetGlyphIndex()` function.

GetGlyphContourData() function

The `GetGlyphContourData()` function returns the set of Bézier curves composing a specific glyph.

Prototype

```
void GetGlyphContourData(const FontHeader *fontHeader,
                         int32 glyphIndex,
                         const SlugFileHeader *slugFileHeader,
                         const Texel16 *curveTexture,
                         QuadraticBezier2D *contourCurve);
```

Parameters

Parameter	Description
`fontHeader`	A pointer to the `FontHeader` structure retrieved with the `GetFontHeader()` function for a particular `.slug` file.
`glyphIndex`	The index of the glyph within the font.
`slugFileHeader`	A pointer to the `SlugFileHeader` structure beginning at the first byte of the contents of a particular `.slug` file.
`curveTexture`	A pointer to the decompressed curve texture data previously extracted with the `ExtractCurveTexture()` function for the same value of the `slugFileHeader` parameter.
`contourCurve`	A pointer to an array to which the Bézier curves are written. The number of entries in this array must be at least the number returned by the `GetGlyphContourCurveCount()` function for the same glyph index.

Description

The `GetGlyphContourData()` function writes the set of quadratic Bézier curves composing the glyph with index specified by the `glyphIndex` parameter in the font specific by the `fontHeader` parameter to the array specified by the `contourCurve` parameter. The application is responsible for allocating the memory for the array, and it must be large enough to hold the number of Bézier curves returned by the `GetGlyphContourCurveCount()` function for the same glyph index in the same font.

The individual Bézier curves are read from the same curve texture data that is used for rendering. Information about this curve texture is stored in the `SlugFileHeader` structure, so it must be passed to the `GetGlyphContourData()` function in the `slugFileHeader` parameter. The data passed through the

`curveTexture` parameter must be the texture data previously extracted from the `.slug` file with the `ExtractCurveTexture()` function.

The glyph index corresponding to a specific Unicode character value can be retrieved with the `GetGlyphIndex()` function.

GetGlyphData() function

The `GetGlyphData()` function returns a pointer to the `GlyphData` structure for a specific character.

Prototype

```
const GlyphData *GetGlyphData(const FontHeader *fontHeader,
                              uint32 unicode);
```

Parameters

Parameter	Description
`fontHeader`	A pointer to the `FontHeader` structure retrieved with the `GetFontHeader()` function for a particular `.slug` file.
`unicode`	The Unicode character value.

Description

The `GetGlyphData()` function returns a pointer to the `GlyphData` structure corresponding to the Unicode character specified by the `unicode` parameter. If the font does not contain a glyph for that character, then the return value is a pointer to the `GlyphData` structure for the glyph used in place of a missing character, which is the same pointer returned when the `unicode` parameter is zero. The return value is never `nullptr`.

GetGlyphIndex() function

The `GetGlyphIndex()` function returns the glyph index for a specific character.

Prototype

```
int32 GetGlyphIndex(const FontHeader *fontHeader,
                    uint32 unicode);
```

Parameters

Parameter	Description
`fontHeader`	A pointer to the `FontHeader` structure retrieved with the `GetFontHeader()` function for a particular `.slug` file.
`unicode`	The Unicode character value.

Description

The `GetGlyphIndex()` function returns the glyph index corresponding to the Unicode character specified by the `unicode` parameter. If the font does not contain a glyph for that character, then the return value is zero, which is the valid index for the glyph used in place of a missing character.

GetIconData() function

The `GetIconData()` function returns a pointer to the `IconData` structure for a specific icon.

Prototype

```
const IconData *GetIconData(const AlbumHeader *albumHeader,
                            uint32 index);
```

Parameters

Parameter	Description
`albumHeader`	A pointer to the `AlbumHeader` structure retrieved with the `GetAlbumHeader()` function for a particular `.slug` file.
`index`	The index of the icon within the album. This index must be between 0 and one less than the count given by the `iconCount` field of the `AlbumHeader` structure, inclusive.

Description

The `GetIconData()` function returns a pointer to the `IconData` structure for the icon within an album corresponding to the `index` parameter. The return value is never `nullptr`.

GetKernValue() function

The `GetKernValue()` function determines the amount of kerning that should be applied between two glyphs.

Prototype

```
float GetKernValue(const FontHeader *fontHeader,
                   const GlyphData *glyphData,
                   int32 leftIndex);
```

Parameters

Parameter	Description
`fontHeader`	A pointer to the `FontHeader` structure retrieved with the `GetFontHeader()` function for a particular `.slug` file.
`glyphData`	A pointer to a `GlyphData` structure corresponding to the second glyph in the pair.
`leftIndex`	The index of the first glyph in the pair.

Description

The `GetKernValue()` function determines whether kerning should be applied between a pair of glyphs and, if so, returns the kern distance in em space. Negative kern values shift the second glyph in the pair to the left, and positive kern values shift the second glyph to the right. If no kerning should be applied to the pair of glyphs, then the return value is zero.

The `glyphData` parameter specifies the `GlyphData` structure corresponding to the *second* glyph in the pair and is typically obtained with the `GetGlyphData()` function. The `leftIndex` parameter specifies the index of the *first* glyph in the pair.

If no kerning data was imported from the original font, then the `GetKernValue()` function always returns zero.

GetShaderIndices() function

The `GetShaderIndices()` function returns vertex shader and fragment shader indices for drawing glyphs or icons with specific rendering options.

Prototype

```
void GetShaderIndices(uint32 renderFlags,
                      uint32 *vertexIndex,
                      uint32 *fragmentIndex);
```

Parameters

Parameter	Description
`renderFlags`	The `renderFlags` specifying the rendering options to be applied.
`vertexIndex`	A pointer to the location that receives the vertex shader index.
`fragmentIndex`	A pointer to the location that receives the fragment shader index.

Description

The `GetShaderIndices()` function returns the internal indices of the vertex shader and fragment shader used to draw glyphs or icons with specific rendering options. The value returned in `vertexIndex` is always less than `kMaxVertexShaderCount`, and the value returned in `fragmentIndex` is always less than `kMaxFragmentShaderCount`. These indices should be passed to the `GetVertexShaderSourceCode()` and `GetFragmentShaderSourceCode()` functions.

The following values can be combined (through logical OR) for the `renderFlags` parameter. For shaders that will be used to render glyphs, the `renderFlags` parameter should have the same value that is specified in `renderFlags` field of the `LayoutData` structure.

Value	Description
`kRenderOpticalWeight`	When rendering, coverage values are remapped to increase the optical weight of the glyphs. This can improve the appearance of small text, but usually looks good only for dark text on a light background.
`kRenderSupersampling`	The fragment shader performs adaptive supersampling for high-quality rendering at small font sizes.

`kRenderLinearCurves`	The fragment shader checks whether an icon or component of a picture is composed of straight lines and chooses a faster path for them when possible. This flag should not be specified for shaders that will be used to render glyphs.
`kRenderStrokes`	The fragment shader is capable of rendering strokes. This flag is required when strokes are rendered individually or as part of a picture. This flag should not be specified for shaders that will be used to render glyphs.
`kRenderGradients`	The fragment shader is capable of rendering gradients. If this flag is not set, then all gradient fills are rendered with solid colors instead. This flag should not be specified for shaders that will be used to render glyphs.
`kRenderMulticolor`	An icon is rendered with multiple color layers, if available. The output color is always premultiplied by the coverage. This flag applies only to icons and must not be specified when rendering glyphs or pictures.
`kRenderPremultiplyCoverage`	The color output by the fragment shader is premultiplied by the coverage. This flag should not be specified if the `kRenderMulticolor` flag is specified.
`kRenderPremultiplyInverseCoverage`	The color output by the fragment shader is premultiplied by the inverse of the coverage. This flag should not be specified if the `kRenderMulticolor` flag or `kRenderPremultiplyCoverage` flag is specified.

GetUnicodeCharacterFlags() function

The `GetUnicodeCharacterFlags()` function returns various property flags for a Unicode character.

Prototype

```
uint16 GetUnicodeCharacterFlags(uint32 unicode);
```

Parameters

Parameter	Description
`unicode`	The Unicode value of the character.

Description

The `GetUnicodeCharacterFlags()` function returns flags corresponding to various Unicode properties of the character specified by the `unicode` parameter. The flags can be a combination (through logical OR) of the following values.

Value	Description
`kCharacterInvisible`	The character is invisible and does not generate any glyphs.
`kCharacterCombiningMark`	The character is a combining mark.
`kCharacterLeftToRight`	The character is strongly left-to-right.
`kCharacterRightToLeft`	The character is strongly right-to-left.
`kCharacterBidiMirror`	The character should be mirrored in right-to-left text.
`kCharacterJoinNext`	The character can join cursively with the next character.
`kCharacterJoinPrevious`	The character can join cursively with the previous character.
`kCharacterJoinIgnore`	The character is ignored for the purposes of cursive joining.
`kCharacterVerticalUpright`	The character is displayed upright (not rotated) in vertical layout.

GetVertexShaderSourceCode() function

The `GetVertexShaderSourceCode()` function returns the source code for a glyph vertex shader.

Prototype

```
int32 GetVertexShaderSourceCode(uint32 vertexIndex,
                                const char **vertexCode,
                                uint32 shaderFlags = kVertexShaderDefault);
```

Parameters

Parameter	Description
`vertexIndex`	The index of the vertex shader returned by the `GetShaderIndices()` function.
`vertexCode`	A pointer to the location that receives an array of pointers to the components of the vertex shader source code. The size of the array must be at least `kMaxVertexStringCount`.
`shaderFlags`	Flags that determine what components of the shader are returned. See the description for information about the individual bits. The value `kVertexShaderDefault` should be specified unless the shader is being incorporated into an external material system.

Description

The `GetVertexShaderSourceCode()` function constructs an array of pointers to strings that make up the source code for a glyph vertex shader. The exact components stored in the array are determined by the rendering options corresponding to the value of the `vertexIndex` parameter and the flags specified by the `shaderFlags` parameter. Pointers to one or more strings are stored in the array specified by the `vertexCode` parameter. The return value is the number of strings that were stored in the array.

The following values can be combined (through logical OR) in the `shaderFlags` parameter.

Value	Description
`kVertexShaderProlog`	A prolog component containing type definitions for compatibility across different shading languages is included in the returned string array.
`kVertexShaderMain`	The component containing a `main()` function and declarations for attributes, uniform parameters, and interpolants is included in the returned string array.

kVertexShaderDefault	All components required for a standalone vertex shader are included in the returned string array.

The shaderFlags parameter is intended to be used to explicitly omit various shader components when the Slug shader is to be incorporated into an external material system. The value kVertexShader-Default should always be specified for standalone vertex shaders, and it has the effect of including all components.

If the shaderFlags parameter is not kVertexShaderDefault, then an external material system must ensure that certain declarations have been made so that the returned shader code is valid. In particular, if the kVertexShaderProlog flag is omitted, then the application must ensure that the type and function identifiers listed in the following table are available. (These identifiers have different names in GLSL.)

Identifier	Description
int4	A signed integer with four components.
uint2	An unsigned integer with two components.
float2	A floating-point value with two components.
float4	A floating-point value with four components.

In a GLSL shader under OpenGL ES, the application must also make the following declarations if the kVertexShaderProlog flag is omitted. (These should not be declared for desktop OpenGL.)

```
precision highp float;
precision highp int;
```

If the kVertexShaderMain flag is omitted, then the application is responsible for declaring the four output interpolants and matching them to the input interpolants in the fragment shader, each described in the following table. In this case, the kFragmentShaderMain flag should generally be omitted when calling the GetFragmentShaderSourceCode() function as well.

Resource	Description
color	The vertex color with four floating-point components. In HLSL and PSSL, this has the user-defined semantic U_COLOR.
texcoord	The vertex texture coordinates with two floating-point components. In HLSL and PSSL, this has the user-defined semantic U_TEXCOORD.

`banding`	The vertex banding data with four floating-point components. This must be declared as being flat shaded (no interpolation). In HLSL and PSSL, this has the user-defined semantic `U_BANDING`.
`glyph`	The vertex glyph data with four signed integer components. This must be declared as being flat shaded (no interpolation). In HLSL and PSSL, this has the user-defined semantic `U_GLYPH`.

Furthermore, if the `kVertexShaderMain` flag is omitted, then additional shader code supplied by the application must call the `SlugUnpack()` and `SlugDilate()` functions with the parameters shown below. The return value of the `SlugDilate()` function contains the output texture coordinates. The vertex shader must also pass the vertex color through to the fragment shader.

```
float4 SlugUnpack(float4 tex,               // Vertex texcoords.
                  float4 bnd,               // Vertex banding.
                  out float4 vbnd,          // Output banding data.
                  out int4 vgly);           // Output glyph data.

float2 SlugDilate(float4 pos,               // Vertex position.
                  float4 tex,               // Vertex texcoords.
                  float4 jac,               // Vertex jacobian.
                  float4 m0,                // MVP matrix, first row.
                  float4 m1,                // MVP matrix, second row.
                  float4 m3,                // MVP matrix, fourth row.
                  float2 dim,               // Viewport width and height.
                  out float2 vpos);         // Output vertex position.
```

The default vertex shader has two uniform inputs named `slug_matrix` and `slug_viewport`. The uniform `slug_matrix` consists of four 4D vectors representing the four rows of the 4×4 model-view-projection matrix. (This consists of 16 floating-point values stored in row-major order for a matrix that transforms column vectors.) The uniform `slug_viewport` consists of one 2D vector containing the width and height of the viewport, in pixels.

The vertex shader also has five varying inputs starting at attribute index 0, and these correspond to the fields of the `Vertex` structure, in declaration order.

GlyphData structure

The `GlyphData` structure contains information about a specific glyph.

Fields

Field	Description
`Vector2D glyphOffset`	An em-space offset that is applied to the glyph. This is nonzero when a glyph has contours identical to those of another glyph but at a different position. This offset is already applied to the bounding box and polygon vertices in the `GraphicData` base structure.
`float advanceWidth`	The horizontal advance width of the glyph, in em units.
`float advanceHeight`	The vertical advance height of the glyph, in em units. Zero for fonts without vertical metrics.
`float verticalOrigin`	The vertical origin of the glyph, in em units. Zero for fonts without vertical metrics.
`uint32 decomposeData`	The high 8 bits contain the number of glyphs into which the glyph decomposes, and the low 24 bits contain the offset into the font's decompose table at which the glyph indexes begin.
`uint32 colorLayerData`	The high 8 bits contain the number of color layers for the glyph, and the low 24 bits contain the offset into the font's layer data table at which color layers begin for the glyph.
`uint32 baseAnchorData`	The high 8 bits contain the number of anchor points to which combining marks can be attached to the glyph. The low 24 bits contain the offset into the font's base anchor data table at which anchor points begin for the glyph.
`uint32 markAttachData`	The high 8 bits contain the number of anchor points at which a combining mark can attach to another glyph. The low 24 bits contain the offset into the font's mark attach data table at which anchor points begin for the glyph. This is nonzero only for glyphs that are combining marks.
`uint32 kernData[2]`	The high 12 bits contain the number of kern pairs for which the glyph is the second of each pair, and the low 20 bits contain the offset into the font's kerning data table at which kern pairs begin for the glyph. The first entry is for horizontal layout, and the second entry is for vertical layout.

`uint32 sequenceData`	The high 12 bits contain the number of sequences for the glyph, and the low 20 bits contain the offset into the font's sequence data table at which sequences begin for the glyph.
`uint32 alternateData`	The high 8 bits contain the number of alternates for the glyph, and the low 24 bits contain the offset into the font's alternate data table at which alternates begin for the glyph.
`uint32 caretPositionData`	The high 8 bits contain the number of caret positions for the glyph, and the low 24 bits contain the offset into the font's caret position data table where the positions are stored.

Description

Each glyph in a font has an associated `GlyphData` structure that contains information about its rendering and layout properties. The `GlyphData` structure adds fields to the `GraphicData` base structure.

The `GlyphData` structure corresponding to a specific Unicode character in a specific font can be retrieved with the `GetGlyphData()` function.

GlyphRange structure

The `GlyphRange` structure contains information about a range of compiled glyphs.

Fields

Field	Description
`int32` `firstGlyph`	The number of the first glyph in the range. This must not be negative or greater than the total number of compiled glyphs.
`int32` `lastGlyph`	The number of the last glyph in the range. This can be set to `kMaxStringGlyphCount` to indicate that all glyphs up to the null terminator should be included. If this is less than the number of the first glyph, then the range is considered empty.
`float` `spaceJustify`	For fully justified text, the advance added to each character designated as a space, in absolute units. This field is used only if the `kLayoutFullJustification` bit is specified in the `layoutFlags` field of the `LayoutData` structure.

Description

The `GlyphRange` structure is generally used by the library to specify a contiguous subset of the glyphs that have been compiled into a `CompiledText` structure by the `CompileString()` function. Many library functions optionally accept a `GlyphRange` structure to limit processing to the range specified. If no `GlyphRange` structure is specified, then all of the compiled glyphs are processed.

The `firstGlyph` and `lastGlyph` fields specify the zero-based indexes of the first and last glyphs in the range. The `firstGlyph` field must be set to a value in the range $[0, n-1]$, where n is the total number of compiled glyphs, but the `lastGlyph` field can have any value. If the `lastGlyph` field is greater than or equal to the total number of compiled glyphs, then it indicates that processing should end with the last compiled glyph. Setting the `lastGlyph` field to `kMaxStringGlyphCount` ensures that glyphs are processed until the null terminator is reached.

If the value of the `lastGlyph` field is less than the value of the `firstGlyph` field, then the range is considered empty. Functions that accept a `GlyphRange` structure recognize this condition and generate outputs appropriate for a zero-length sequence of glyphs.

The `spaceJustify` field is used to perform full justification when the `kLayoutFullJustification` bit is specified in the `layoutFlags` field of the `LayoutData` structure. It contains the extra amount of advance width to apply to each space character in a line of text so that the glyphs perfectly fill the distance between the left and right margins. The value in the `spaceJustify` field is calculated by the library by functions that generate `LineData` structures, and it would normally be set to 0.0 by the application when the `GlyphRange` structure is used by itself.

GraphicData structure

The `GraphicData` structure contains information common to glyphs and icons.

Fields

Field	Description
`Box2D boundingBox`	The bounding box of the graphic in the coordinate system of the em square.
`uint16 bandLocation[2]`	The location in the band texture at which data for the graphic begins.
`int16 bandCount[2]`	The number of vertical and horizontal bands holding curve index data for the graphic. These are zero if and only if the graphic has no curves (e.g., for a space glyph).
`Vector2D bandScale`	The *x* and *y* scales by which em-space coordinates are multiplied to calculate band indices.
`uint16 contourCurveCount`	The total number of Bézier curves defining the contours of the graphic. If contour curve data is not present in a font, then this is zero, so this field should not be used to determine whether a glyph has curves.
`uint16 polygonCode`	The code for the graphic's bounding polygon. If this is zero, then the graphic is always rendered with its bounding box quad.
`uint32 contourData`	The offset into the contour data table at which contour data begins for the graphic.
`PolygonVertex polygonVertex[6];`	The vertex coordinates for the bounding polygon when the `polygonCode` field is nonzero.

Description

The `GlyphData` and `IconData` structures each have a `GraphicData` base structure that holds information common to both types of graphics.

IconData structure

The `IconData` structure contains information about a specific icon.

Fields

Field	Description
`uint16 iconFlags`	Flags indicating various properties of the icon. See the description for information about the individual bits.
`uint16 gradientCode`	A code corresponding to the type of gradient used by the icon. This is zero if no gradient is applied.
`uint16 colorDataLocation[2]`	The location in the band texture at which color data for the icon begins, if available.

Description

Each icon has an associated `IconData` structure that contains information about its rendering and layout properties. The `IconData` structure adds fields to the `GraphicData` base structure.

The following values can be combined (through logical OR) in the `iconFlags` field.

Value	Description
`kIconLinear`	The icon is composed entirely of straight lines. The `kRenderLinearCurves` flag may be specified for the `GetShaderIndices()` function to enable the shader optimization, but it is not required.
`kIconMulticolor`	The icon contains multiple color layers. The `kRenderMulticolor` flag should be specified for the `GetShaderIndices()` function to enable multicolor rendering.

