Introduction

Text Rendering involves both the handling of Unicode standards regarding character encoding as well as the generation and rendering of the glyphs. The sub-chapters here explain the basic principles.

Character Encodings

character encoding system consists of a code that pairs each character from a given repertoire with something else such as a bit pattern, sequence of natural numbers, octets, or electrical pulses (Source: Wikipedia).

Unicode is the most important standard for character encodings and defines (among other things) the mapping of a character to a Unicode code point (bit pattern):

- Fixed length encodings of code points (ASCII, UCS2, UCS4 etc).
- Variable length encoding of code points (UTF-8, UTF-16, UTF-32)
- Chinese GB18030 is the only relevant other encoding standard

All texts which are rendered within the TextEngine have to be of type TChar*. Text in TChar* have to be encoded in UTF-8. UTF-8 uses a varying number of TChars to encode one glyph. E.g. a UTF-8 encoded string with special characters: "&auml&Auml&ouml&Ouml&uuml&Uuml&szlig&#8364" consists of 8 glyphs but of 17 bytes because several of these special characters are encoded in more than one byte.

The methods FeatStd::String::GetCharCount() and FeatStd::String::GetCodePointCount() can be used to get the numbers as explained in the example above. An instance of FeatStd::String class can be constructed with TChar* by the constructor FeatStd::String::String(const TChar *).

> Char data type must not be used for I18N text and always maps to ASCII.

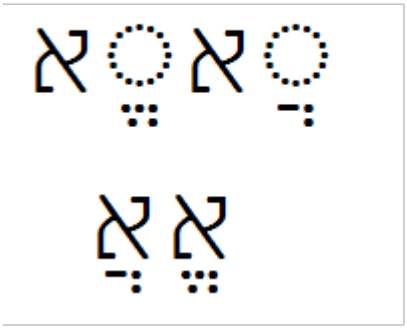
Complex Script Languages

A complex script is a writing system which requires complex transformation of glyphs like substitution and positioning. Examples of such writing systems are arabic, hebrew and thai. Complex transformations include

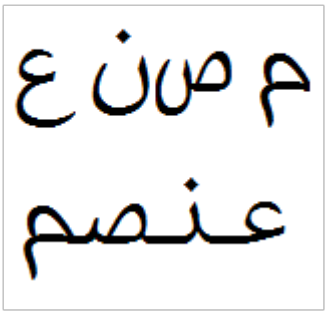
- bi-directional ordering of characters,

- contextual shaping which means that displayed glyphs depend on the surrounding code points in the text - displayed glyphs can have different shapes depending on their position in a word or can be combined to ligatures,
- combining characters where several characters are combined into one shape

Example of hebrew script:



Example of arabic script:



Above the characters as they are written and below as they are displayed.

Text Rendering - Functional Steps

Text Rendering involves the following functional steps:

1. Glyph Rendering
2. Shaping
3. Layouting

1. Glyph Rendering

Takes a single code point and font metrics to render the according glyph. The Freetype font engine for example takes a code point and outputs a 8 bit alpha.

2. Shaping

Shaping is the transformation from logical to display presentation of a text. The Shaping process is responsible for considering complex scripts. The Shaping process is applied on paragraph level.

GPOS/GSUB Tables

OpenType fonts may include GSUB (glyph substitution) and GPOS (glyph positioning) tables.

The GSUB table contains (among other things)

- Ligature substitution (replace multiple code points with single code point)
- Contextual substitution (alternative presentation of the glyph depending on his context) If no GSUB table is available, the standard substitutions defined by Unicode apply (eg. mandatory set of Arabic ligatures).

The GPOS table contains (among other things)

- glyph placement
- adjustment of marks
- kerning
- attachment points for combining characters

Shaping Libraries

- FEAT ComplexScript library does shaping for Arabic language
- FreeBidi - functional equivalent to FEAT ComplexScript
- Harfbuzz - Arabic + many other scripts

A Candera feature define is available in CMake to select between the 3rd party software components *ComplexScriptLib* and *HarfBuzz* for text shaping.

3. Layouting

Layout text that has been shaped. Typical operations:

- Line breaking
- Word breaking
- Text alignment (horizontal and vertical)

Revision #7
Created 2023-02-22 01:47:28 UTC by Tsuyoshi.Kato
Updated 2025-01-10 13:51:36 UTC by Elisabeth Zauner