ImportIconData() function

The `ImportIconData()` function generates the texture data and icon properties structure for a monochrome icon defined by a set of quadratic Bézier curves.

Prototype

```
bool ImportIconData(int32 curveCount,
                    const QuadraticBezier2D *curveArray,
                    TextureBuffer *curveTextureBuffer,
                    TextureBuffer *bandTextureBuffer,
                    IconData *iconData,
                    int32 maxVertexCount = 4,
                    float interiorEdgeFactor = 1.0F,
                    int32 maxBandCount = kMaxImportBandCount,
                    FillWorkspace *workspace = nullptr);
```

Parameters

Parameter	Description
`curveCount`	The number of quadratic Bézier curves making up the icon.
`curveArray`	A pointer to an array containing the quadratic Bézier curves making up the icon.
`curveTextureBuffer`	A pointer to a `TextureBuffer` structure describing the curve texture map.
`bandTextureBuffer`	A pointer to a `TextureBuffer` structure describing the band texture map.
`iconData`	A pointer to an `IconData` structure to which the properties of the icon are written.
`maxVertexCount`	The maximum number of vertices allowed in the icon's bounding polygon. The valid values for this parameter are 0, 4, 5, and 6.
`interiorEdgeFactor`	A factor that multiplies the cost of interior edges in the icon's bounding polygon. This is ignored if `maxVertexCount` is 0.
`maxBandCount`	The maximum number of bands that can be generated for the icon. This must be in the range [1, `kMaxImportBandCount`].

workspace	A pointer to an FillWorkspace structure that will be used for temporary storage. If this is nullptr, then an internal workspace shared among all callers is used. See the FillWorkspace structure for information about reentrancy.

Description

The ImportIconData() function imports an icon defined by the set of quadratic Bézier curves specified by the curveCount and curveArray parameters. The control points are stored in the curve texture map specified by the curveTextureBuffer parameter, and the band information is stored in the band texture map specified by the bandTextureBuffer parameter. The icon properties are stored in the structure specified by the iconData parameter.

The TextureBuffer structures specified by the curveTextureBuffer and bandTextureBuffer parameters should be initialized to refer to memory buffers allocated by the application for the curve and band textures. The textures must be large enough to hold the data generated for all of the icons being imported (through multiple calls to the ImportIconData() function), and it is usually the case that much more space than the anticipated requirements is allocated. The unused portion of each texture can be discarded after the icons are imported. The writeLocation field of each TextureBuffer structure should initially be set to (0, 0). After each successful icon import, this field will have been updated to the location of the first unused texel position, and another icon can be imported without changing its value. The final value of the writeLocation field indicates how much of each texture was actually used.

The ImportIconData() function returns true if there is enough space remaining in both the curve and band textures to store the icon. Otherwise, if not enough space is available, then the return value is false.

Each call to the ImportIconData() function must specify a different IconData structure for the iconData parameter. This structure is filled with information about the icon needed to render it, such as the location within the band texture where data for the icon begins. It is used to uniquely identify the icon for the BuildIcon() function.

The maxVertexCount parameter specifies the maximum number of vertices allowed in the bounding polygon calculated for the icon. This value is clamped to the range 4–6, and a bounding polygon is calculated having a minimum of three sides and the maximum number of sides given by the clamped value. Note that larger values of maxVertexCount require significantly more computation, so the default value of 4 should be used when the performance of the ImportIconData() function is more important than a potentially small increase in the icon's rendering performance. If the maxVertexCount parameter is zero, then it indicates that the icon's geometry is always to be built using a quad corresponding to its bounding box, regardless of the geometry type passed to the BuildIcon() function.

ImportMulticolorIconData() function

The `ImportIconData()` function generates the texture data and icon properties structure for a multicolor icon defined by a set of quadratic Bézier curves.

Prototype

```
bool ImportMulticolorIconData(int32 layerCount,
                              const Color4U *layerColor,
                              const int32 *curveCount,
                              const QuadraticBezier2D *const *curveArray,
                              TextureBuffer *curveTextureBuffer,
                              TextureBuffer *bandTextureBuffer,
                              IconData *iconData,
                              int32 maxVertexCount = 4,
                              float interiorEdgeFactor = 1.0F,
                              int32 maxBandCount = kMaxImportBandCount,
                              FillWorkspace *workspace = nullptr);
```

Parameters

Parameter	Description
`layerCount`	The number of color layers making up the icon.
`layerColor`	A pointer to an array specifying the color of each layer.
`curveCount`	A pointer to an array containing the number of quadratic Bézier curves making up each layer.
`curveArray`	A pointer to an array containing a pointer to the quadratic Bézier curves making up each layer.
`curveTextureBuffer`	A pointer to a `TextureBuffer` structure describing the curve texture map.
`bandTextureBuffer`	A pointer to a `TextureBuffer` structure describing the band texture map.
`iconData`	A pointer to an `IconData` structure to which the properties of the icon are written.
`maxVertexCount`	The maximum number of vertices allowed in the icon's bounding polygon. The valid values for this parameter are 0, 4, 5, and 6.
`interiorEdgeFactor`	A factor that multiplies the cost of interior edges in the icon's bounding polygon. This is ignored if `maxVertexCount` is 0.

`maxBandCount`	The maximum number of bands that can be generated for the icon. This must be in the range [1, `kMaxImportBandCount`].
`workspace`	A pointer to an `FillWorkspace` structure that will be used for temporary storage. If this is `nullptr`, then an internal workspace shared among all callers is used. See the `FillWorkspace` structure for information about reentrancy.

Description

The `ImportMulticolorIconData()` function is similar to the `ImportIconData()` function, except that it imports an icon defined by multiple layers having different colors.

The `layerCount` parameter specifies the number of layers in the icon. The `layerColor`, `curveCount`, and `curveArray` parameters each point to an array having the length given by `layerCount`. Each entry in the `curveCount` array specifies the number of curves in an array pointed to by the corresponding entry in the `curveArray` array. Layers should be arranged in back-to-front order.

Each entry in the `layerColor` array specifies the color of the corresponding layer in the sRGB color space with gamma correction applied. The alpha channel of each color is unused and should be set to 255, representing full opacity.

LayoutData structure

The `LayoutData` structure contains the state that controls the typesetting options for a line of text.

Fields

Field	Description
`uint32` `fontType`	The font type code used for layout with a font map. Set to 0 when building glyph data with a single font.
`float` `fontSize`	The font size, in absolute units. This corresponds to the size of the em square.
`float` `fontStretch`	A stretch factor applied to the font only in the horizontal direction. This is a scale applied to each glyph as well as the advance width, tracking, and kerning distances. Set to 1.0 for no stretch.
`float` `textTracking`	The extra horizontal space added between all pairs of base glyphs, in em units. This can be positive or negative. Set to 0.0 for no tracking.
`float` `textSkew`	A skew value representing the ratio of the change in *x* coordinate to the change in *y* coordinate measured upward from the baseline. Positive values cause the text to slant to the right, and negative values cause the text to slant to the left. Set to 0.0 for no skew.
`Vector2D` `textScale`	An *x* and *y* scale adjustment applied to each glyph. The font size and stretch are multiplied by this scale to obtain the final em size of each glyph. Text decorations are not affected by the scale. These scales must be positive values. Set to (1.0, 1.0) for no scale.
`Vector2D` `textOffset`	An *x* and *y* offset adjustment applied to each glyph. Positive values offset right and upward, and negative values offset left and downward. The offset is applied after the scale. Text decorations are not affected by the offset. Set to (0.0, 0.0) for no offset.
`ColorData` `textColor`	The solid color or gradient applied to the text. See the description of the `ColorData` structure.
`AlignmentType` `textAlignment`	The alignment applied to multi-line text. See the description for information about possible values. If the `kLayoutFullJustification` flag is specified in the `layoutFlags` field, then the alignment applies to the last line of each paragraph.
`float` `textLeading`	The amount of leading applied to multi-line text, in em units. This is the distance between the baselines of consecutive lines of text.

`float` `paragraphSpacing`	Additional spacing added between paragraphs, in em units. This field is used only if the `kLayoutParagraphAttributes` bit is specified in the `layoutFlags` field.
`float` `leftMargin`	The paragraph left margin, in absolute units. This field is used only if the `kLayoutParagraphAttributes` bit is specified in the `layoutFlags` field.
`float` `rightMargin`	The paragraph right margin, in absolute units. This field is used only if the `kLayoutParagraphAttributes` bit is specified in the `layoutFlags` field.
`float` `firstLineIndent`	The indent distance for the first line of each paragraph, in absolute units. This is added to the left or right margin, and it can be positive or negative. This field is used only if the `kLayoutParagraphAttributes` bit is specified in the `layoutFlags` field.
`float` `tabSize`	The distance between consecutive tab stops, in absolute units. The tab size must be greater than zero. This field is used only if the `kLayoutTabSpacing` bit is specified in the `layoutFlags` field.
`float` `tabRound`	The tab stop rounding distance, in em units. This is added to the current drawing position when determining the next tab stop.
`uint32` `layoutFlags`	Various flags that specify layout options. See the description for information about the individual bits. A value of zero causes kerning, sequence replacement, combining marks, and multicolor glyphs to be enabled, and it causes format directives and clipping planes to be disabled.
`uint32` `renderFlags`	Various flags that specify rendering options. See the description for information about the individual bits. A value of zero causes kerning, combining marks, and sequence replacement to be enabled, and it causes format directives, clipping planes, grid mode, paragraph attributes, tab spacing, and full justification to be disabled.
`GeometryType` `geometryType`	The type of geometry used to render each glyph. See the description for information about the various types.
`uint32` `formatMask`	A mask that determines which embedded format directives can be applied. A one bit indicates that the corresponding format directive is enabled. See the description for information about the individual bits. This field is used only if the `kLayoutFormatDirectives` bit is specified in the `layoutFlags` field. A value of `~0` enables all format directives.

`uint32 sequenceMask`	A mask that determines which types of sequence replacements are applied. A one bit indicates that the corresponding type of sequence replacement is enabled. See the description for information about the individual bits. This field is used only if the `kLayoutSequenceDisable` is not specified in the `layoutFlags` field.
`uint32 alternateMask`	A mask that determines which types of alternate substitution features are applied. A one bit indicates that the corresponding type of alternate substitution is enabled. See the description for information about the individual bits. This field is used only if the `kLayoutAlternateDisable` is not specified in the `layoutFlags` field.
`uint32 styleIndex`	The style index used when the `kAlternateStylistic` bit is set in the `alternateTypeMask` field. This value should be in the range 1–20, and it corresponds to the style set defined by the original font.
`int32 scriptLevel`	The transform-based superscript or subscript level applied to the glyphs. Zero means no script transform, positive values correspond to superscripts, and negative values correspond to subscripts.
`bool decorationFlag [kDecorationCount]`	An array of flags that specify which decorations to apply to the text. See the description for information about the individual array indexes.
`int32 spaceCount`	The number of space characters specified by the `spaceArray` field. This field is used only if the `kLayoutFullJustification` bit is specified in the `layoutFlags` field.
`const uint32 *spaceArray`	A pointer to an array of space characters with `spaceCount` entries. The values in this array are Unicode characters, and they must be sorted in ascending order. This field is used only if the `kLayoutFullJustification` bit is specified in the `layoutFlags` field.
`uint32 placeholderBase`	The base Unicode value for an application-defined range of special characters that indicate placeholders for external graphics. This field must not be zero if placeholders are enabled. A character with the Unicode value specified by this field corresponds to the placeholder with index zero.
`int32 placeholderCount`	The number of Unicode values in the range of special characters that indicate placeholders. The maximum allowed value is given by the `kMaxStringPlaceholderCount` constant, which is 65536. Placeholder functionality is enabled when this field is not zero.

`const float *placeholderWidthArray`	A pointer to an array containing the widths of all placeholders, in absolute units. If the `placeholderCount` field is not zero, then this field must point to an array having `placeholderCount` entries.
`GlyphEffectType effectType`	The type of the glyph effect to apply. Set to `kGlyphEffectNone` for no glyph effect.
`Vector2D effectOffset`	The *x* and *y* offsets at which the glyph effect is rendered, in em units. Positive values offset right and downward. This field is used only if the `effectType` field is not `kGlyphEffectNone`.
`ColorData effectColor`	The solid color or gradient applied to the glyph effect. See the description of the `ColorData` structure. This field is used only if the `effectType` field is not `kGlyphEffectNone`.
`Vector2D objectScale`	The *x* and *y* scales applied to the final vertex positions for each glyph.
`Vector2D objectOffset`	The *x* and *y* offsets applied to the final vertex positions for each glyph after the scales are applied.
`float clipLeft`	The *x* coordinate at which glyphs are clipped on the left. This field is used only if the `kLayoutClippingPlanes` bit is specified in the `layoutFlags` field.
`float clipRight`	The *x* coordinate at which glyphs are clipped on the right. This field is used only if the `kLayoutClippingPlanes` bit is specified in the `layoutFlags` field.
`int32 missingGlyphIndex`	The index of the glyph to be drawn when a character is missing from a font.
`const LayoutData *defaultLayoutData`	An optional pointer to another `LayoutData` structure containing default values.

Description

The `LayoutData` structure controls everything about the typesetting options that ultimately determine the appearance of a line of text. The fields of the `LayoutData` structure can be initialized to their default values by calling the `SetDefaultLayoutData()` function.

Several important functions in the Slug library take a pointer to a `LayoutData` structure on entry, and most of those functions can return an updated `LayoutData` structure on exit. The layout data can change because many of the fields belonging to this structure can be dynamically modified by format directives embedded in a text string if the `kLayoutFormatDirectives` bit is set in the `layoutFlags` field. (See Chapter 6 for more information about using format directives.)

The `renderFlags` field may contain a combination (through logical OR) of the flag values listed for the `renderFlags` parameter passed to the `GetShaderIndices()` function that specifically apply to glyphs.

The following values are the alignment types that can be specified in the `textAlignment` field.

Value	Description
`kAlignmentLeft`	Text is aligned with the rendering position on the left end of each line.
`kAlignmentRight`	Text is aligned with the rendering position on the right end of each line.
`kAlignmentCenter`	Text is aligned with the rendering position at the center of each line.

The following values can be combined (through logical OR) in the `layoutFlags` field.

Value	Description
`kLayoutFormatDirectives`	Format directives embedded in the text are recognized and applied.
`kLayoutClippingPlanes`	Each glyph is clipped against left and right boundaries specified by the `clipLeft` and `clipRight` fields.
`kLayoutKernDisable`	Kerning is not applied to the text. When this flag is not set, kerning is applied according to the tables specified in the original font.
`kLayoutMarkDisable`	Combining marks are not repositioned in the text. When this flag is not set, each combining mark is moved to the appropriate attachment point belonging to the preceding glyph.
`kLayoutDecomposeDisable`	Glyphs are not decomposed in the text. When this flag is not set, specific glyphs can be decomposed into multiple glyphs as specified in the original font.
`kLayoutSequenceDisable`	Sequences are not matched in the text. When this flag is not set, tables specified in the original font can cause certain sequences of glyphs to be replaced by one or more other glyphs.
`kLayoutAlternateDisable`	Alternate glyph substitution is not applied in the text. When this flag is not set, various substitution features can cause glyphs to be replaced by alternate forms.
`kLayoutLayerDisable`	Only monochrome glyphs are rendered, and color layers in emoji are ignored.

`kLayoutLayerTextColor`	The color of each layer in an emoji is multiplied by the current text color.
`kLayoutNonlinearColor`	Colors are not linearized but instead passed through in gamma space.
`kLayoutFullJustification`	Text is rendered with full justification.
`kLayoutRightToLeft`	The forward writing direction is right-to-left.
`kLayoutBidirectional`	Bidirectional text layout is enabled.
`kLayoutVerticalRotation`	Vertical text layout is enabled. When this flag is set, glyphs corresponding to upright characters use vertical metrics, can be replaced by alternate vertical forms, and are rotated 90 degrees counterclockwise.
`kLayoutGridPositioning`	Grid positioning mode is enabled. When this flag is set, glyph advance widths and kerning are ignored, and each glyph is centered on the current drawing position. The drawing position is advanced from one glyph to the next only by the tracking value.
`kLayoutParagraphAttributes`	Paragraph spacing, margins, and first-line indent are enabled.
`kLayoutTabSpacing`	Tab spacing is enabled.
`kLayoutSoftHyphen`	Breaking lines at soft hyphens is enabled.
`kLayoutWrapDisable`	Lines are broken only at hard break characters and not when the maximum span is exceeded.

The following values are the glyph geometry types that can be specified in the `geometryType` field.

Value	Description
`kGeometryQuads`	Each glyph is rendered with a single quad composed of four vertices and two triangles.
`kGeometryPolygons`	Each glyph is rendered with a tight bounding polygon having between 3 and 6 vertices. If polygon data is not available in the font, then each glyph is rendered as a quad.
`kGeometryRectangles`	Each glyph is rendered with exactly one triangle with the expectation that the window-aligned bounding rectangle will be filled. In this case, no data is written to the `triangleData` array specified in the `GeometryBuffer` structure, so it can be `nullptr`. This type can be used only when rectangle primitives are available, such as provided by the `VK_NV_fill_rectangle` and `GL_NV_fill_rectangle` extensions.

The following values can be combined (through logical OR) in the `formatMask` field. The format mask is used only when the `kLayoutFormatDirectives` bit is set in the `layoutFlags` field.

Value	Description
`kFormatFont`	Font type directive `font()`.
`kFormatSize`	Font size directive `size()`.
`kFormatStretch`	Stretch directive `stretch()`.
`kFormatTracking`	Tracking directive `track()`.
`kFormatSkew`	Skew directive `skew()`.
`kFormatScale`	Text scale directive `scale()`.
`kFormatOffset`	Text offset directive `offset()`.
`kFormatColor`	Color directives `color()` and `color2()`.
`kFormatGradient`	Gradient directives `grad()` and `gcoord()`.
`kFormatAlignment`	Alignment directives `left()`, `right()`, `center()`, and `just()`.
`kFormatLeading`	Leading directive `lead()`.
`kFormatParagraph`	Paragraph directives `pspace()`, `margin()`, and `indent()`.

`kFormatTab`	Tab size directive `tab()`.
`kFormatKern`	Kerning directive `kern()`.
`kFormatMark`	Mark placement directive `mark()`.
`kFormatDecompose`	Glyph decompose directive `decomp()`.
`kFormatSequence`	Sequence directive `seq()` and all specific sequence replacement directives.
`kFormatAlternate`	Alternate directive `alt()` and all specific alternate substitution directives.
`kFormatLayer`	Color layer directive `lay()`.
`kFormatDecoration`	Underline directive `under()` and strikethrough directive `strike()`.
`kFormatScript`	Transform-based subscript and superscript directive `script()`.
`kFormatGrid`	Grid positioning directive `grid()`.

The following values can be combined (through logical OR) in the `sequenceMask` field. The sequence mask is ignored if the `kLayoutSequenceDisable` bit is set in the `layoutFlags` field.

Value	Description
`kSequenceGlyphComposition`	Glyph compositions expected by a font to be applied in all circumstances.
`kSequenceStandardLigatures`	Standard ligatures provided by a font as substitute glyphs for letter groupings such as “fi” or “ffl”. (Some fonts may specify these as discretionary.)
`kSequenceRequiredLigatures`	Required ligatures considered by a font to be mandatory in some writing systems for various letter groupings.
`kSequenceDiscretionaryLigatures`	Discretionary ligatures provided by a font as additional glyphs that are considered optional.
`kSequenceHistoricalLigatures`	Historical ligatures provided by a font as optional forms.
`kSequenceAlternativeFractions`	Alternative fractions provided by a font for specific numerators and denominators separated by a slash.

The following values can be combined (through logical OR) in the `alternateMask` field. The alternate mask is ignored if the `kLayoutAlternateDisable` bit is set in the `layoutFlags` field.

Value	Description
`kAlternateStylistic`	Replace glyphs with alternates from a stylistic set. The `styleIndex` field determines which set is used.
`kAlternateHistorical`	Replace glyphs with historical alternates.
`kAlternateLowerSmallCaps`	Replace lowercase characters with small caps variants.
`kAlternateUpperSmallCaps`	Replace uppercase characters with small caps variants.
`kAlternateTitlingCaps`	Replace glyphs with titling caps variants.
`kAlternateUnicase`	Replace both cases with forms having equal heights.
`kAlternateCaseForms`	Replace case-sensitive punctuation with uppercase forms.
`kAlternateSlashedZero`	Replace the number zero with a slashed variant.
`kAlternateHyphenMinus`	Replace the hyphen (U+002D) with a minus sign (U+2212).
`kAlternateFractions`	Replace figures separated by a slash with numerators, the fraction slash, and denominators.
`kAlternateLiningFigures`	Replace old-style figures with lining figures.
`kAlternateOldstyleFigures`	Replace lining figures with old-style figures.
`kAlternateTabularFigures`	Replace proportional figures with tabular figures.
`kAlternateProportionalFigures`	Replace tabular figures with proportional figures.
`kAlternateSubscript`	Replace glyphs with subscript variants.
`kAlternateSuperscript`	Replace glyphs with superscript variants.
`kAlternateInferiors`	Replace glyphs with subscripts intended for scientific formulas.
`kAlternateOrdinals`	Replace glyphs with superscripts intended for ordinal numbers.

The following values are the glyph effect types that can be specified in the `effectType` field.

Value	Description
`kGlyphEffectNone`	No glyph effect is applied.
`kGlyphEffectShadow`	A drop shadow is rendered beneath each glyph.
`kGlyphEffectOutline`	An expanded outline is rendered beneath each glyph. The font must contain outline glyph data for this effect to appear.

LayoutMultiLineText() function

The `LayoutMultiLineText()` function generates the glyph indices and drawing positions for multiple lines of text.

Prototype

```
int32 LayoutMultiLineText(const CompiledText *compiledText,
                          const FontHeader *fontHeader,
                          int32 lineIndex,
                          int32 lineCount,
                          const LineData *lineDataArray,
                          const Point2D& position,
                          float maxSpan,
                          int32 *glyphIndexBuffer,
                          Point2D *positionBuffer,
                          Matrix2D *matrixBuffer,
                          Color4U *colorBuffer,
                          PlaceholderBuffer *placeholderBuffer = nullptr);
```

Parameters

Parameter	Description
`compiledText`	A pointer to a `CompiledText` object returned by a preceding call to the `CompileString()` function.
`fontHeader`	A pointer to the `FontHeader` structure retrieved with the `GetFontHeader()` function for a particular `.slug` file.
`lineIndex`	The zero-based index of the first line of text to build.
`lineCount`	The number of lines of text to build.
`lineDataArray`	A pointer to an array of `LineData` structures containing information about each line of text. The lines of text to be built correspond to elements indexed `lineIndex` through `lineIndex + lineCount - 1` in this array.
`position`	The *x* and *y* coordinates of the first glyph at the baseline of the first line of text.
`maxSpan`	The maximum physical horizontal span of the text.

glyphIndexBuffer	A pointer to an array of integers that receives glyph indices. This array must be at least as large as the glyph count returned by the CountMultiLineText() function.
positionBuffer	A pointer to an array of Point2D structures that receives glyph indices. This array must be at least as large as the glyph count returned by the CountMultiLineText() function.
matrixBuffer	A pointer to an array of Matrix2D structures that receives glyph transforms. This parameter can be nullptr, in which case no per-glyph transforms are returned. Otherwise, this array must be at least as large as the glyph count returned by the CountMultiLineText() function.
colorBuffer	A pointer to an array of Color4U structures that receives glyph colors. This parameter can be nullptr, in which case no per-glyph colors are returned. Otherwise, this array must be at least as large as the glyph count returned by the CountMultiLineText() function.
placeholderBuffer	A pointer to a PlaceholderBuffer structure containing information about where the output placeholder data is stored. This parameter can be nullptr, in which case no placeholder data is generated.

Description

The LayoutMultiLineText() function generates the glyph indices, drawing positions, transformations, and colors for multiple lines of text. The text is processed in the same way that it is for the BuildMultiLineText() function. After possible modification by the application, the information returned by the LayoutMultiLineText() function can be passed to the AssembleSlug() function to generate vertex and triangle data.

The compiledText parameter should be a pointer to a CompiledText structure returned by a preceding call to the CompileString() function. The pointer passed to the fontHeader parameter must be the same that was passed to the fontHeader parameter of the CompileString() function.

Before the LayoutMultiLineText() function can be called, the BreakMultiLineText() and CountMultiLineText() functions must be called for the same compiled string to determine the locations where lines break and the maximum amount of storage that the LayoutMultiLineText() function will need to write its data. The compiled string must be exactly the same for all three functions to ensure that the correct amount of storage can be allocated and that the data generated by the LayoutMultiLineText() function stays within the calculated limits.

The lineIndex parameter specifies the zero-based index of the first line of text to build, and the lineCount parameter specifies the number of lines to build. The lineDataArray parameter must point to an array of LineData structures containing at least lineIndex + lineCount elements. These would normally have been generated by a previous call to the BreakMultiLineText() function.

The `position` parameter specifies the *x* and *y* coordinates of the left side of the first glyph at the baseline of the first line of text. This is often (0, 0) when the transformation matrix applied externally by the application includes an object-space position.

The `maxSpan` parameter specifies the maximum physical horizontal span for all lines of text, and it's value should match the value previously passed to the `BreakMultiLineText()` function to generate the array of `LineData` structures. If the text alignment is `kAlignmentRight` or `kAlignmentCenter`, as specified by the `textAlignment` field of the `LayoutData` structure, then the `maxSpan` parameter is used to determine the proper horizontal position at which each line of text is rendered. If embedded format directives are enabled, then the alignment can be changed within a line of text, but the new alignment does not take effect until the next line is started.

The `glyphIndexBuffer` parameter points to an array into which glyph indices are written, and the `positionBuffer` parameter points to an array into which glyph positions are written. Each glyph position accounts for tracking, kerning, the *x* and *y* text offsets, and combining mark attachments. Glyph indices and positions are always returned, and these parameters cannot be `nullptr`.

The `matrixBuffer` parameter optionally points to an array into which glyph transforms are written. Each glyph transform accounts for the font size, font stretch, *x* and *y* text scales, and the text skew that would be applied by the `BuildMultiLineText()` function. The `matrixBuffer` parameter can be `nullptr` if this information is not needed.

The `colorBuffer` parameter optionally points to an array into which glyph colors are written. The `colorBuffer` parameter can be `nullptr` if this information is not needed.

The number of entries written to each array is always the same as the number of glyphs returned by the `CountMultiLineText()` function. The data written to the output buffers includes glyphs that do not have any geometry, such as the glyph corresponding to the space character. Glyph effects and text decorations are ignored and have do not affect the number of glyphs.

If placeholders are being used, the `placeholderBuffer` parameter points to a `PlaceholderBuffer` structure containing the address of the storage into which placeholder information is written. Upon return from the `LayoutMultiLineText()` function, the `PlaceholderBuffer` structure is updated so that the `placeholderData` field points to the next element past the data that was written. The actual number of placeholders generated by the `LayoutMultiLineText()` function should be determined by examining the pointer in the `PlaceholderBuffer` structure upon return and subtracting the original value of that pointer. The resulting difference can be less than the maximum value returned by the `CountMultiLine-Text()` function.

Any characters in the original text string designated as control characters by the Unicode standard do not generate any output. These characters never contribute any spacing in the slug layout, even if the original font defines nonzero advance widths for them.

When a new line is started, it is placed at a distance below the previous line given by the product of the font size and leading, as specified by the `fontSize` and `textLeading` fields of the `LayoutData` structure. If paragraph attributes are enabled and the new line is the first line in a new paragraph, then the leading is increased by the `paragraphSpacing` field of the `LayoutData` structure. If embedded format directives are enabled, the leading and paragraph spacing values can be changed within a line of text, but the new line spacing takes effect when the next line or paragraph is started.

LayoutMultiLineTextEx() function

The `LayoutMultiLineTextEx()` function generates the glyph indices and drawing positions for multiple lines of text.

Prototype

```
int32 LayoutMultiLineTextEx(const CompiledText *compiledText,
                            int32 fontCount,
                            const FontDesc *fontDesc,
                            int32 lineIndex,
                            int32 lineCount,
                            const LineData *lineDataArray,
                            const Point2D& position,
                            float maxSpan,
                            uint8 *fontIndexBuffer,
                            int32 *glyphIndexBuffer,
                            Point2D *positionBuffer,
                            Matrix2D *matrixBuffer,
                            Color4U *colorBuffer,
                            PlaceholderBuffer *placeholderBuffer = nullptr);
```

Parameters

Parameter	Description
`compiledText`	A pointer to a `CompiledText` object returned by a preceding call to the `CompileStringEx()` function.
`fontCount`	The total number of fonts that may be utilized. This must be at least 1.
`fontDesc`	A pointer to an array of `FontDesc` structures describing the fonts that may be utilized. The number of elements in this array must be equal to the value of the `fontCount` parameter.
`lineIndex`	The zero-based index of the first line of text to build.
`lineCount`	The number of lines of text to build.
`lineDataArray`	A pointer to an array of `LineData` structures containing information about each line of text. The lines of text to be built correspond to elements indexed `lineIndex` through `lineIndex + lineCount - 1` in this array.
`position`	The *x* and *y* coordinates of the first glyph at the baseline of the first line of text.

`maxSpan`	The maximum physical horizontal span of the text.
`fontIndexBuffer`	A pointer to an array of integers that receives font indices. This parameter can be `nullptr`, in which case no per-glyph font indices are returned. Otherwise, this array must be at least as large as the glyph count returned by the `CountMultiLineTextEx()` function.
`glyphIndexBuffer`	A pointer to an array of integers that receives glyph indices. This array must be at least as large as the glyph count returned by the `CountMultiLineTextEx()` function.
`positionBuffer`	A pointer to an array of `Point2D` structures that receives glyph indices. This array must be at least as large as the glyph count returned by the `CountMultiLineTextEx()` function.
`matrixBuffer`	A pointer to an array of `Matrix2D` structures that receives glyph transforms. This parameter can be `nullptr`, in which case no per-glyph transforms are returned. Otherwise, this array must be at least as large as the glyph count returned by the `CountMultiLineTextEx()` function.
`colorBuffer`	A pointer to an array of `Color4U` structures that receives glyph colors. This parameter can be `nullptr`, in which case no per-glyph colors are returned. Otherwise, this array must be at least as large as the glyph count returned by the `CountMultiLineTextEx()` function.
`placeholderBuffer`	A pointer to a `PlaceholderBuffer` structure containing information about where the output placeholder data is stored. This parameter can be `nullptr`, in which case no placeholder data is generated.

Description

The `LayoutMultiLineTextEx()` function is an extended version of the `LayoutMultiLineText()` function capable of handling multiple fonts through the mapping mechanism described in Section 4.6. A call to the `LayoutMultiLineText()` function is internally forwarded to the `LayoutMultiLineTextEx()` function with the `fontCount` parameter set to 1, the `fontDesc` parameter set to the address of a single `FontDesc` structure containing the font header with default scale and offset, the `fontMap` parameter set to `nullptr`, and the `fontIndexBuffer` parameter set to `nullptr`.

The `compiledText` parameter should be a pointer to a `CompiledText` structure returned by a preceding call to the `CompileStringEx()` function. The value of the `fontCount` parameter and the entries of the array specified by the `fontDesc` parameter must be exactly the same values that were passed to the `fontCount` and `fontDesc` parameters of the `CompileStringEx()` function.

After possible modification by the application, the information returned by the `LayoutMultiLineTextEx()` function can be passed to the `AssembleSlugEx()` function to generate vertex and triangle data.

After the first three parameters, the remaining parameters passed to the `LayoutMultiLineTextEx()` function have the same meanings as the parameters with the same names passed to the `LayoutMultiLineText()` function. The additional `fontIndexBuffer` parameter optionally points to an array that receives the index of the font used by each glyph.

LayoutSlug() function

The `LayoutSlug()` function generates the glyph indices, drawing positions, transformations, and colors for a single line of text, or "slug".

Prototype

```
int32 LayoutSlug(const CompiledText *compiledText,
                 const GlyphRange *glyphRange,
                 const FontHeader *fontHeader,
                 const Point2D& position,
                 int32 *glyphIndexBuffer,
                 Point2D *positionBuffer,
                 Matrix2D *matrixBuffer,
                 Color4U *colorBuffer,
                 PlaceholderBuffer *placeholderBuffer = nullptr,
                 Point2D *exitPosition = nullptr);
```

Parameters

Parameter	Description
`compiledText`	A pointer to a `CompiledText` object returned by a preceding call to the `CompileString()` function.
`glyphRange`	A pointer to a `GlyphRange` structure specifying the range of glyphs to process. This parameter can be `nullptr`, in which case all of the glyphs stored in the `CompiledText` object are processed.
`fontHeader`	A pointer to the `FontHeader` structure retrieved with the `GetFontHeader()` function for a particular `.slug` file.
`position`	The *x* and *y* coordinates of the first glyph at the baseline.
`glyphIndexBuffer`	A pointer to an array of integers that receives glyph indices. This parameter cannot be `nullptr`, and the array it points to must be at least as large as the glyph count returned by the `CountSlug()` function.
`positionBuffer`	A pointer to an array of `Point2D` structures that receives glyph positions. This parameter cannot be `nullptr`, and the array it points to must be at least as large as the glyph count returned by the `CountSlug()` function.

`matrixBuffer`	A pointer to an array of `Matrix2D` structures that receives glyph transforms. This parameter can be `nullptr`, in which case no per-glyph transforms are returned. Otherwise, this array must be at least as large as the glyph count returned by the `CountSlug()` function.
`colorBuffer`	A pointer to an array of `Color4U` structures that receives glyph colors. This parameter can be `nullptr`, in which case no per-glyph colors are returned. Otherwise, this array must be at least as large as the glyph count returned by the `CountSlug()` function.
`placeholderBuffer`	A pointer to a `PlaceholderBuffer` structure containing information about where the output placeholder data is stored. This parameter can be `nullptr`, in which case no placeholder data is generated.
`exitPosition`	A pointer to a `Point2D` structure that receives the *x* and *y* coordinates of the new drawing position after it has been advanced past the final glyph. This parameter can be `nullptr`, in which case the updated position is not returned.

Description

The `LayoutSlug()` function generates the glyph indices, drawing positions, transformations, and colors for a single line of text, or "slug". The text is processed in the same way that it is for the `BuildSlug()` function. After possible modification by the application, the information returned by the `LayoutSlug()` function can be passed to the `AssembleSlug()` function to generate vertex and triangle data.

The `compiledText` parameter should be a pointer to a `CompiledText` structure returned by a preceding call to the `CompileString()` function. A pointer to a `GlyphRange` structure may be passed to the `glyphRange` parameter to specify that only a subset of glyphs are to be processed. The pointer passed to the `fontHeader` parameter must be the same that was passed to the `fontHeader` parameter of the `CompileString()` function.

Before the `LayoutSlug()` function can be called, the `CountSlug()` function must be called for the same compiled string to determine the maximum amount of storage that the `LayoutSlug()` function will need to write its data. The compiled string must be exactly the same for both functions to ensure that the correct amount of storage can be allocated and that the data generated by the `LayoutSlug()` function stays within the calculated limits.

The `position` parameter specifies the *x* and *y* coordinates of the left side of the first glyph at the baseline. This is often (0, 0) when the transformation matrix applied externally by the application includes an object-space position.

The `glyphIndexBuffer` parameter points to an array into which glyph indices are written, and the `positionBuffer` parameter points to an array into which glyph positions are written. Each glyph position accounts for tracking, kerning, the *x* and *y* text offsets, and combining mark attachments. Glyph indices and positions are always returned, and these two parameters cannot be `nullptr`.

The `matrixBuffer` parameter optionally points to an array into which glyph transforms are written. Each glyph transform accounts for the font size, font stretch, *x* and *y* text scales, and the text skew that would be applied by the `BuildSlug()` function. The `matrixBuffer` parameter can be `nullptr` if this information is not needed.

The `colorBuffer` parameter optionally points to an array into which glyph colors are written. The `colorBuffer` parameter can be `nullptr` if this information is not needed.

The number of entries written to each array is always the same as the number of glyphs returned by the `CountSlug()` function. The data written to the output buffers includes glyphs that do not have any geometry, such as the glyph corresponding to the space character. Glyph effects and text decorations are ignored and have do not affect the number of glyphs.

If placeholders are being used, the `placeholderBuffer` parameter points to a `PlaceholderBuffer` structure containing the address of the storage into which placeholder information is written. Upon return from the `LayoutSlug()` function, the `PlaceholderBuffer` structure is updated so that the `placeholderData` field points to the next element past the data that was written. The actual number of placeholders generated by the `LayoutSlug()` function should be determined by examining the pointer in the `PlaceholderBuffer` structure upon return and subtracting the original value of that pointer. The resulting difference can be less than the maximum value returned by the `CountSlug()` function.

If the `exitPosition` parameter is not `nullptr`, then the final drawing position is written to it. The final drawing position corresponds to the position after the advance width for the final glyph has been applied along with any tracking that may be in effect.

Any characters in the original text string designated as control characters by the Unicode standard do not generate any output. These characters never contribute any spacing in the slug layout, even if the original font defines nonzero advance widths for them.

LayoutSlugEx() function

The `LayoutSlugEx()` function generates the glyph indices, drawing positions, transformations, and colors for a single line of text, or "slug".

Prototype

```
int32 LayoutSlugEx(const CompiledText *compiledText,
                   const GlyphRange *glyphRange,
                   int32 fontCount,
                   const FontDesc *fontDesc,
                   const Point2D& position,
                   uint8 *fontIndexBuffer,
                   int32 *glyphIndexBuffer,
                   Point2D *positionBuffer,
                   Matrix2D *matrixBuffer,
                   Color4U *colorBuffer,
                   PlaceholderBuffer *placeholderBuffer = nullptr,
                   Point2D *exitPosition = nullptr);
```

Parameters

Parameter	Description
`compiledText`	A pointer to a `CompiledText` object returned by a preceding call to the `CompileStringEx()` function.
`glyphRange`	A pointer to a `GlyphRange` structure specifying the range of glyphs to process. This parameter can be `nullptr`, in which case all of the glyphs stored in the `CompiledText` object are processed.
`fontCount`	The total number of fonts that may be utilized. This must be at least 1.
`fontDesc`	A pointer to an array of `FontDesc` structures describing the fonts that may be utilized. The number of elements in this array must be equal to the value of the `fontCount` parameter.
`position`	The *x* and *y* coordinates of the first glyph at the baseline.
`fontIndexBuffer`	A pointer to an array of integers that receives font indices. This parameter can be `nullptr`, in which case no per-glyph font indices are returned. Otherwise, this array must be at least as large as the glyph count returned by the `CountSlugEx()` function.

`glyphIndexBuffer`	A pointer to an array of integers that receives glyph indices. This parameter cannot be `nullptr`, and the array it points to must be at least as large as the glyph count returned by the `CountSlugEx()` function.
`positionBuffer`	A pointer to an array of `Point2D` structures that receives glyph positions. This parameter cannot be `nullptr`, and the array it points to must be at least as large as the glyph count returned by the `CountSlugEx()` function.
`matrixBuffer`	A pointer to an array of `Matrix2D` structures that receives glyph transforms. This parameter can be `nullptr`, in which case no per-glyph transforms are returned. Otherwise, this array must be at least as large as the glyph count returned by the `CountSlugEx()` function.
`colorBuffer`	A pointer to an array of `Color4U` structures that receives glyph colors. This parameter can be `nullptr`, in which case no per-glyph colors are returned. Otherwise, this array must be at least as large as the glyph count returned by the `CountSlugEx()` function.
`placeholderBuffer`	A pointer to a `PlaceholderBuffer` structure containing information about where the output placeholder data is stored. This parameter can be `nullptr`, in which case no placeholder data is generated.
`exitPosition`	A pointer to a `Point2D` structure that receives the *x* and *y* coordinates of the new drawing position after it has been advanced past the final glyph. This parameter can be `nullptr`, in which case the updated position is not returned.

Description

The `LayoutSlugEx()` function is an extended version of the `LayoutSlug()` function capable of handling multiple fonts through the mapping mechanism described in Section 4.6. A call to the `LayoutSlug()` function is internally forwarded to the `LayoutSlugEx()` function with the `fontCount` parameter set to 1, the `fontDesc` parameter set to the address of a single `FontDesc` structure containing the font header with default scale and offset, the `fontMap` parameter set to `nullptr`, and the `fontIndexBuffer` parameter set to `nullptr`.

The `compiledText` parameter should be a pointer to a `CompiledText` structure returned by a preceding call to the `CompileStringEx()` function. A pointer to a `GlyphRange` structure may be passed to the `glyphRange` parameter to specify that only a subset of glyphs are to be processed. The value of the `fontCount` parameter and the entries of the array specified by the `fontDesc` parameter must be exactly the same values that were passed to the `fontCount` and `fontDesc` parameters of the `CompileStringEx()` function.

After possible modification by the application, the information returned by the `LayoutSlugEx()` function can be passed to the `AssembleSlugEx()` function to generate vertex and triangle data.

After the first four parameters, the remaining parameters passed to the `LayoutSlugEx()` function have the same meanings as the parameters with the same names passed to the `LayoutSlug()` function. The additional `fontIndexBuffer` parameter optionally points to an array that receives the index of the font used by each glyph.

LineData structure

The `LineData` structure contains information about a line of text.

Fields

Field	Description
`int32` `glyphCount`	The number of glyphs belonging to the full line of text, including glyphs corresponding to break characters and trimmed characters.
`uint32` `lineFlags`	Flags indicating various properties of the line of text. See the description for information about the individual bits.
`float` `lineSpan`	The span of the line of text, in absolute units, excluding trimmed characters at the end.
`int32` `trimTextLength`	The length, in bytes, of the string composing the line of text after excluding trimmed characters at the end.
`int32` `fullTextLength`	The length, in bytes, of the string composing the full line of text, including trimmed characters at the end.

Description

The `LineData` structure contains information about the byte length and physical horizontal span of a line of text. The `LineData` structure adds fields to the `GlyphRange` base structure.

The `firstGlyph` and `lastGlyph` fields of the `GlyphRange` base structure represent the first glyph at the beginning of the line and the last glyph on the line corresponding to an untrimmed character. This range can be empty if the line contains no untrimmed characters. The `glyphCount` field contains the total number of glyphs belonging to the line beginning with the first glyph. The glyph count includes any break character at the end of the line and all trimmed characters at the end of the line, but it excludes the null terminator if one occurs at the end of the line. If the `LineData` structure does not correspond to the final line in a block of text, then the first glyph on the next line of text is always given by the sum of the `firstGlyph` and `glyphCount` fields.

The `lineFlags` field can be zero or the following value.

Value	Description
`kLineParagraphLast`	The line is the last line in a paragraph. If paragraph attributes are enabled for multi-line text, this flag means that paragraph spacing is applied after this line and the next line is indented as the first line in a new paragraph.

The `lineSpan` field contains the horizontal span of the untrimmed text belonging to the line, in absolute units. This is the total sum of the advance widths of the glyphs that fit between the left and right margins of a paragraph.

The `trimTextLength` and `fullTextLength` fields contain the byte lengths in the original text string corresponding to the trimmed and untrimmed glyphs belonging to the line.

LineData structures are typically generated by the library and are not usually created by the application. A single `LineData` structure is generated by the `BreakSlug()` function, and multiple `LineData` structures can be generated by the `BreakMultiLineText()` function. `LineData` structures are consumed by the `CountMultiLineText()`, `BuildMultiLineText()`, and `LayoutMultiLineText()` functions.

LocateSlug() function

The `LocateSlug()` function determines caret positioning information for specific byte locations within a text string representing a single line of text, or "slug".

Prototype

```
void LocateSlug(const CompiledText *compiledText,
                const GlyphRange *glyphRange,
                const FontHeader *fontHeader,
                int32 locationCount,
                const int32 *byteOffsetArray,
                LocationData *locationArray);
```

Parameters

Parameter	Description
`compiledText`	A pointer to a `CompiledText` object returned by a preceding call to the `CompileString()` function.
`glyphRange`	A pointer to a `GlyphRange` structure specifying the range of glyphs to process. This parameter can be `nullptr`, in which case all of the glyphs stored in the `CompiledText` object are processed.
`fontHeader`	A pointer to the `FontHeader` structure retrieved with the `GetFontHeader()` function for a particular `.slug` file.
`locationCount`	The number of byte offsets stored in the array specified by the `byteOffsetArray` parameter, and the number of `LocationData` entries returned in the array specified by the `locationArray` parameter.
`byteOffsetArray`	A pointer to an array of integers that each specify a byte location within the original text string. The byte offsets in this array must be sorted in ascending order.
`locationArray`	A pointer to an array of `LocationData` structures that each receives caret position and glyph information. This array must be large enough to hold `locationCount` entries.

Description

The `LocateSlug()` function determines where an insertion caret should be placed for a set of byte locations within a text string and returns a `LocationData` structure for each one.

The `compiledText` parameter should be a pointer to a `CompiledText` structure returned by a preceding call to the `CompileString()` function. A pointer to a `GlyphRange` structure may be passed to the `glyphRange` parameter to specify that only a subset of glyphs are to be processed. The pointer passed to the `fontHeader` parameter must be the same that was passed to the `fontHeader` parameter of the `CompileString()` function.

If any of the byte offsets specified by the `byteOffsetArray` parameter precede the characters covered by the range of glyphs specified by the `glyphRange` parameter, then the information returned for that byte offset is the same as if the byte offset corresponded to the first character covered by the glyph range. If any of the byte offsets are greater than the offset of the final character covered by the glyph range, then the information returned for that byte offset is the same as if the byte offset corresponded to the first character following the end of the glyph range.

If any of the byte offsets are greater than or equal to the length of the entire text string (and the end of the string is included in the glyph range), then the information returned for that byte offset corresponds to the terminator character, and the caret position corresponds to the end of the string.

Each byte offset should ideally specify the first byte in any multibyte UTF-8 encoding sequence. However, in the case that any offset actually corresponds to a continuation byte, the information returned is the same as if the offset corresponded to the first byte of the sequence.

LocateSlugEx() function

The `LocateSlugEx()` function determines caret positioning information for specific byte locations within a text string representing a single line of text, or "slug".

Prototype

```
void LocateSlugEx(const CompiledText *compiledText,
                  const GlyphRange *glyphRange,
                  int32 fontCount,
                  const FontDesc *fontDesc,
                  int32 locationCount,
                  const int32 *byteOffsetArray,
                  LocationData *locationArray);
```

Parameters

Parameter	Description
`compiledText`	A pointer to a `CompiledText` object returned by a preceding call to the `CompileString()` function.
`glyphRange`	A pointer to a `GlyphRange` structure specifying the range of glyphs to process. This parameter can be `nullptr`, in which case all of the glyphs stored in the `CompiledText` object are processed.
`fontCount`	The total number of fonts that may be utilized. This must be at least 1.
`fontDesc`	A pointer to an array of `FontDesc` structures describing the fonts that may be utilized. The number of elements in this array must be equal to the value of the `fontCount` parameter.
`locationCount`	The number of byte offsets stored in the array specified by the `byteOffsetArray` parameter, and the number of `LocationData` entries returned in the array specified by the `locationArray` parameter.
`byteOffsetArray`	A pointer to an array of integers that each specify a byte location within the original text string. The byte offsets in this array must be sorted in ascending order.
`locationArray`	A pointer to an array of `LocationData` structures that each receives caret position and glyph information. This array must be large enough to hold `locationCount` entries.

Description

The `LocateSlugEx()` function determines where an insertion caret should be placed for a set of byte locations within a text string and returns a `LocationData` structure for each one. A call to the `LocateSlug()` function is internally forwarded to the `LocateSlugEx()` function with the `fontCount` parameter set to 1, the `fontDesc` parameter set to the address of a single `FontDesc` structure containing the font header with default scale and offset, the `fontMap` parameter set to `nullptr`, and the `fontIndexBuffer` parameter set to `nullptr`.

The `compiledText` parameter should be a pointer to a `CompiledText` structure returned by a preceding call to the `CompileStringEx()` function. A pointer to a `GlyphRange` structure may be passed to the `glyphRange` parameter to specify that only a subset of glyphs are to be processed. The value of the `fontCount` parameter and the entries of the array specified by the `fontDesc` parameter must be exactly the same values that were passed to the `fontCount` and `fontDesc` parameters of the `CompileStringEx()` function.

After the first four parameters, the remaining parameters passed to the `LocateSlugEx()` function have the same meanings as the parameters with the same names passed to the `LocateSlug()` function.

LocationData structure

The `LocationData` structure contains information about a caret location within a text string.

Fields

Field	Description
`Point2D caretPosition`	The caret position corresponding to the character in the text string at the specified byte offset. This is usually the position on the baseline at which the character's glyph is drawn, but in the case that the glyph is a ligature for multiple characters, the *x* coordinate is advanced in the glyph's run direction by the distance in the `subglyphOffset` field.
`float dualCaretOffset`	The horizontal offset to the dual caret position in bidirectional text when the caret position falls on the boundary between directional runs. This is zero unless the main caret position falls at the beginning of a new run direction, in which case it specifies the delta between the main caret *x* coordinate and dual caret *x* coordinate at the ending position of the previous run.
`int32 glyphNumber`	The number of glyphs preceding the glyph corresponding to the character in the text string. This can be used to index into the `compiledGlyph` array in the `CompiledText` structure.
`uint32 subglyphIndex`	The zero-based index within a ligature glyph, if applicable. This is always zero for glyphs that correspond to only a single character.
`float subglyphOffset`	The horizontal offset of the caret position within a ligature glyph. Subtracting this value from the *x* coordinate of the `caretPosition` field gives the beginning position of the entire glyph in its run direction. This is always zero for glyphs that correspond to only a single character.

Description

The `LocationData` structure is returned by the `LocateSlug()` function, and it contains information about the caret position corresponding to a specified byte offset within a text string.

The `caretPosition` field is usually the position on the baseline at which the character's glyph is drawn, relative to an origin at the beginning of the text string. The `glyphNumber` field contains the number of preceding glyphs, and it can be used to index into the `compiledGlyph` array in the `CompiledText` structure to access the glyph that would be drawn at the caret position.

In the case that the glyph is a ligature for multiple characters, the `subglyphIndex` field contains the number of characters preceding the caret location within the ligature. The `subglyphOffset` field

contains the additional horizontal offset of the caret position to account for its location within the ligature. This offset is already included in the *x* coordinate of the `caretPosition` field, so subtracting it gives the beginning position of the entire glyph in its run direction. The value of `subglyphOffset` is positive or zero in a left-to-right run, and it is negative or zero in a right-to-left run.

MakeCompactCompiledText() function

The `MakeCompactCompiledText()` function writes a compiled text string in a compact memory buffer of minimum size.

Prototype

```
const CompiledText *MakeCompactCompiledText(const CompiledText *compiledText,
                                            void *compactStorage);
```

Parameters

Parameter	Description
`compiledText`	A pointer to a `CompiledText` structure containing a compiled string of text.
`compactStorage`	A pointer to the memory buffer to which the compiled text data is written.

Description

The `MakeCompactCompiledText()` function creates a compact version of a compiled text string suitable for long-term storage. The `compiledText` parameter must point to a `CompiledText` structure that was previously returned by the `CompileString()` function. The information stored in that object is copied to the memory buffer specified by the `compactStorage` parameter in a compact form, and a pointer to that memory buffer is returned after being cast to a pointer to a `CompiledText` structure. The return value can be passed to any library function that accepts a `CompiledText` structure as a parameter.

Before calling the `MakeCompactCompiledText()` function, the application must call the `GetCompact-CompiledStorageSize()` function to determine how large the memory buffer needs to be. It is the application's responsibility to allocate the memory buffer and release it when it's no longer in use.

MeasureSlug() function

The `MeasureSlug()` function measures the physical horizontal span of a line of text, or "slug".

Prototype

```
float MeasureSlug(const CompiledText *compiledText,
                  const GlyphRange *glyphRange,
                  const FontHeader *fontHeader,
                  int32 trimCount = 0,
                  const uint32 *trimArray = nullptr,
                  float *trimSpan = nullptr);
```

Parameters

Parameter	Description
`compiledText`	A pointer to a `CompiledText` object returned by a preceding call to the `CompileString()` function.
`glyphRange`	A pointer to a `GlyphRange` structure specifying the range of glyphs to process. This parameter can be `nullptr`, in which case all of the glyphs stored in the `CompiledText` object are processed.
`fontHeader`	A pointer to the `FontHeader` structure retrieved with the `GetFontHeader()` function for a particular `.slug` file.
`trimCount`	The number of trim characters specified by the `trimArray` parameter.
`trimArray`	A pointer to an array of trim characters with `trimCount` entries. The values in this array are Unicode characters, and they must be sorted in ascending order. This parameter can `nullptr` only if the `trimCount` parameter is 0.
`trimSpan`	A pointer to a location to which the trimmed physical horizontal span of the string is written. This parameter can be `nullptr`, in which case the trimmed span is not returned.

Description

The `MeasureSlug()` function calculates the total physical span for a line of text and returns the horizontal difference between the initial and final drawing positions for the set of glyphs that would be generated for the text using the same font and layout state.

The `compiledText` parameter should be a pointer to a `CompiledText` structure returned by a preceding call to the `CompileString()` function. A pointer to a `GlyphRange` structure may be passed to the

`glyphRange` parameter to specify that only a subset of glyphs are to be processed. The pointer passed to the `fontHeader` parameter must be the same that was passed to the `fontHeader` parameter of the `CompileString()` function.

The return value is the physical horizontal span of the entire set of characters that were processed. If the `trimSpan` parameter is not `nullptr`, then a possibly shorter span that excludes a set of specific characters at the end of the string is written to the location that the `trimSpan` parameter points to. The set of excluded characters is specified by the `trimCount` and `trimArray` parameters. If `trimCount` is not zero, then the `trimArray` parameter must point to an array of Unicode characters having the number of entries specified by `trimCount`. Values specified in this array typically include spaces and other characters that do not generate any geometry.

Any characters in the original text string designated as control characters by the Unicode standard do not contribute to the span measurement. These characters never contribute any spacing in the slug layout, even if the original font defines nonzero advance widths for them.

MeasureSlugEx() function

The `MeasureSlugEx()` function measures the physical horizontal span of a line of text, or "slug".

Prototype

```
float MeasureSlugEx(const CompiledText *compiledText,
                    const GlyphRange *glyphRange,
                    int32 fontCount,
                    const FontDesc *fontDesc,
                    int32 trimCount = 0,
                    const uint32 *trimArray = nullptr,
                    float *trimSpan = nullptr);
```

Parameters

Parameter	Description
`compiledText`	A pointer to a `CompiledText` object returned by a preceding call to the `CompileStringEx()` function.
`glyphRange`	A pointer to a `GlyphRange` structure specifying the range of glyphs to process. This parameter can be `nullptr`, in which case all of the glyphs stored in the `CompiledText` object are processed.
`fontCount`	The total number of fonts that may be utilized. This must be at least 1.
`fontDesc`	A pointer to an array of `FontDesc` structures describing the fonts that may be utilized. The number of elements in this array must be equal to the value of the `fontCount` parameter.
`trimCount`	The number of trim characters specified by the `trimArray` parameter.
`trimArray`	A pointer to an array of trim characters with `trimCount` entries. The values in this array are Unicode characters, and they must be sorted in ascending order. This parameter can `nullptr` only if the `trimCount` parameter is 0.
`trimSpan`	A pointer to a location to which the trimmed physical horizontal span of the string is written. This parameter can be `nullptr`, in which case the trimmed span is not returned.

Description

The `MeasureSlugEx()` function is an extended version of the `MeasureSlug()` function capable of handling multiple fonts through the mapping mechanism described in Section 4.6. A call to the

`MeasureSlug()` function is internally forwarded to the `MeasureSlugEx()` function with the `fontCount` parameter set to 1, the `fontDesc` parameter set to the address of a single `FontDesc` structure containing the font header with default scale and offset, and the `fontMap` parameter set to `nullptr`.

The `compiledText` parameter should be a pointer to a `CompiledText` structure returned by a preceding call to the `CompileStringEx()` function. A pointer to a `GlyphRange` structure may be passed to the `glyphRange` parameter to specify that only a subset of glyphs are to be processed. The value of the `fontCount` parameter and the entries of the array specified by the `fontDesc` parameter must be exactly the same values that were passed to the `fontCount` and `fontDesc` parameters of the `CompileStringEx()` function.

After the first three parameters, the remaining parameters passed to the `MeasureSlugEx()` function have the same meanings as the parameters with the same names passed to the `MeasureSlug()` function.

PictureData structure

The `PictureData` structure contains information about a specific picture.

Fields

Field	Description
`uint32 pictureFlags`	Flags indicating various properties of the picture.
`Box2D canvasBox`	The box defining the full area of the canvas for the picture.
`Box2D boundingBox`	The box representing the bounding box of all geometry contained in the picture.
`int32 componentCount`	The number of icon components making up the picture.
`int32 componentDataOffset`	The offset to the component data table.

Description

Each picture in an album has an associated `PictureData` structure that contains information about its rendering and layout properties.

The following values can be combined (through logical OR) in the `pictureFlags` field.

Value	Description
`kPictureLinearFills`	The picture contains at least one fill that is defined entirely by linear curves. The `kRenderLinearCurves` flag may be specified for the `GetShaderIndices()` function to enable the shader optimization, but it is not required.
`kPictureStrokes`	The picture contains at least one stroke. The `kRenderStrokes` flag must be specified for the `GetShaderIndices()` function in order to render the picture. The `kRenderStrokes` flag is not optional, and it is not the case that strokes simply will not be rendered without it.
`kPictureGradients`	The picture contains at least one gradient. The `kRenderGradients` flag should be specified for the `GetShaderIndices()` function in order to render the gradients. The `kRenderGradients` flag is not strictly required, and gradient fills will simply be rendered as solid colors without it.

PlaceholderBuffer structure

The `PlaceholderBuffer` structure contains a pointer to the storage location where placeholder data is written.

Fields

Field	Description
`PlaceholderData *placeholderData`	A pointer to the location where placeholder data structures are written.

Description

The `PlaceholderBuffer` structure contains information that tells several functions where to write the placeholder data that they generate when placeholders are enabled.

When a build function returns, the `placeholderData` field of the `PlaceholderBuffer` structure has been updated to point to the end of the data that was written, and the same `PlaceholderBuffer` structure can be passed to additional function calls to append more data to the same buffer.

PlaceholderData structure

The `PlaceholderData` structure contains information about a placeholder that was encountered when text was laid out by the `BuildSlug()` function, `LayoutSlug()` function, `BuildMultiLineText()` function, or `LayoutMultiLineText()` function.

Fields

Field	Description
`int32 glyphNumber`	The number of glyphs preceding the glyph corresponding to the placeholder in the text string. This can be used to index into the `compiledGlyph` array in the `CompiledText` structure.
`int32 placeholderIndex`	The index of the placeholder, given by the difference between the Unicode value representing the placeholder and the `placeholderBase` field in the `LayoutData` structure.
`Point2D placeholderPosition`	The *x* and *y* coordinates of the placeholder's position where it occurs in the text, in absolute units.

Description

When placeholders are being used as text is laid out, information about where they occurred is returned through `PlaceholderData` structures. The `BuildSlug()`, `LayoutSlug()`, `BuildMultiLineText()`, and `LayoutMultiLineText()` functions each accept an optional pointer to a `PlaceholderBuffer` structure containing a pointer to the array of `PlaceholderData` structures allocated by the application. The index of each placeholder and its position inside the laid-out text is stored in this array so the application knows where to draw its own graphics in the space reserved by the placeholders.

ResolveGlyph() function

The `ResolveGlyph()` function performs alternate substitution for a specific glyph.

Prototype

```
int32 ResolveGlyph(const FontHeader *fontHeader,
                   int32 glyphIndex,
                   uint32 alternateMask,
                   uint32 styleIndex = 0);
```

Parameters

Parameter	Description
`fontHeader`	A pointer to the `FontHeader` structure retrieved with the `GetFontHeader()` function for a particular `.slug` file.
`glyphIndex`	The index of the glyph to resolve.
`alternateMask`	A mask specifying the alternate substitution types that may be applied.
`styleIndex`	The style set to be applied when the `kAlternateStylistic` bit is included in the `alternateMask` parameter. This should be in the range 1–20.

Description

The `ResolveGlyph()` function determines whether an alternate should be substituted for a glyph, and if so, returns the index of the substitute glyph. If no substitution is performed, then the return value is equal to the `glyphIndex` parameter.

The `glyphIndex` parameter specifies the index of the glyph to resolve and is typically obtained with the `GetGlyphIndex()` function for a particular Unicode character. The `alternateMask` parameter specifies the alternate substitution types that may be applied, and it can be a combination (through logical OR) of the bit values that can be set for the `alternateMask` field of the `LayoutData` structure. If the `kAlternateStylistic` bit is included in the mask, then the `styleIndex` parameter specifies the style set to be applied in the range 1–20. If the `kAlternateStylistic` bit is not set, then the `styleIndex` parameter is ignored and should be set to zero.

If no alternate data was imported from the original font, then the `ResolveGlyph()` function always returns the value specified by the `glyphIndex` parameter.

RunData structure

The `RunData` structure contains information about a directional run within a text string.

Fields

Field	Description
`float runDirection`	The relative run direction. This has a value of +1.0 for the primary writing direction and −1.0 for the opposite writing direction.

Description

The `CompiledText` structure contains an array of `RunData` structures that contain information about each distinct directional run within a text string.

SetDefaultFillData() function

The `SetDefaultFillData()` function initializes all fields of the `FillData` structure to their default values.

Prototype

```
void SetDefaultFillData(FillData *fillData);
```

Parameters

Parameter	Description
`fillData`	A pointer to a `FillData` structure to which the default values are written.

Description

The `SetDefaultStrokeData()` function initializes all fields of the `FillData` structure to the default values specified in the following table.

Field	Default Value
`fillColor`	{0, 0, 0, 255}
`gradientType`	`kGradientNone`
`gradientLine`	{0.0, 0.0, 0.0}
`gradientCircle`	{0.0, 0.0, 0.0}
`gradientColor[0]`	{0, 0, 0, 255}
`gradientColor[1]`	{0, 0, 0, 255}

SetDefaultLayoutData() function

The `SetDefaultLayoutData()` function initializes all fields of the `LayoutData` structure to their default values.

Prototype

```
void SetDefaultLayoutData(LayoutData *layoutData);
```

Parameters

Parameter	Description
`layoutData`	A pointer to a `LayoutData` structure to which the default values are written.

Description

The `SetDefaultLayoutData()` function initializes all fields of the `LayoutData` structure to the default values specified in the following table.

Field	Default Value
`fontType`	0
`fontSize`	16.0
`fontStretch`	1.0
`textTracking`	0.0
`textSkew`	0.0
`textScale`	{1.0, 1.0}
`textOffset`	{0.0, 0.0}
`textColor.color[0]`	{0, 0, 0, 255}
`textColor.color[1]`	{0, 0, 0, 255}
`textColor.gradient[0]`	0.0
`textColor.gradient[1]`	1.0
`textColor.gradientFlag`	`false`
`textAlignment`	`kAlignmentLeft`

textLeading	1.2
paragraphSpacing	0.0
leftMargin	0.0
rightMargin	0.0
firstLineIndent	0.0
tabSize	64.0
tabRound	0.0
layoutFlags	0
renderFlags	0
geometryType	kGeometryQuads
formatMask	~0 (all enabled)
sequenceMask	kSequenceDefaultMask
alternateMask	0
styleIndex	0
scriptLevel	0
decorationFlag[]	false (all entries)
spaceCount	0
spaceArray	nullptr
placeholderBase	0x0F0000
placeholderCount	0
placeholderWidthArray	nullptr
effectType	kGlyphEffectNone
effectOffset	{0.0, 0.0}
effectColor.color[0]	{0, 0, 0, 255}
effectColor.color[1]	{0, 0, 0, 255}
effectColor.gradient[0]	0.0
effectColor.gradient[1]	1.0

`effectColor.gradientFlag`	`false`
`objectScale`	{1.0, 1.0}
`objectOffset`	{0.0, 0.0}
`clipLeft`	0.0
`clipRight`	0.0
`missingGlyphIndex`	0
`defaultLayoutData`	`nullptr`

SetDefaultStrokeData() function

The `SetDefaultStrokeData()` function initializes all fields of the `StrokeData` structure to their default values.

Prototype

```
void SetDefaultStrokeData(StrokeData *strokeData);
```

Parameters

Parameter	Description
`strokeData`	A pointer to a `StrokeData` structure to which the default values are written.

Description

The `SetDefaultStrokeData()` function initializes all fields of the `StrokeData` structure to the default values specified in the following table.

Field	Default Value
`strokeWidth`	1.0
`strokeColor`	{0, 0, 0, 255}
`strokeCapType`	`kStrokeCapFlat`
`strokeJoinType`	`kStrokeJoinBevel`
`miterLimit`	4.0
`dashCount`	0
`dashOffset`	0.0
`dashArray`	`nullptr`

SlugFileHeader structure

The `SlugFileHeader` structure contains general information about a font or album.

Fields

Field	Description
`uint32 resourceSignature`	An internal signature for `.slug` files.
`uint32 resourceVersion`	The version of Slug that generated the `.slug` file.
`uint32 resourceType`	The type of Slug resource, either `kResourceFont` or `kResourceAlbum`.
`uint32 resourceCount`	The number of separate resources in the `.slug` file. This can be greater than one for fonts, but it is always one for albums.
`TextureType curveTextureType`	The format of the curve texture, which is either `kTextureFloat16` or `kTextureFloat32`.
`Integer2D curveTextureSize`	The dimensions of the texture map containing the control points for the quadratic Bézier curves.
`uint32 curveCompressionType`	The compression type applied to the curve texture map.
`uint32 curveCompressedDataSize`	The size of the compressed curve texture map data, in bytes.
`int32 curveTextureOffset`	The offset to the compressed curve texture map data, in bytes, from the beginning of this header structure.
`TextureType bandTextureType`	The format of the band texture, which is always `kTextureUint16`.
`Integer2D bandTextureSize`	The dimensions of the texture map containing the multi-band Bézier curve index data.
`uint32 bandCompressionType`	The compression type applied to the band texture map.
`uint32 bandCompressedDataSize`	The size of the compressed band texture map data, in bytes.
`int32 bandTextureOffset`	The offset to the compressed band texture map data, in bytes, from the beginning of this header structure.

Description

The `SlugFileHeader` structure contains information that is common to both fonts and albums. Every `.slug` file begins with a `SlugFileHeader` structure beginning with the first byte of its contents. A pointer to a `FontHeader` structure or `AlbumHeader` structure can be obtained by calling the `GetFontHeader()` function or `GetAlbumHeader()` function.

StrokeData structure

The `StrokeData` structure controls the options that determine the appearance of a stroked path.

Fields

Field	Description
`float` `strokeWidth`	The stroke width, in object space. Half of this width lies on each side of a path.
`Color4U` `strokeColor`	The color of the stroke.
`StrokeCapType` `strokeCapType`	The type of cap applied at the beginning and end of the stroke and on the ends of dashes.
`StrokeJoinType` `strokeJoinType`	The type of join applied between consecutive curves when their tangents are not parallel and the miter limit is exceeded.
`float` `miterLimit`	The minimum ratio of the miter width to the stroke width for which consecutive curves are joined as specified by the `strokeJoinType` field instead of a miter. If the miter limit is zero, then curves with non-parallel tangents are always joined as specified by the `strokeJoinType` field.
`int32` `dashCount`	The sum of the number of dash lengths and gap lengths contained in the array specified by the `dashArray` field. This must be an even number less than 256. If this is not a positive number, then no dashing is applied.
`float` `dashOffset`	The length offset within the dash array at which the stroke begins. This is ignored if the `dashCount` field is not a positive number.
`const float` `*dashArray`	A pointer to an array of dash and gap lengths. The number of elements in this array must be equal to the value of the `dashCount` field. Even numbered entries specify dash lengths, and odd numbered entries specify the gap lengths between dashes. This is ignored if the `dashCount` field is not a positive number.

Description

The `StrokeData` structure controls the options that determine the appearance of a stroked path. The fields of the `StrokeData` structure can be initialized to their default values by calling the `SetDefaultStrokeData()` function.

The following values are the cap types that can be specified in the `strokeCapType` field.

Value	Description
`kStrokeCapFlat`	Strokes do not have caps.
`kStrokeCapTriangle`	Strokes have triangular caps.
`kStrokeCapSquare`	Strokes have square caps.
`kStrokeCapRound`	Strokes have round caps.

The following values are the join types that can be specified in the `strokeJoinType` field.

Value	Description
`kStrokeJoinBevel`	Strokes are joined by beveled corners.
`kStrokeJoinRound`	Strokes are joined by rounded corners.

StrokeWorkspace structure

The `StrokeWorkspace` structure is used internally for temporary storage by the library functions that generate geometry and texture data for strokes.

Description

The `StrokeWorkspace` structure serves as temporary storage space while data is being processed by the Slug library functions that generate geometry and texture data for strokes. When the library is used in a single-threaded context, there is no need to allocate and specify `StrokeWorkspace` structures because the library can use its own internal storage. However, if these library functions are called from multiple threads, then the application must ensure that a different `StrokeWorkspace` structure is specified for each thread so that the library is safely reentrant.

`StrokeWorkspace` structures allocated by the application are passed to Slug library functions that need them as the last parameter. Due to the large size of the `StrokeWorkspace` structures, they should not be allocated on the stack, but only on the heap or as a static part of the program binary.

TestData structure

The `TestData` structure holds information about how a test position interacts with a line of text.

Fields

Field	Description
`bool` `trailingHitFlag`	A flag indicating whether the test position is greater than the center position of the intersected glyph.
`bool` `rightToLeftFlag`	A flag indicating whether the intersected glyph belongs to a right-to-left run of text.
`bool` `subligatureFlag`	A flag indicating that the caret position has been placed after at least one, but not all, of the characters composing a ligature glyph.
`float` `positionOffset`	The distance between the beginning position of the intersected glyph and the test position.
`float` `advanceWidth`	The advance width of the intersected glyph. If the test position is outside the text, then this is zero.
`float` `caretPosition`	The caret position derived from the test position and run direction. If the `trailingHitFlag` field is `true`, then the caret position follows the intersected glyph.
`int32` `textLength`	The length, in bytes, of the text preceding the caret position. If the `trailingHitFlag` field is `true`, then this includes the characters to which the intersected glyph corresponds.
`int32` `glyphNumber`	The number of glyphs preceding the intersected glyph. This has the same value regardless of whether the `trailingHitFlag` field is `true`.

Description

The `TestData` structure is used by the `TestSlug()` and `TestSlugEx()` functions to return information about how a test position interacts a line of text.

TestSlug() function

The `TestSlug()` function determines how a test position interacts with a single line of text, or "slug".

Prototype

```
bool TestSlug(const CompiledText *compiledText,
              const GlyphRange *glyphRange,
              const FontHeader *fontHeader,
              float position,
              TestData *testData);
```

Parameters

Parameter	Description
`compiledText`	A pointer to a `CompiledText` object returned by a preceding call to the `CompileString()` function.
`glyphRange`	A pointer to a `GlyphRange` structure specifying the range of glyphs to process. This parameter can be `nullptr`, in which case all of the glyphs stored in the `CompiledText` object are processed.
`fontHeader`	A pointer to the `FontHeader` structure retrieved with the `GetFontHeader()` function for a particular `.slug` file.
`position`	The horizontal position to test, relative to the beginning of the text.
`testData`	A pointer to a `TestData` structure in which information about the test is returned.

Description

The `TestSlug()` function determines which glyph in a line of text corresponds to a given test position and calculates the appropriate position for an insertion caret. The `text` parameter should point to a null-terminated string of characters encoded as UTF-8. The `position` parameter specifies the horizontal position to be tested, relative to an origin placed at the beginning of the text. The `testData` parameter specifies a pointer to a `TestData` structure that is filled out upon return.

The `compiledText` parameter should be a pointer to a `CompiledText` structure returned by a preceding call to the `CompileString()` function. A pointer to a `GlyphRange` structure may be passed to the `glyphRange` parameter to specify that only a subset of glyphs are to be processed. The pointer passed to the `fontHeader` parameter must be the same that was passed to the `fontHeader` parameter of the `CompileString()` function.

The `TestSlug()` function works in a symmetric manner with respect to the primary writing direction of the text, as specified by the presence or absence of the `kLayoutRightToLeft` flag in the `layoutFlags` field of the `LayoutData` structure. For text having a primary direction of left-to-right, the glyphs laid out for the text fill the range [0, *s*), where *s* is the full span of the text. For text having a primary direction of right-to-left, the glyphs fill the range (−*s*, 0]. If the `position` parameter falls inside the pertinent range, then the return value of the `TestSlug()` function is `true`, and the `TestData` structure contains information about the glyph that was hit.

The `trailingHitFlag` field of the `TestData` structure is `false` if the test position falls within the first half of the glyph that was hit, and it is `true` otherwise. The `rightToLeftFlag` field is `false` if the glyph is part of a left-to-right run of characters, and it is `true` otherwise. Whether the first half of a glyph corresponds to the left side or right side depends on the directionality of the text for that glyph.

The `positionOffset` field of the `TestData` structure specifies the difference between the beginning of the glyph that was hit and the test position. This is always a nonnegative value, and it measures from the left side of a glyph belonging to left-to-right run and from the right side of a glyph belonging to a right-to-left run.

The `advanceWidth` field of the `TestData` structure contains the advance width of the glyph that was hit.

The `caretPosition` field of the `TestData` structure is set to the position at which the insertion caret should be placed based on the test position. If the `trailingHitFlag` field is `false`, then the caret position is equal to the beginning position of the glyph that was hit, which is on the left for a left-to-right run and on the right for a right-to-left run. If the `trailingHitFlag` field is `true`, then the caret position is equal to the ending position of the glyph, which is offset from the beginning position by the glyph's advance width in the direction of the run to which it belongs.

If a glyph is followed by empty spacing due to a positive tracking value or a positive kerning adjustment against the next glyph, then that spacing is considered to be part of the trailing side of the glyph. The empty spacing does not affect the boundary between the first half and second half, but extends the second half by the width of the spacing. When the caret position is placed after a glyph, it is placed before any empty spacing.

The `textLength` field of the `TestData` structure contains the number of bytes in the original text string that correspond to the glyphs preceding the caret position. Consequently, this length includes the character(s) for the glyph that was hit only if the `trailingHitFlag` field is `true`.

The `glyphNumber` field of the `TestData` structure contains the number of glyphs preceding the glyph that was hit, which gives the index in the array of glyphs derived from the original text string. This value does not depend on the `trailingHitFlag` field.

If the glyph that was hit is a ligature, then the `TestSlug()` function can determine whether the caret position should be placed between two of the characters whose standalone glyphs were replaced by the ligature glyph. When this happens, the `subligatureFlag` field of the `TestData` structure is set to `true`, and the position in the `caretPosition` field falls somewhere in the middle of the ligature glyph. In this case, the `glyphNumber` field is not affected, but the `textLength` field is adjusted to include only the characters that precede the caret position, which is less than the total number of characters preceding

the end of the ligature glyph. The `trailingHitFlag` field is set to `true` only if the test position precedes the caret position.

If the test position given by the `position` parameter falls outside the range filled by glyphs, either because the position falls beyond the last glyph specified by the `glyphRange` parameter or beyond the null terminator, then the return value is `false`. In this case, the `trailingHitFlag` field is always `false`, the `rightToLeftFlag` field reflects the primary writing direction specified in the `LayoutData` structure, the `subligatureFlag` field is always `false`, the `positionOffset` field is always zero, and the `advanceWidth` field is always zero. If the test position is less than zero for left-to-right text, or if it is greater than zero for right-to-left text, then the `caretPosition` field is set to zero, the `textLength` field is set to zero, and the `glyphNumber` field is set to zero. If the test position lies beyond the end of the span, then the `caretPosition` field is set to the end of the span, the `textLength` field is set to the length of the original text string corresponding to the glyphs specified by the `glyphRange` parameter or the full length of the text in the case of a null terminator, and the `glyphNumber` field is set to one greater than the last glyph in the range specified by the `glyphRange` parameter or the index of the null terminator.

TestSlugEx() function

The `TestSlugEx()` function determines how a test position interacts with a single line of text, or "slug".

Prototype

```
bool TestSlugEx(const CompiledText *compiledText,
                const GlyphRange *glyphRange,
                int32 fontCount,
                const FontDesc *fontDesc,
                float position,
                TestData *testData);
```

Parameters

Parameter	Description
`compiledText`	A pointer to a `CompiledText` object returned by a preceding call to the `CompileStringEx()` function.
`glyphRange`	A pointer to a `GlyphRange` structure specifying the range of glyphs to process. This parameter can be `nullptr`, in which case all of the glyphs stored in the `CompiledText` object are processed.
`fontCount`	The total number of fonts that may be utilized. This must be at least 1.
`fontDesc`	A pointer to an array of `FontDesc` structures describing the fonts that may be utilized. The number of elements in this array must be equal to the value of the `fontCount` parameter.
`position`	The horizontal position to test, relative to the beginning of the text.
`testData`	A pointer to a `TestData` structure in which information about the test is returned.

Description

The `TestSlugEx()` function is an extended version of the `TestSlug()` function capable of handling multiple fonts through the mapping mechanism described in Section 4.6. A call to the `TestSlug()` function is internally forwarded to the `TestSlugEx()` function with the `fontCount` parameter set to 1, the `fontDesc` parameter set to the address of a single `FontDesc` structure containing the font header with default scale and offset, and the `fontMap` parameter set to `nullptr`.

The `compiledText` parameter should be a pointer to a `CompiledText` structure returned by a preceding call to the `CompileStringEx()` function. A pointer to a `GlyphRange` structure may be passed to the

`glyphRange` parameter to specify that only a subset of glyphs are to be processed. The value of the `fontCount` parameter and the entries of the array specified by the `fontDesc` parameter must be exactly the same values that were passed to the `fontCount` and `fontDesc` parameters of the `CompileStringEx()` function.

After the first four parameters, the remaining parameters passed to the `TestSlugEx()` function have the same meanings as the parameters with the same names passed to the `TestSlug()` function.

TextureBuffer structure

The `TextureBuffer` structure holds the address, format, size, and current storage state of a curve or band texture.

Fields

Field	Description
`void *textureData`	A pointer to the memory buffer where the texture is stored.
`TextureType textureType`	The format of the texture. See the description for information about possible values.
`Integer2D textureSize`	The *x* and *y* dimensions of the texture, in texels. If the *y* size is greater than one, then the *x* size must be 4096.
`Integer2D writeLocation`	The *x* and *y* coordinates of the first unused texel in the texture.

Description

The `TextureBuffer` structure is used by functions that import icon data or create fills and strokes at run time. The `textureData` and `textureSize` fields specify the location in memory and the dimensions of a curve texture or band texture.

The following values are the formats that can be specified in the `textureType` field.

Value	Description
`kTextureDefault`	The texture uses the default format, which is `kTextureFloat16` for curve textures and `kTextureUint16` for band textures.
`kTextureFloat16`	Each of the four channels of the texture contains a 16-bit float-point value (curve textures only).
`kTextureFloat32`	Each of the four channels of the texture contains a 32-bit float-point value (curve textures only).
`kTextureUint16`	Each of the four channels of the texture contains a 16-bit unsigned integer (band textures only).

The `writeLocation` field holds the coordinates of the first unused texel in the texture in a top-to-bottom, left-to-right order. It should initially be set to (0, 0), and it is updated by the functions that generate texture data so that information about multiple icons or paths and be stored in the same textures maps.

Triangle structure

The `Triangle16` and `Triangle32` structures contains vertex indices for a triangle.

Triangle16 Fields

Field	Description
`uint16 index[3]`	The three 16-bit vertex indices used by the triangle.

Triangle32 Fields

Field	Description
`Uint32 index[3]`	The three 32-bit vertex indices used by the triangle.

Description

Triangles generated by building functions can be stored as `Triangle16` or `Triangle32` structures, which simply hold three vertex indices that are either 16-bit or 32-bit unsigned integers. Triangles are generated by the building functions that write data into buffers specified by one or more `GeometryBuffer` structures. The `indexType` field of the `GeometryBuffer` structure specifies whether 16-bit or 32-bit vertex indices are written.

The `Triangle` type is an alias for the `Triangle16` type.

UpdateLayoutData() function

The `UpdateLayoutData()` function updates the layout state to account for embedded format directives in a given text string.

Prototype

```
void UpdateLayoutData(const LayoutData *layoutData,
                      const char *text,
                      int32 maxLength,
                      LayoutData *exitLayoutData);
```

Parameters

Parameter	Description
`layoutData`	A pointer to a `LayoutData` structure containing the initial text layout state.
`text`	A pointer to a text string. The characters must be encoded as UTF-8. If the `maxLength` parameter is –1, then this string must be null terminated.
`maxLength`	The maximum number of bytes to be processed in the string. If this is set to –1, then the string must be null terminated, and the entire string is processed.
`exitLayoutData`	A pointer to a `LayoutData` structure to which the updated text layout state is written. It is safe to specify the same pointer for both the `layoutData` and `exitLayoutData` parameters. This parameter cannot be `nullptr`.

Description

The `UpdateLayoutData()` function processes any embedded format directives in a given text string and returns a modified `LayoutData` structure.

The initial format state is specified by the `layoutData` parameter, and the updated format state is written to the location pointed to by the `exitLayoutData` parameter. It is safe to specify the same pointer for both the `layoutData` and `exitLayoutData` parameters. The `kLayoutFormatDirectives` bit must be set in the `layoutFlags` field of the `LayoutData` structure specified by the `layoutData` parameter for any changes to be made to the layout state.

The `text` parameter should point to a string of characters encoded as UTF-8. The `maxLength` parameter specifies the maximum number of bytes to be searched for embedded format directives within the string. All directives beginning within this number of bytes are processed, even if the complete directive extends beyond the limit. If a null terminator is encountered before the number of bytes specified by `maxLength` has been processed, then processing stops at the null terminator. If `maxLength` is set to –1, then the string must be null terminated, and the entire string is processed.

Vertex structure

The `Vertex` structure contains the data for each vertex used to render a glyph, icon, fill, or stroke.

Fields

Field	Description
`Vector4D position`	The *x* and *y* components contain the object-space position of the vertex. The *z* and *w* components contain a normal vector used for dynamic dilation.
`Vector4D texcoord`	The *x* and *y* components contain the vertex coordinates in em space. The *z* and *w* components contain internal information about the data location and band counts needed by the Slug shaders.
`Vector4D jacobian`	This attribute contains the 2×2 inverse Jacobian matrix relating em-space derivatives to object-space derivatives.
`Vector4D banding`	This attribute contains the horizontal and vertical band transforms needed by the Slug shaders.
`Color4U color`	This attribute contains the linear RGBA color at the position of the vertex.

Description

The input to the Slug vertex shader consists of a 2D object-space position, a 2D normal vector, 2D texture coordinates corresponding to the vertex coordinates in em space, internal data location information, four floating-point values holding a Jacobian matrix, four floating-point values holding the band transforms, and a 32-bit linear RGBA color. This information is encapsulated in a `Vertex` structure that is consumed directly by the GPU. Vertices are generated by the building functions that write data into buffers specified by one or more `GeometryBuffer` structures.

The fields of the `Vertex` structure correspond to the five interleaved vertex attribute arrays consumed by the vertex shader. The byte offsets, number of components, and component type for each attribute array are listed in the following table. These are the values that should be specified to the rendering API. The number of bytes occupied by a single vertex is 68, equal to `sizeof(Vertex)`, and this should be specified as the vertex array stride.

Array	Offset (bytes)	Components	Component Type
`position`	0	4	32-bit floating-point
`texcoord`	16	4	32-bit floating-point
`jacobian`	32	4	32-bit floating-point
`banding`	48	4	32-bit floating-point
`color`	64	4	8-bit unsigned integer

6

Format Directives

Slug has the ability to recognize special format directives embedded in the text strings that it processes. These directives allow properties of the text, such as the size, color, or underlining state, to be changed dynamically as a line of text is laid out. Embedded format directives are enabled by setting the `kLayoutFormatDirectives` bit in the `layoutFlags` field of the `LayoutData` structure.

An embedded list of format directives begins with the two-character sequence `{#` and ends with the closing brace `}`. Each directive inside these delimiters has the form `directive(`*`params`*`)`, where *`params`* lists the parameters that apply to the directive. Multiple format directives may be issued at once by separating the directives with semicolons. For example, the following pair of directives sets the font size to 24 and the text color to bright red.

```
{#size(24);color(255,0,0)}
```

Boolean, integer, and floating-point values accepted by format directives are parsed using the OpenDDL syntax. (See openddl.org.) Whitespace inside format directives is allowed.

When format directives are enabled, the text making up the directives themselves is treated as if it does not exist for the purposes of rendering and text layout calculations. If format directives are disabled, then any braced sequences of format directives appearing in a text string are rendered normally as if they had no meaning.

Every format directive can masked off by clearing bits in the `formatMask` field of the `LayoutData` structure. This can be useful for allowing only a subset of format directives to be applied in text that was entered by the user. For example, an application may want to allow the color to be changed in a player name entered by the user, but not other properties such as the font size, scale, or skew. When format directives are generally enabled by the `kLayoutFormatDirectives` bit, but a specific type of directive is masked off, that directive is still consumed by text processing functions, but it has no effect.

The following tables list the format directives recognized by Slug.

Directive	Description
font(*value*)	Set the font type to *value*, where *value* is a 32-bit unsigned integer.
size(*value*)	Set the font size to *value* in absolute units, where *value* is a floating-point number. Ignored if *value* is not greater than zero.
stretch(*value*)	Set the text stretch to *value*, where *value* is a floating-point number. Ignored if *value* is not greater than zero.
track(*value*)	Set the text tracking to *value* in em units, where *value* is a floating-point number.
skew(*value*)	Set the text skew to *value*, where *value* is a floating-point number. Positive values skew to the right, and negative values skew to the left.
scale(*x*,*y*)	Set the text scale to (*x*, *y*), where *x* and *y* are floating-point numbers. The *y* component may be omitted, in which case it is set equal to the *x* component. Ignored if either *x* or *y* is not greater than zero.
offset(*x*,*y*)	Set the text offset to (*x*, *y*) in em units, where *x* and *y* are floating-point numbers. Positive values offset right and upward, and negative values offset left and downward.
under(*value*)	Set the underline decoration state to *value*, where value is either true or false. Ignored if the font does not contain underline information.
strike(*value*)	Set the strikethrough decoration state to *value*, where value is either true or false. Ignored if the font does not contain strikethrough information.
script(*value*)	Set the transform-based script state to *value*, where *value* is an integer in the range [−3, 3]. If *value* is 0, then the text scale and text offset states are set to the identity transform. If *value* is positive, then the superscript scale and offset are applied *value* times. If *value* is negative, then the subscript scale and offset are applied *value* times. Ignored if *value* is out of range or the font does not contain transform-based script information.

`left()`	Set the text alignment to left. The new alignment takes effect at the beginning of the next line.
`right()`	Set the text alignment to right. The new alignment takes effect at the beginning of the next line.
`center()`	Set the text alignment to center. The new alignment takes effect at the beginning of the next line.
`just(`*`value`*`)`	Set the full justification state to *`value`*, where *`value`* is either `true` or `false`.
`lead(`*`value`*`)`	Set the leading to *`value`* in em units, where *`value`* is a floating-point number. The new leading takes effect at the beginning of the next line.
`pspace(`*`value`*`)`	Set the paragraph spacing to *`value`* in em units, where *`value`* is a floating-point number.
`margin(`*`left`*`,`*`right`*`)`	Set the left and right paragraph margins to *`left`* and *`right`* in absolute units. The right margin may be omitted, in which case it is set to the same value as the left margin.
`indent(`*`value`*`)`	Set the paragraph first-line indent to *`value`* in absolute units, where *`value`* is a floating-point number.
`tab(`*`value`*`)`	Set the tab size to *`value`* in absolute units, where *`value`* is a floating-point number. Ignored if *`value`* is not greater than zero.
`kern(`*`value`*`)`	Set the kerning state to *`value`*, where value is either `true` or `false`.
`mark(`*`value`*`)`	Set the combining mark positioning state to *`value`*, where *`value`* is either `true` or `false`.
`decomp(`*`value`*`)`	Set the decompose state to *`value`*, where *`value`* is either `true` or `false`.
`seq(`*`value`*`)`	Set the sequence replacement state to *`value`*, where *`value`* is either `true` or `false`.
`alt(`*`value`*`)`	Set the alternate substitution state to *`value`*, where *`value`* is either `true` or `false`.
`lay(`*`value`*`)`	Set the color layer state to *`value`*, where *`value`* is either `true` or `false`.

`laymul(value)`	Set the layer multiplied by text color state to *value*, where *value* is either `true` or `false`.
`grid(value)`	Set the grid positioning state to *value*, where value is either `true` or `false`.
`color(red,green,blue,alpha)`	Set the primary text color to (*red*, *green*, *blue*, *alpha*), where each component is an integer in the range [0, 255]. The red, green, and blue components are specified in the sRGB color space, and the alpha component is linear. The alpha component may be omitted, in which case it is 255 by default. Ignored if any component is out of range.
`color2(red,green,blue,alpha)`	Set the secondary text color to (*red*, *green*, *blue*, *alpha*) using the same format as the primary color. The secondary text color is used only when gradients are enabled.
`gcoord(y1,y2)`	Set the gradient coordinates to *y1* and *y2*. These are the distances above the baseline at which the gradient is equal to the primary and secondary color, respectively. Negative values are allowed.
`grad(value)`	Set the gradient state to *value*, where *value* is either `true` or `false`.
`effect_color (red,green,blue,alpha)`	Set the primary effect color to (*red*, *green*, *blue*, *alpha*), where each component is an integer in the range [0, 255]. The red, green, and blue components are specified in the sRGB color space, and the alpha component is linear. The alpha component may be omitted, in which case it is 255 by default. Ignored if any component is out of range.
`effect_color2 (red,green,blue,alpha)`	Set the secondary effect color to (*red*, *green*, *blue*, *alpha*) using the same format as the primary color. The secondary effect color is used only when effect gradients are enabled.
`effect_gcoord(y1,y2)`	Set the effect gradient coordinates to *y1* and *y2*. These are the distances above the baseline at which the gradient is equal to the primary and secondary color, respectively. Negative values are allowed.
`effect_grad(value)`	Set the effect gradient state to *value*, where *value* is either `true` or `false`.

`reset()`	Reset all format state to the default values given by the layout data referenced by the `defaultLayoutData` field of the `LayoutData` structure. If the `defaultLayoutData` field is `nullptr`, then use the initial values passed to an API function in the `LayoutData` structure.

When sequence replacement is not disabled by the `kLayoutSequenceDisable` bit in the `layoutFlags` field of the `LayoutData` structure, the following format directives can be used to control what types of sequences are replaced.

Directive	Description
`comp(`*`value`*`)`	Set the glyph composition state to *`value`*, where *`value`* is either `true` or `false`.
`slig(`*`value`*`)`	Set the standard ligatures state to *`value`*, where *`value`* is either `true` or `false`.
`rlig(`*`value`*`)`	Set the required ligatures state to *`value`*, where *`value`* is either `true` or `false`.
`dlig(`*`value`*`)`	Set the discretionary ligatures state to *`value`*, where *`value`* is either `true` or `false`.
`hlig(`*`value`*`)`	Set the historical ligatures state to *`value`*, where *`value`* is either `true` or `false`.
`afrc(`*`value`*`)`	Set the alternative fractions state to *`value`*, where *`value`* is either `true` or `false`.

When alternate substitution is not disabled by the `kLayoutAlternateDisable` bit in the `layoutFlags` field of the `LayoutData` structure, the following format directives can be used to control what types of alternates are substituted.

Directive	Description
`style(`*`value`*`)`	Set the stylistic alternates state to *`value`*, where *`value`* is an integer in the range [0, 20]. If *`value`* is nonzero, stylistic alternates are enabled, and they use the set specified by *`value`*. If *`value`* is zero, stylistic alternates are disabled. Ignored if *`value`* is out of range.
`historical(`*`value`*`)`	Set the historical alternates state to *`value`*, where *`value`* is either `true` or `false`.
`smallcap(`*`value`*`)`	Set the lowercase small caps state to *`value`*, where *`value`* is either `true` or `false`.
`capsmall(`*`value`*`)`	Set the uppercase small caps state to *`value`*, where *`value`* is either `true` or `false`.

`titling(value)`	Set the titling caps state to *value*, where *value* is either `true` or `false`.
`unicase(value)`	Set the unicase state to *value*, where *value* is either `true` or `false`.
`caseform(value)`	Set the case-sensitive forms state to *value*, where *value* is either `true` or `false`.
`slashzero(value)`	Set the slashed zero state to *value*, where *value* is either `true` or `false`.
`hyphenminus(value)`	Set the hyphen minus state to *value*, where *value* is either `true` or `false`.
`frac(value)`	Set the fraction state to *value*, where *value* is either `true` or `false`.
`lining(value)`	Set the lining figures state to *value*, where *value* is either `true` or `false`. If *value* is `true`, then the old-style figures state is disabled.
`oldstyle(value)`	Set the old-style figures state to *value*, where *value* is either `true` or `false`. If *value* is `true`, then the lining figures state is disabled.
`tabfig(value)`	Set the tabular figures state to *value*, where *value* is either `true` or `false`. If *value* is `true`, then the proportional figures state is disabled.
`propfig(value)`	Set the proportional figures state to *value*, where *value* is either `true` or `false`. If *value* is `true`, then the tabular figures state is disabled.
`sub(value)`	Set the subscript state to *value*, where *value* is either `true` or `false`. If *value* is `true`, then the superscript, scientific inferiors, and ordinals states are disabled.
`sup(value)`	Set the superscript state to *value*, where *value* is either `true` or `false`. If *value* is `true`, then the subscript, scientific inferiors, and ordinals states are disabled.
`inf(value)`	Set the scientific inferiors state to *value*, where *value* is either `true` or `false`. If *value* is `true`, then the subscript, superscript, and ordinals states are disabled.
`ord(value)`	Set the ordinals state to *value*, where *value* is either `true` or `false`. If *value* is `true`, then the subscript, superscript, and scientific inferiors states are disabled.

7

Font Conversion

Slug includes a command-line tool called `slugfont` that reads files in both the TrueType and PostScript flavors of the OpenType format and converts them to the Slug font format. Every font to be used with Slug must first go through this conversion process.

The syntax for the `slugfont` tool is as follows:

```
slugfont inputfile -o outputfile options
```

The *`inputfile`* must be an OpenType font with the `.ttf` or `.otf` extension, a font collection with the `.ttc` or `.otc` extension, or an OpenVEX file with the `.ovex` extension. The *`outputfile`* would normally have the `.slug` extension.

Options can be specified with the switches listed in the following table.

Switch	Description
`-f` *`index`*	The index of the font to import from a collection in which glyph data is not shared among fonts. The value of *`index`* must be less than the number of fonts contained in the collection. This is zero by default.
`-g` *`ovexfile`*	Generate an OpenVEX file with the name given by *`ovexfile`* containing the entire font that was read from the input file. This is a valid switch for all input formats, including OpenVEX itself.
`-compress` `-no-compress`	Turn output compression on/off. This is on by default.
`-poly` `-no-poly`	Turn bounding polygon generation on/off. This is on by default.

`-contours` `-no-contours`	Turn contour data generation on/off. This is off by default.
`-kern` `-no-kern`	Turn kerning data import on/off. This is on by default.
`-mark` `-no-mark`	Turn combining mark data import on/off. This is on by default.
`-sequence` `-no-sequence`	Turn sequence replacement data import on/off. This is on by default.
`-alternate` `-no-alternate`	Turn alternate substitution data import on/off. This is on by default.
`-layer` `-no-layer`	Turn color layer data import on/off. This is on by default.
`-subscript-xscale` *`scale`*	Set the transform-based subscript *x* scale to *`scale`*. If *`scale`* is not greater than zero, or if this switch is not specified, then the default subscript *x* scale defined by the input font is used.
`-subscript-yscale` *`scale`*	Set the transform-based subscript *y* scale to *`scale`*. If *`scale`* is not greater than zero, or if this switch is not specified, then the default subscript *y* scale defined by the input font is used.
`-subscript-xoffset` *`offset`*	Set the transform-based subscript *x* offset, in em units, to *`offset`*. If this switch is not specified, then the default subscript *x* offset defined by the input font is used.
`-subscript-yoffset` *`offset`*	Set the transform-based subscript *y* offset, in em units, to *`offset`*. If this switch is not specified, then the default subscript *y* offset defined by the input font is used.
`-superscript-xscale` *`scale`*	Set the transform-based superscript *x* scale to *`scale`*. If *`scale`* is not greater than zero, or if this switch is not specified, then the default superscript *x* scale defined by the input font is used.
`-superscript-yscale` *`scale`*	Set the transform-based superscript *y* scale to *`scale`*. If *`scale`* is not greater than zero, or if this switch is not specified, then the default superscript *y* scale defined by the input font is used.

`-superscript-xoffset` *`offset`*	Set the transform-based superscript *x* offset, in em units, to *`offset`*. If this switch is not specified, then the default superscript *x* offset defined by the input font is used.
`-superscript-yoffset` *`offset`*	Set the transform-based superscript *y* offset, in em units, to *`offset`*. If this switch is not specified, then the default superscript *y* offset defined by the input font is used.
`-underline` `-no-underline`	Turn underline glyph synthesis on/off. This is on by default.
`-underline-size` *`size`*	Set the underline stroke size, in em units, to *`size`*. If *`size`* is not greater than zero, or if this switch is not specified, then the default underline size defined by the input font is used.
`-underline-position` *`position`*	Set the em-space *y* position of the bottom of the underline stroke to *`position`*. If this switch is not specified, then the default underline position defined by the input font is used.
`-strikethrough` `-no-strikethrough`	Turn strikethrough glyph synthesis on/off. This is on by default.
`-strikethrough-size` *`size`*	Set the strikethrough stroke size, in em units, to *`size`*. If *`size`* is not greater than zero, or if this switch is not specified, then the default strikethrough size defined by the input font is used.
`-strikethrough-position` *`position`*	Set the em-space *y* position of the bottom of the strikethrough stroke to *`position`*. If this switch is not specified, then the default strikethrough position defined by the input font is used.
`-poly-vertex-count` *`count`*	Set the maximum number of vertices generated for a glyph's bounding polygon to *`count`*. If nonzero, the value of *`count`* is clamped to the range 4–6. If this switch is not specified, then the default value is 4. This is used only if the bounding polygon generation is turned on.
`-poly-edge-factor` *`factor`*	Set the interior edge cost factor for a glyph's bounding polygon to *`factor`*. If this switch is not specified, then the default value is 1.0. This is used only if the bounding polygon generation is turned on.

`-max-band-count` *`count`*	Set the maximum number of horizontal and vertical bands to *`count`*. The value of *`count`* is clamped to the range 1–32. If this switch is not specified, then the default value is 32. Lower values require less storage in the band texture, but higher values produce better rendering performance.
`-outline` *`size`*	Set the outline effect size, in em units, to *`size`*. If *`size`* is greater than zero, then the outline effect is turned on. Otherwise the outline effect is turned off. This is off by default.
`-miter` *`limit`*	Set the miter limit for the outline effect to *`limit`*. The value of *`limit`* is the ratio of the miter length to the outline effect size, and it is clamped to a minimum value of 1.0. This is used only if the outline effect is turned on.
`-bevel`	Set the join style for the outline effect to beveled corners, which is the default style. This is used only if the outline effect is turned on.
`-round`	Set the join style for the outline effect to rounded corners. This is used only if the outline effect is turned on.

Character Ranges

Many fonts contain glyphs for far more characters than are needed by an application, and including them all can waste space. Ranges of Unicode characters can be included or excluded from a font during the import process by using the following switches:

```
-range(begin,end)
-no-range(begin,end)
```

The values of *`begin`* and *`end`* are integers between `0` and `0x10FFFF`. No whitespace is allowed. Each value may be expressed in decimal, hex, octal, or binary. The exact syntax is defined by the rules for integers in the OpenDDL language. (See openddl.org.)

All of the `-range` switches are processed before all of the `-no-range` switches, regardless of their order on the command line. If no `-range` switches are specified, then the following ranges are included by default:

```
(0x000020,0x00007E)
(0x0000A0,0x00D7FF)
(0x00F900,0x00FFFD)
(0x010000,0x10FFFF)
```

This includes all characters currently defined by Unicode. The excluded ranges correspond to control characters and UTF-16 surrogate pairs that do not have associated glyphs.

If at least one `-range` switch is specified, then the included ranges are all of those specified by the `-range` switches plus the ranges `(0x00020,0x0007E)`, which is the Basic Latin block, and `(0x000A0,0x000FF)`, which is the Latin-1 Supplement block.

After all of the included characters have been established, the `-no-range` switches are processed to remove unwanted character ranges.

As an example, the following command generates a font containing the Latin Extended-A block, the Latin Extended-B block, and the Combining Diacritical Marks block in addition to the default Basic Latin and Latin-1 Supplement blocks.

```
slugfont arial.ttf -o arial.slug -range(0x100,0x24F) -range(0x300,0x36F)
```

A range may include characters that don't have glyphs in the font. Those characters are simply skipped.

Glyph Processing

As a glyph is imported, quadratic Bézier curves are generated for each contour defining the shape of the glyph. If a contour has fewer than three control points, then it is deleted. If a quadratic curve is degenerate because its endpoints are equal, then it is deleted. Each cubic curve belonging to a PostScript outline is approximated by one or more quadratic curves that preserve positions and tangents at the original endpoints and at critical points that occur inside the curve.

Color Layers

The `slugfont` tool recognizes multicolor glyphs that use the layered vector graphics mechanism implemented by the `'COLR'` and `'CPAL'` tables in the OpenType format. If color layer data import is enabled (which is the default), then color layers defined by these tables for specific glyphs in the font are imported in addition to an ordinary monochrome glyph. (Note, however, that some fonts do not include monochrome versions of multicolor glyphs.) Glyphs having multiple color layers are rendered as multicolor glyphs unless the `kLayoutLayerDisable` flag is specified in the `layoutFlags` field of the `LayoutData` structure.

Some fonts use entirely different formats and methods to define multicolor glyphs, but these are not supported by Slug. In particular, fonts containing pixel images defined by the `'CBDT'` or `'SBIX'` tables cannot be used with Slug because they are not based on vector graphics.

Glyph Contours

By default, a `.slug` file does not contain enough information to retrieve the contours of a glyph despite the fact that the curve texture contains all of the Bézier curves. The data is organized in such a way that it is most efficient to access Bézier curves in thin horizontal and vertical bands inside the Slug pixel shaders. If the `-contours` switch is specified on the `slugfont` command line, then additional

information about glyph contours is included in the output `.slug` file, and this allows the `GetGlyphContourData()` function to extract the contours for any particular glyph.

Cap Height and Ex Height

When converted to the Slug format, all fonts contain cap and ex height information that can be accessed by passing the key `kFontKeyHeight` to the `GetFontKeyData()` function. Every TrueType font is supposed to contain this information, but if the original font does not for some reason, or if the height values are zero, then it is inferred from the heights of the uppercase letter H and the lowercase letter x, as recommended by the OpenType specification.

Subscripts and Superscripts

Every TrueType font is supposed to contain information about the preferred scale and offset for transform-based subscripts and superscripts. By default, these values are imported to the Slug format unchanged, but they can be individually overridden by using the appropriate command-line switches. The values that are ultimately used during the import process can be retrieved from the Slug font by calling the `GetFontKeyData()` function with the keys `kFontKeySubscript` and `kFontKeySuperscript`. Because the amount of additional data created for subscript and superscript transforms is very small, there is ordinarily no reason to disable it.

If the original font does not contain subscript and superscript transform data for some reason, then the x and y scales for both are set to 0.65, and the x offset for both is set to zero. The y offset for subscripts is set to −0.15 em, and the y offset for superscripts is set to 0.45 em.

Decorations

When underline and strikethrough are enabled (which is the default), a small amount of extra data is added to the curve and band texture data for the font, and it is used to render the strokes associated with these decorations. The exact increase in the size of the uncompressed texture data is 104 bytes per decoration.

Every TrueType font is supposed to contain information about the preferred position and size of the underline and strikethrough strokes. By default, these values are imported to the Slug format unchanged, but they can be individually overridden by using the appropriate command-line switches. The values that are ultimately used during the import process can be retrieved from the Slug font by calling the `GetFontKeyData()` function with the keys `kFontKeyUnderline` and `kFontKeyStrikethrough`. Because the amount of additional data created for underline and strikethrough decorations is very small, there is ordinarily no reason to disable them.

If the original font does not contain underline and strikethrough position and size information for some reason, then the size of both is set to 0.05 em, the position of the underline is set to −0.1 em, and the position of the strikethrough is set to 0.1746 em, as recommended by the OpenType specification.

Bounding Polygons

When bounding polygon generation is enabled, the `-poly-vertex-count` and `-poly-edge-factor` switches can be used to control the number of vertices a polygon can have and the cost of interior edges.

The maximum number of polygon vertices can be 4, 5, or 6. Polygons with higher numbers of vertices require significantly more computation during the import process, but they produce tighter boundaries that can lead to better rendering performance for the font. If zero is specified as the maximum number of vertices, then it is as if the `-no-poly` switch had been specified.

The edge factor affects the calculation that determines when a polygon with fewer vertices is preferred over another polygon with more vertices when considering the cost of interior edges with the polygon's optimal triangulation. A larger edge factor means that a polygon with more vertices would need to have a smaller overall area in order to be chosen over a polygon with fewer vertices. The edge factor can be zero, and this means that the polygon with the smallest area is chosen without regard for the cost of interior edges.

For more information about bounding polygons, see Section 4.14.

Outline Effect

If the `-outline` option is specified on the command line, then two sets of geometric data are generated for each glyph, one having the ordinary shape defined by the font, and the other having an expanded outline. This data is necessary in order to render the glyph outline effect, but it roughly doubles the file size of a font.

The `-miter`, `-round`, and `-bevel` switches control how geometry is generated at corners in the expanded outline. The value specified for the miter limit is the ratio of the miter length to the outline effect size, and it must be at least 1.0. When the miter limit is exceeded where two Bézier curves meet in the original glyph contours, either a rounded corner or beveled corner is generated, depending on which option was selected on the command line. The values that are used during the import process can be retrieved from the Slug font by calling the `GetFontKeyData()` function with the keys `kFontKeyOutline`.

Note that expanded outlines are very sensitive to errors in the glyph contours appearing in low-quality fonts. If artifacts are visible when the outline effect is rendered, the original font should be checked for problems such as tiny loops or cusps.

Font Collections

If the input file is a font collection, then there are two possible situations. First, the collection may contain multiple fonts that all share a single set of glyph outline curves. In this case, all of the fonts in the collection are imported, and the output `.slug` file contains multiple `FontHeader` structures that each reference glyphs with curves contained within common texture data. Second, the collection may contain multiple independent fonts that each have separate tables of glyph outline curves. In this case, only one font is imported, and which one is determined by the `-f` option on the command line.

8

Album Creation

Slug includes a command-line tool called `slugicon` that reads files in the Scalable Vector Graphics (SVG) and Open Vector Graphics Exchange (OpenVEX) formats and converts them to the Slug album format. Icons and pictures that are to be stored in a resource and later built with the `BuildIcon()` or `BuildPicture()` functions must first go through this conversion process. Icons can also be created at run time using the `ImportIconData()` and `ImportMulticolorIconData()` functions without the use of the `slugicon` tool. Any graphics created at run time with the `CreateFill()` and `CreateStroke()` functions also do not make use of the `slugicon` tool.

The syntax for the `slugicon` tool is as follows:

```
slugicon inputfile -o outputfile options
```

The *`inputfile`* must be an SVG file with the `.svg` extension or an OpenVEX file with the `.ovex` extension. The *`outputfile`* would normally have the `.slug` extension.

Options can be specified with the switches listed in the following table.

Switch	Description
`-pict`	Create a picture for each layer in the input file, where each picture is composed of separate icons generated for each geometry in the corresponding layer. If this switch is not specified, then a single icon is created for each layer.
`-float`	For increased precision, use a 32-bit floating-point format for the curve texture instead of the default 16-bit format. This switch is ignored if the `-pict` switch is not also specified.
`-no-compress`	Turn output compression off.
`-g` *`ovexfile`*	Generate an OpenVEX file with the name given by *`ovexfile`* containing the geometry that was read from the input file. This is a valid switch for all input formats, including OpenVEX itself.

`-poly-vertex-count` *`count`*	Set the maximum number of vertices generated for an icon's bounding polygon to *`count`*. If nonzero, the value of *`count`* is clamped to the range 4–6. If this switch is not specified, then the default value is 4.
`-poly-edge-factor` *`factor`*	Set the interior edge cost factor for a icon's bounding polygon to *`factor`*. If this switch is not specified, then the default value is 1.0.
`-max-band-count` *`count`*	Set the maximum number of horizontal and vertical bands to *`count`*. The value of *`count`* is clamped to the range 1–32. If this switch is not specified, then the default value is 32. Lower values require less storage in the band texture, but higher values produce better rendering performance.

The `slugicon` tool can interpret the following subset of the full SVG 1.1 format:

- The `<line>` element. This always produces a stroke but no fill.
- The `<rect>` element. Ordinary rectangles and rectangles with rounded corners are supported.
- The `<circle>` and `<ellipse>` elements.
- The `<polygon>` and `<polyline>` elements. The only difference between these is that the stroke for a `<polygon>` element is always closed, and the stroke for a `<polyline>` element is never closed.
- The `<path>` element. Lines, quadratic curves, and cubic curves are supported. In the SVG path syntax, the M, L, H, V, Q, T, C, S, and Z commands are supported. Elliptical arcs are not supported. A path may have multiple subpaths.
- The `<g>` element. Groups generally have no affect on the final icon geometry. If a `<g>` element has an `id` attribute whose value begins with the string "layer" (not case sensitive), then it is interpreted as a layer instead of a plain group.
- The `<linearGradient>`, `<radialGradient>`, and `<stop>` elements. Only the first and last stops in a gradient are supported. Any intermediate stops are ignored.
- The following styling properties: `transform`, `fill`, `fill-opacity`, `stroke`, `stroke-opacity`, `stroke-width`, `stroke-linecap`, `stroke-linejoin`, `stroke-miterlimit`, `stroke-dasharray`, `stroke-dashoffset`, and `opacity`.
- Properties specified in presentation attributes such as `fill="#00CCFF"`.
- Properties specified in style attributes such as `style="fill:#00CCFF;"`.
- Butt, square, and round stroke caps. Bevel and round stroke joins. Arbitrary dashing.

The `x` , `y`, `width`, and `height` attributes of the root `<svg>` element determine the bounds of the drawing canvas. When icons are converted to the `.slug` format, they are scaled so that the canvas fits into the unit square between zero and one while maintaining the aspect ratio.

If icons are being generated (because the `-pict` switch was not specified), then each layer in the input file becomes a single icon in the `.slug` file. All of the geometry elements in each layer are combined into a single set of curves. If every geometry belonging to a layer has the same fill color, regardless of what that color actually is, then the corresponding icon is monochrome. Otherwise, the corresponding icon is multicolor. The number of color layers in the multicolor icon is determined by how many times the fill color changes as geometries are processed in back-to-front order. For best performance, geometries having identical fill colors should be arranged consecutively in the stacking order to minimize the number of color layers.

If pictures are being generated, then each layer in the input file becomes a single picture in the `.slug` file. Each geometry element in a layer becomes an individual icon that is scaled so that its bounding box fits in the unit square between zero and one. When a picture is built for rendering, the icons composing it are transformed so that they appear at their original positions and sizes, and they are given their original fill colors.

By default, a bounding polygon is generated for each individual icon and for each icon composing a picture. As with fonts, the `-poly-vertex-count` and `-poly-edge-factor` switches can be used to control the number of vertices a polygon can have and the cost of interior edges, and they have the same effect as they do for fonts. If zero is specified for the maximum number of polygon vertices, then polygons are not generated, and icons are always rendered as quads. See the Bounding Polygons section in Chapter 7 for details.

A

Release Notes

Slug 7.0

The following notes summarize the changes that were made in Slug version 7.0.

- The font and album file formats have been updated in this version to support new features and to allow some existing features to be implemented more cleanly. All `.slug` files, whether the resources they contain are fonts or icon/picture albums, now begin with a common header described by the `SlugFileHeader` structure. Information about the curve and band textures is stored in this header, so it's now possible to extract the texture data without knowing which type of resource a file contains.

- The format of the band texture has been changed so it now uses two 16-bit unsigned integer channels instead of four. The second pair of values was needed only for the symmetric bands optimization, which has been removed from the library. Band textures now occupy half the amount of memory that they needed in previous versions of Slug.

- The library now supports compiled text objects of different sizes. The `CompiledText` structure is now a header for a `CompiledStorage` structure that contains the maximum storage space for a compiled string of text. The `CompileString()` function stores a compiled string in a `CompiledStorage` object and returns a pointer to the `CompiledText` header. The `MakeCompactCompiledText()` function takes an existing compiled text object and copies its contents into a compact buffer allocated by the application for long-term storage. Library functions that accept a pointer to a compiled text string can accept pointers to either the full-size `CompiledStorage` object or a compact storage object.

- The multi-line layout functions in the library now support soft hyphens specified in a text string with Unicode value U+00AD. Soft hyphens can be inserted into words to indicate optional line-breaking locations, and they are enabled by specifying the `kLayoutSoftHyphen` flag in the `layoutFlags` field of the `LayoutData` structure. A soft hyphen is rendered only if it is the final character on a line of text, and it is not displayed nor does it contribute to line length otherwise.

- The library now supports a form of vertical glyph layout, primarily for Japanese writing. When the `kLayoutVerticalRotation` flag is set in the `layoutFlags` field of the `LayoutData` structure, glyphs corresponding to characters designated by the Unicode standard as upright in vertical text are rotated 90 degrees counterclockwise. These glyphs are shifted to account for their vertical origins (derived from bounding box height and top side bearing), and their spacing is determined by advance height instead of advance width. Vertical alternates are also substituted as appropriate for those glyphs that

have them. In vertical layout mode, text is still laid out horizontally, and the application is responsible for transforming the final set of vertices as a whole with a 90-degree clockwise rotation.

- For language systems that require it, glyphs can now be automatically decomposed into multiple components that are later combined with other glyphs independently. This applies specifically to the Sara Am glyph in the Thai writing system and is necessary for correct mark layout.
- The sequence replacement functionality previously used for basic glyph composition and ligature substitution has been greatly expanded. The library now recognizes backtrack and lookahead contexts that can be specified for conditional sequence matching, and the actions that can be taken once a match is found have been generalized. In this version of Slug, functionality exists for performing one-to-one glyph substitution and ligature substitution in the context of specific sets of surrounding glyphs. This provides support for many new glyph replacement rules that some fonts contain to support specific writing systems.
- The `PlaceholderData` structure contains a new `glyphNumber` field that holds the number of glyphs preceding the location of the placeholder character in the original string.
- The `LocationData` structure contains a new `dualCaretOffset` field that holds a horizontal delta between the primary caret position and a secondary caret position in bidirectional text. There can be two separate caret positions when the insertion point falls on the boundary between runs of text having opposite writing directions. The two caret positions correspond to where new characters would be inserted depending on whether those characters belonged to left-to-right or right-to-left writing systems.
- The font conversion tool can now optionally include extra information in a `.slug` file that allows glyph contours to be extracted by an application at run time. This is enabled by specifying `-contours` on the command line for the `slugfont` tool. At run time, the Bézier curves making up the contours of a glyph can be retrieved by calling the new `GetGlyphContourCurveCount()` and `GetGlyphContour-Data()` functions.
- A new `kStrokeContours` flag can be specified in the `strokeFlags` parameter of the `CountStroke()` and `CreateStroke()` functions. This flag allows the set of Bézier curves to be composed of multiple closed contours, which is exactly what is returned by the `GetGlyphContourData()` function.

Slug 6.5

The following notes summarize the changes that were made in Slug version 6.5.

- The existing `TextWorkspace` structure has been renamed to `CompiledText`, and its fields have been documented as part of the official API. Most of the information in this structure is organized into arrays of `CompiledCharacter` and `CompiledGlyph` structures.
- The `GetCharacterData()` function and `CharacterData` structure have been removed from the library because they are unnecessary once the data in the `CompiledText` structure is exposed.

- The existing `GetUnicodeCharacterFlags()` function has been documented as part of the official API. This function returns flags corresponding to various Unicode properties for a character code. A new flag indicating whether a character is a combining mark has also been added.
- The `LocateSlug()` function has been added to the library. This function determines caret positioning information for specific byte locations within a text string.
- A new command line switch called `no-compress` has been added to the `slugfont` and `slugicon` tools to disable compression of the curve and band textures.
- A mechanism for injecting a root signature into DX12 shaders has been added for the Xbox platform. The identifier `SLUG_SIG` can be defined as the quoted string containing the root signature.

Slug 6.4

The following notes summarize the changes that were made in Slug version 6.4.

- To eliminate some redundant text processing work that occurs with typical usage of the library, a new set of API functions has been added that operate on a precompiled structure containing character, glyph, and layout information instead of the original text string. The `CompileString()` and `CompileStringEx()` functions generate this information and store it in a `TextWorkspace` structure. That workspace can then be repeatedly passed to multiple API functions such as `CountSlug()` and `BuildSlug()` without having to process the original string again.
- It is now possible to insert placeholders for externally rendered graphics into text strings. The `placeholderBase` and `placeholderCount` fields of the `LayoutData` structure define a range of Unicode values that are interpreted a placeholders, and the `placeholderWidthArray` field points to an array of widths that tell Slug how much space to reserve for each type of placeholder. Functions such as `BuildSlug()` and `LayoutSlug()` can then return information about the positions where placeholders occur when text is laid out.
- When bidirectional text layout is enabled, Unicode characters with the mirrored property are now replaced with their mirrored counterparts when they occur inside a right-to-left run of text.
- The `kLayoutNonlinearColor` flag can now be specified in the `layoutFlags` field of the `LayoutData` structure. This flag prevents color values from being linearized in the vertex data so they are passed through in nonlinear gamma space.
- The `BuildTruncatableSlug()` function has been extended to support several new optional features.
- The symmetric bands optimization has been deprecated and should not be used. The `kRenderSymmetricBands` flag should not be passed to the `GetShaderIndices()` function, and it should not be set in the `renderFlags` field of the `LayoutData` structure. The removal of this optimization will allow the size of the data stored in the band texture to be cut in half in future versions of Slug, and this produces a small speed improvement for most fonts due to better texture caching on the GPU.

Slug 6.3

The following notes summarize the changes that were made in Slug version 6.3.

- The `kLayoutWrapDisable` flag can now be specified in the `layoutFlags` field of the `LayoutData` structure. This flag prevents lines from being broken by the `BreakSlug()` and `BreakMultiLineText()` functions when the maximum span is exceeded, meaning that lines are broken only at hard break characters.
- The `kLayoutLayerTextColor` flag can now be specified in the `layoutFlags` field of the `LayoutData` structure. This flag causes the colors of all layers in a multicolor glyph to be multiplied by the current text color, which can include a gradient. This flag can be controlled by the `laymul` format directive.
- The `geometryBuffer` parameter passed to the `BuildSlug()`, `BuildMultiLineText()`, `AssembleSlug()`, and `BuildTruncatableSlug()` functions can now be `nullptr`. In this case, no vertex and triangle geometry is generated, but all other information returned by these functions is still valid.
- Triangles written by any of the geometry building functions can now contain 32-bit vertex indices. A new field called `indexType` has been added to the `GeometryBuffer` structure to indicate whether 16-bit or 32-bit indices should be generated. (This field is initialized for 16-bit indices by default, so existing code does not need to be updated.)
- Miter, bevel, and round joins inside stroked paths now generate fewer vertices and triangles in cases where the two curves joined together are straight lines.

Slug 6.2

The following notes summarize the changes that were made in Slug version 6.2.

- The `kLayoutFullJustification` flag can now be specified in the `layoutFlags` field of the `LayoutData` structure to enable full justification in multi-line text. This is independent of the value specified by the `textAlignment` field, which still applies to the last line in each paragraph.
- The `graphicsFlags` field of the `GraphicData` structure now contains the `kIconMulticolor` flag to indicate when an icon can be rendered in multicolor mode.
- The `tabRound` field has been added to the `LayoutData` structure. This is an em-space distance added to the current drawing position when determining where the next tab stop should be. This is set to 0.0 by default to maintain backward compatibility.
- The `defaultLayoutData` field has been added to the `LayoutData` structure. If this is not `nullptr`, then it provides the values used by the `reset()` format directive. If this is `nullptr`, then the values used by the `reset()` format directive are the values initially passed to an API function through the `LayoutData` structure.

Slug 6.1

The following notes summarize the changes that were made in Slug version 6.1.

- Slug can now render filled paths with linear and radial gradients. The `FillData` structure passed to the `CreateFill()` function can specify gradient information. Gradients in SVG files are recognized and imported by the `slugicon` tool. A `Gradient` structure has been added to the OpenVEX format.
- Either of the `curveTexture` and `bandTexture` parameters passed to the `ExtractFontTextures()` and `ExtractAlbumTextures()` functions can now be `nullptr`. This makes it possible to extract only one of the curve or band textures at a time.
- The `missingGlyphIndex` field has been added to the `LayoutData` structure. This field specifies the index of the glyph to draw whenever a character is missing in a font.

Slug 6.0

The following notes summarize the changes that were made in Slug version 6.0.

- Slug now supports arbitrary stroked paths with optional dashing and standard cap and join styles. Stroked paths can appear in a picture that was imported to an album resource, or they can be created at run time with the `CreateStroke()` function. A new render flag, `kRenderStrokes`, has been defined and must be used to select the proper shader whenever stroked paths are rendered.
- Two new functions called `CreateFill()` and `CreateStroke()` provide the ability to generate the data for filled and stroked paths at run time. These functions generate data that is stored in the curve and band textures as well as vertex and triangle data. They can be used to incrementally build arbitrarily complex vector graphics at run time that can then be rendered in a single draw call.
- The extended functions that work with multiple fonts now accept an array of `FontDesc` structures instead of an array of pointers to `FontHeader` structures. Each `FontDesc` structure contains a pointer to a `FontHeader` structure and also allows a per-font scale and offset to be specified. This is useful for adjusting the sizes of different fonts so that they appear to have the same visual size, which is sometimes necessary because different fonts can have different capital heights within the em square. Adapter functions are included so that the previous API can still be used.
- Several data structures have been renamed. For most cases, this was done to reflect the fact that they are no longer used exclusively for glyphs, but now also for general geometry. The old names are still available as aliases. The `GlyphVertex` and `GlyphTriangle` structures have been renamed to `Vertex` and `Triangle`. The `GlyphBuffer` structure has been renamed to `GeometryBuffer`. The `TextureState` structure has been renamed to `TextureBuffer`. The `Workspace` structure has been renamed to `TextWorkspace`, and the `ImportWorkspace` structure has been renamed to `FillWorkspace`.
- The run-time library can now share contour data among glyphs that are identical except for an (x, y) offset in em space. This allows the font importer to recognize cases in which one glyph is identical to another glyph with an offset and reduce the size of the data. This typically happens in TrueType fonts

that include both tabular and proportional figures, numerators and denominators, subscripts and superscripts, etc. Some fonts will decrease in size significantly when re-imported.

- The font importer can now synthesize tabular figures if they are not included in the original font. Each tabular figure has an advance width equal to the largest of the proportional figures, and the bounding box of each glyph is centered within that width.
- Alternative fraction sequences have been added to the types of sequences that are supported by Slug. This corresponds to the `'afrc'` feature in OpenType.
- The `BreakSlug()` and `BreakMultiLineText()` functions can now combine two different consecutive hard break characters into a single line break. This is useful for text containing a mixture of CR, LF, and CRLF characters, each of which should cause only one break.
- If a font contains caret positioning data for ligatures in its `GDEF` table, it is now recognized during the font import process. If that information is not available, then it is automatically generated based on the proportional bounding box widths of the characters that are replaced by each ligature. The `TestSlug()` and `TestSlugEx()` functions use the caret positioning data to place the caret between characters inside a single ligature glyph.
- All text layout functions now support basic tab spacing. When the `kLayoutTabSpacing` flag is specified in the `layoutFlags` field of the `LayoutData` structure, tab characters in the text advance the drawing position to the next multiple of the spacing specified in the `tabSize` field.
- The `renderFlags` parameter was removed from the `BuildPicture()` function.

Slug 5.5

The following notes summarize the changes that were made in Slug version 5.5.

- Support for font collections has been added to the library and import tools. A new `GetFontCount()` function has been added to the library, and the `GetFontHeader()` function has been updated to accept an optional font index.
- An `exitPosition` parameter has been added to the `BuildSlug()` and `BuildSlugEx()` functions. This parameter is optional, and it has the same meaning as the existing `exitPosition` parameter for the `LayoutSlug()` function.
- The `CalculateTextLength()` and `CalculateTextLengthEx()` functions have been modified to accept a maximum string length. In addition to the maximum span, these functions can now calculate text lengths for a second truncation span. This is convenient for determining where to cut off a string that won't fit in a given span so that a suffix such as an ellipses can be added.
- New high-level functions called `BuildTruncatableSlug()` and `BuildTruncatableSlugEx()` have been added to the library. These functions automatically determine whether a text string needs to be truncated in order to fit inside a maximum span, and if so, append an arbitrary suffix string without overflowing.

- New `TestSlug()` and `TestSlugEx()` functions were added to the API. These functions determine how a test position interacts with a single line of text.
- The `ImportMulticolorIconData()` function has been changed so that it accepts an array of curve counts and an array of pointers to separate arrays containing the curves used by each color layer.
- The `slugfont` tool now remaps the (very old) symbol encoding to the equivalent Unicode characters and aliases the characters in the range U+0020 to U+007E.
- The `slugfont` tool can now import glyphs having Unicode values in planes 0x03 to 0x10, which includes the private use area in planes 0x0F and 0x10.
- The OpenVGX (Open Vector Graphics Exchange) format has been renamed OpenVEX to avoid confusion with an independent and unrelated API.

Slug 5.1

The following notes summarize the changes that were made in Slug version 5.1.

- When drop shadow and outline effects are enabled, the geometry they generate is no longer interleaved. All effect geometry precedes all ordinary geometry in the vertex buffer. This is necessary for correct cursive joining with an effect enabled. This change is completely internal, and there are no associated API changes.
- A new command-line switch called `-max-band-count` is available in the `slugfont` and `slugicon` tools. It provides control over the maximum number of horizontal and vertical bands that can be generated for each glyph or icon. A lower number of bands requires less storage space in the band texture, but a higher number of bands produces better performance.
- The maximum number of bands can also be controlled for icons imported at run-time through the new `maxBandCount` parameter added to the `ImportIconData()` and `ImportMulticolorIconData()` functions.
- The `slugfont` tool now calculates font-wide bounding boxes representing the maximum extents of all glyphs imported for a font. One bounding box is calculated for all ordinary base glyphs, and a separate bounding box is calculated for all glyphs that are combining marks. This information is stored in a `FontBoundingBoxData` structure, and it can be retrieved by an application by calling the `GetFontKeyData()` function with the key `kFontKeyBoundingBox`.

Slug 5.0

The following notes summarize the changes that were made in Slug version 5.0.

- Some of the source code files have been reorganized and renamed. Files belonging exclusively to Slug are now prefixed with `SL`, and they are stored in the `SlugCode` directory. Files containing more generic code shared by multiple technologies continue to be prefixed with `TS`, and they are stored in the `TerathonCode` directory.

- The term "pixel shader" has been replaced with the term "fragment shader" throughout Slug. This makes it consistent with the terminology used by the majority of the supported rendering libraries, and it distinguishes the use of "fragment" from the use of "pixel" in other contexts.
- Glyphs containing multiple color layers are no longer rendered with a special version of the Slug fragment shader that loops over the color layers at every pixel. Instead, multicolor glyphs are rendered as multiple glyphs stacked on top of each other with separate vertex and triangle geometry for each layer. This has two big advantages: First, it means that multicolor glyphs can be rendered alongside ordinary monochrome glyphs without any penalty for glyphs not having color layers. Second, because many layers tend to cover areas much smaller than the whole glyph, rendering performance for color emoji is significantly better. Depending on the font and particular emoji, rendering times can be reduced by 15–20% in some cases and more than 50% in others.
- Any fonts containing multicolor glyphs (emoji) need to be reimported through the `slugfont` tool before they can be rendered in this version.
- The `GetShaderIndices()` function has been changed so that it no longer takes a pointer to a font header. The index of the fragment shader no longer depends on single-pass multicolor information being present in the font. When retrieving shader indices for the purposes of rendering text, the value of the `renderFlags` parameter passed to the `GetShaderIndices()` function may no longer contain the `kRenderMulticolor` bit.
- A new indirect font mapping mechanism has been added to Slug that provides a flexible way to utilize multiple font types in the same text string with optional fallbacks for missing glyphs. Font types are identified by application-defined 32-bit codes, and each type can reference multiple source fonts in a master list of fonts specified for the string. All of the library functions that operate on a single "slug" or on multi-line text now have extended versions that perform the same operation with a given font mapping structure.
- The `font()` format directive has been added to allow the font type to be changed anywhere inside a text string. The corresponding font type is stored in the new `fontType` field of the `LayoutData` structure.
- When drop shadow and outline effects are enabled, the geometry they generate is now interleaved with the primary glyph geometry. The counting functions that would have previously returned two vertex counts and two triangle counts when effects are enabled now return only one of each, and the building functions that would have previously written data to two `GeometryBuffer` structures now write to only one.
- A nonzero script level specified in the `LayoutData` structure no longer affects the `textScale` and `textOffset` fields, which can now be used independently of transform-based subscripts and superscripts.
- Some optimizations have been made to make text layout functions faster on the CPU.
- The `slugfont` tool has been modified to handle some unusual not-necessarily-standard-compliant kerning tables found to be used by some fonts in the wild.

Slug 4.2

The following notes summarize the changes that were made in Slug version 4.2.

- The method by which bounding polygons for glyphs and icons are calculated has been changed in this version. Previously, regions of empty space at the corners of the aligned bounding box were trimmed, where possible, to produce a polygon having between 4 and 8 sides. This had the tendency to create sliver triangles, and it did not allow single triangles to be used where the underlying shape was naturally triangular. Now, a more optimal bounding polygon is calculated independently of the bounding box, and it can have between 3 and 6 sides. In the process of choosing the best polygon, the rendering cost of the shortest possible set of interior edges for the polygon's triangulation is taken in account. In general, glyphs and icons using the new bounding polygons render roughly 5–10% faster, and they use an average of 20–25% fewer vertices. See Section 4.14 for more information. Note that fonts and albums generated by previous versions of the `slugfont` and `slugicon` tools will continue to be rendered in the same way as they previously were, with trimmed corners. To take advantage of the new bounding polygons, fonts and albums need to be generated with the tools included with Slug 4.2.

- The `slugfont` and `slugicon` tools have new command-line parameters that control the generation of bounding polygons by specifying the maximum number of vertices and an interior edge cost factor. The `FontPolygonData` structure is now included with an imported font so that the specific settings used during import can be retrieved with the `GetFontKeyData()` function.

- The `ImportIconData()` and `ImportMulticolorIconData()` functions take new optional parameters that control bounding polygon generation in the same way that command-line parameters for the `slugicon` tool do. Since these parameters precede the optional `workspace` parameter for these functions, any existing code that uses the `workspace` parameter will need to be updated.

- The `slugfont` import tool has been updated so it can read CID-keyed type 2 fonts stored in a file with the `.otf` extension.

Slug 4.1

The following notes summarize the changes that were made in Slug version 4.1.

- Some internal changes were made to accommodate the arbitrary per-glyph transforms available in this release and the arbitrary glyph boundary polygons that will be implemented in a future release. It was necessary to modify the vertex format used by the Slug shaders, and the total size of a single vertex has increased by eight bytes. As shown by the `Vertex` data structure, each vertex now has four different four-component floating-point attributes and a single four-component 8-bit unsigned integer attribute. Application code that specifies the format of these attributes to a rendering API must be updated to reflect the new format.

- The `LayoutSlug()` and `LayoutMultiLineText()` functions have been expanded so that, in addition to glyph indices and drawing positions, they can return per-glyph transformation matrices and per-glyph colors. A new function called `AssembleSlug()` has been added to the library, and it generates vertex and triangle data using application-supplied glyph positions, transforms, and colors. This

allows an application to have full control over custom glyph layouts, but still use Slug to apply typographic features and special effects.

Slug 4.0

The following notes summarize the changes that were made in Slug version 4.0.

- The `slugfont` import tool now supports the PostScript flavor of OpenType fonts. These are the fonts that contain embedded Adobe type 2 fonts in a `'CFF '` table and have the `.otf` file extension.
- Icons can now be stored in "album" files, and pictures can be defined by sets of icons. Several new functions and data structures have been added to the library to support album files and pictures.
- Slug now includes an icon import tool called `slugicon` that can read SVG and OpenVEX files and produce album files containing icons or pictures. Support for SVG is limited to a specific subset of functionality, as described in Chapter 8.
- A new rendering option has been added to the Slug fragment shaders that enables a fast path for icons having outlines composed entirely of straight lines. This optimization is typically used to render pictures for which many components have polygonal geometry. This option is selected by including the `kRenderLinearCurves` flag in the `renderFlags` parameter passed to the `GetShaderIndices()` function.
- New paragraph-level attributes are now available when laying out multi-line text. Additional spacing may be inserted between paragraphs, left and right margins can be defined, and the first line of each paragraph can be indented. These attributes are enabled by including the `kLayoutParagraph-Attributes` flag in the `layoutFlags` field of the `LayoutData` structure, and their values are specified in the `paragraphSpacing`, `leftMargin`, `rightMargin`, and `firstLineIndent` fields.
- The `BreakMultiLineText()` function has been changed to properly support paragraph attributes over multiple calls. It now takes a `LineData` parameter instead of a string offset to establish the properties of the preceding line of text, if any.
- The `ImportMulticolorIconData()` function has been changed so that it accepts a single array of Bézier curves, and the curves belonging to each color layer are identified by a starting index within that array and a count of the curves in the layer.
- The `LayoutSlug()` and `LayoutMultiLineText()` functions have been changed so they can return scale information in addition to glyph positions.
- When generating expanded glyph outlines with the `slugfont` tool, the default outline join style is now bevel instead of round to be consistent with the default line join style used by vector graphics standards.

Slug 3.5

The following notes summarize the changes that were made in Slug version 3.5.

- A major new feature called dynamic glyph dilation has been added to the library. The vertex shader now performs a calculation that dilates a glyph's bounding polygon by the optimal amount. This ensures that the smallest possible area is filled by the fragment shader, improving performance in cases where text is dynamically rendered at different scales. It also ensures that bounding polygons are as large as necessary for antialiasing, improving quality in cases where text is rendered at very small sizes. When text is viewed in perspective, the amount of dilation can be different for each vertex of a glyph.
- The `Vertex` structure has been modified to support dynamic glyph dilation. The `position` and `texcoord` fields now have four components each instead of the previous two. This affects the offsets that need to be specified for vertex attribute arrays.
- The `dilationFactor` field of the `LayoutData` structure has been removed because it is no longer necessary.
- The `slugfont` tool has been modified so that kerning information can be imported from both the `GPOS` and `kern` tables for the same font. It was discovered that some fonts were storing data in the `kern` table that was not properly duplicated in the `GPOS` table. In this version, if the `GPOS` table does not contain kerning data for a particular glyph, then any kerning data found in the `kern` table for the same glyph will be imported.
- The `slugfont` tool has been modified to recognize a default glyph classification, undocumented in the OpenType specification, used by some fonts for class-based kerning. If an imported font is kerning in some cases, but not in others where kerning is expected, this update should resolve the problem.

Slug 3.0

The following notes summarize the changes that were made in Slug version 3.0.

- The library is now able to render arbitrary vector-based icons that are specified independently of any font. An icon is created by supplying a set of closed quadratic Bézier curves to the `ImportIconData()` or `ImportMulticolorIconData()` function. The `BuildIcon()` function then generates vertex and triangle data that can be rendered with the same shaders used to render glyphs.
- The `GetShaderIndices()` function has been changed so that the second parameter is a `uint32` containing the render flags instead of a pointer to a `LayoutData` structure. This was done because the function applies to icon rendering as well as font rendering, and icons do not use `LayoutData` structures. Existing code can be updated by replacing the pointer to a `LayoutData` structure with the value of the `renderFlags` member of that same structure.
- The `CountSlug()` and `CountMultiLineText()` functions have been updated to return the number of glyphs that would be generated by a text string. Compatibility is not broken because only the return values of these functions have been changed from `void` to `int32`.
- New `LayoutSlug()` and `LayoutMultiLineText()` functions have been added to the library. They generate glyph index and position arrays for a text string instead of vertex and triangle buffers.

- A `.slug` file may now contain a table that maps glyph indices used by Slug to the glyph indices used in the original font file. This table is generated when the `-glyph-mapping` option is specified during font conversion.
- A new grid positioning mode has been added for text layout. When enabled, advance widths and kerning are ignored, and glyphs are centered on regularly-spaced positions determined only by the tracking value.
- Stronger compression is now applied to the curve and band textures. A new optional parameter has been added to the `ExtractFontTextures()` function to provide the per-thread temporary storage that is now needed for decompression.
- New types of key data containing ascent, descent, and line gap values have been added to every font imported with this version of Slug. There are two variants, as described in the documentation for the `GetFontKeyData()` function.

Slug 2.0

The following notes summarize the changes that were made in Slug version 2.0.

- The `.slug` file format has been updated, and files created with earlier versions need to be created with the `slugfont` utility again to use the new format.
- In OpenGL, the glyph shaders now assume that the curve and band textures have been bound to the `GL_TEXTURE_2D` target instead of the `GL_TEXTURE_RECTANGLE` target.
- Performance has been improved significantly by increasing the maximum number of bands into which each glyph can be divided and making the band scale independent for the horizontal and vertical directions. Rendering time is reduced by approximately 25% for simple fonts and by 50% or more for complex fonts.
- The internal processing of character strings has been changed and now uses temporary storage called a workspace. An application can allocate `Workspace` structures and pass them to the library functions that need them so that Slug functions can be called concurrently from multiple threads. If reentrancy is not required, then the library uses a shared internal workspace.
- Support has been added for languages with right-to-left writing directions, and the library can perform bidirectional text layout. These features are enabled by setting the `kLayoutRightToLeft` and `kLayoutBidirectional` flags in the `layoutFlags` field of the `LayoutData` structure.
- Initial, medial, and final forms are now substituted in cursive languages like Arabic based on the Unicode joining properties of the surrounding characters.
- Object-space scale and offset values have been added to the `LayoutData` structure to provide a way to transform the final vertex positions of each glyph.

- New `renderFlags` and `geometryType` fields have been added to the `LayoutData` structure, and they provide separate fields for new options and some existing options that were previously specified in the `layoutFlags` field.
- The `LayoutData` structure contains a new `dilationFactor` field that provides control over how much glyph bounding boxes are dilated.
- The library now supports alternate substitution for fractions. When enabled, a slash character preceded and followed by one or more numerical digits is transformed into a sequence of numerator alternates, a fraction slash, and a sequence of denominator alternates.
- The glyph shaders have been reorganized in such a way that it's possible for an application to incorporate them into larger shaders. This allows text to be rendered with surface materials in a 3D game scene, for example.
- Glyph shaders are now returned as an array of strings by the `GetVertexShaderSourceCode()` and `GetFragmentShaderSourceCode()` functions based on what rendering features are enabled and which components are needed by the application.
- The glyph shader can now perform adaptive supersampling for high-quality minified rendering. This feature is enabled by setting the `kRenderSupersampling` flag in the `renderFlags` field of the `LayoutData` structure.