Demystifying Unicode Text Display: From Unicode Code Points to Positioned Glyphs

Demystifying Unicode Text Display: From Unicode Code Points to Positioned Glyphs

Conference Session Notes - Unicode Technical Workshop 2025
Presenters: Microsoft & Apple Text Layout Teams

Overview

This session explores the complex journey from Unicode characters in a file to the positioned glyphs you see on screen. Understanding this process is crucial for developers working on internationalization, text analysis, localization testing, and anyone dealing with multi-script content.

Why Learn About Text Display?

- **Software Development**: Working on text display software or browsers
- **Writing Systems**: Understanding how different scripts are implemented
- **Unicode Encoding**: Planning to propose new writing system encodings
- **Localization**: Testing scenarios with different languages and scripts
- **Text Analysis**: Understanding the relationship between encoded characters and visual output

Text Display & Fonts

Core Concepts & Terminology

Key Terms

- **Characters** - Abstract units stored in data files
- **Code Points** - Numeric values representing characters in Unicode
- **String** - Sequence of characters
- **Glyphs** - Actual visual shapes rendered on screen

- **Glyph Run** - Sequence of positioned glyphs
- **Font Family** - Set of fonts sharing design traits (e.g., Arial)
- **Font Style** - Specific variant within family (e.g., Arial Bold)

Critical Distinction

- **Character**: Capital letter "A" (abstract concept)
- **Glyph**: The specific visual shape of "A" from a particular font

Basic Text Layout Process

Simple Case: Single Line, Latin Characters

The most basic form of text layout involves:

- Sequence of glyphs arranged on a baseline
- Each glyph positioned adjacent to the previous one
- Left-to-right progression

Font Data Structure

Font files are organized as databases containing:

- **Name Table**: Strings describing font metadata
- **Glyph Table**: Actual glyph outline data
- **Metrics**: Measurements for font and individual glyphs
- **CMap Table**: Character-to-glyph mapping

All data is organized into tables with 4-character mnemonic names.

Character-to-Glyph Mapping

- **CMap Table** provides initial character→glyph mapping
- **Glyph IDs** are arbitrary numbers assigned by font designer
- **Not all characters** may be supported by a given font
- This is called the "nominal mapping" or "default glyph mapping"

Glyph Positioning Basics

Each glyph has:

- **Origin Point**: Where X=0, Y=baseline intersection
- **Left Side Bearing**: Distance from origin to left edge
- **Advance Width**: Distance to move for next glyph position
- **Outline Data**: Control points defining the shape

Layout Process:

1. Align glyph origin with current drawing position
2. Render the glyph
3. Move drawing position by advance width
4. Repeat for next character

Advanced Layout Requirements

Simple character-by-character layout is insufficient for:

1. Kerning

Adjusting spacing between specific letter pairs for better visual balance

- Example: "VA" or "To" - reducing space for optical balance

2. Contextual Positioning

Arabic Script Example:

- Letters change shape based on position in word
- Connecting scripts require precise glyph alignment
- Marks above letters must adjust to letter height

3. Combining Marks

Diacritical Marks:

- Must position accurately relative to base letters
- Avoid collisions with other marks
- Handle complex combinations (multiple accents)

4. Glyph Substitution

Contextual Forms:

- Same character may need different glyphs based on context
- Arabic: initial, medial, final, isolated forms
- Complex scripts require cluster analysis

5. Ligature Substitution

Typographic Ligatures:

- Replace character sequences with single composed glyphs
- Example: "ffi" → single ligature glyph
- Improves readability and aesthetics

6. Language-Specific Variants

Same character may have different appearances in different languages

- Example: Cyrillic letters in Russian vs. Bulgarian

7. Bidirectional Text (BIDI)

Some writing systems require right-to-left text processing

- Hebrew and Arabic written right-to-left
- Mixed direction content requires Unicode Bidirectional Algorithm
- Glyph reordering within clusters may be needed

Implications

- Advanced line layout is required for high quality typography & many scripts
- Complex character-to-glyph associations - no longer one-to-one mapping
- Default glyph metrics alone don't determine final positions
- Additional software logic is required beyond basic font data
- Font-specific details drive advanced layout behavior

OpenType Layout System

Advanced Layout Engine Requirements

- **General Advanced Layout Logic**
- **Script-Specific Behavior Logic**: Based on Unicode character properties
- **Font-Specific Data**: Substitution and positioning rules

OpenType Tables

- **GDEF (Glyph Definition Table)**: Classifies glyphs by type (base, mark, ligature, component)
- **GSUB (Glyph Substitution Table)**: Defines character/glyph substitution rules, handles contextual forms, ligatures, alternates
- **GPOS (Glyph Positioning Table)**: Defines positioning adjustments, handles kerning, mark positioning, cursive attachment

Text Shaping Engines

Platform-specific implementations:

- **CoreText** (macOS)
- **DirectWrite** (Windows)
- **HarfBuzz** (Linux/Cross-platform)

Text Processing Pipeline

1. Run Segmentation

Script Itemization:

- Segment text by Unicode script properties
- Group characters requiring similar processing

BiDi Level Analysis:

- Apply Unicode Bidirectional Algorithm
- Determine text direction runs
- Handle mixed left-to-right and right-to-left content

2. Shaping Process

For each text run:

Canonical Decomposition (UAX #15):

- Normalize character sequences
- Handle composed vs. decomposed forms

Cluster Analysis (UAX #29):

- Identify character clusters that must be processed together
- Critical for complex scripts like Devanagari, Arabic

Glyph Substitution:

- Apply contextual forms
- Process ligatures
- Handle language-specific variants

3. Positioning

- Apply kerning adjustments
- Position combining marks using anchor points
- Handle cursive attachment
- Calculate final glyph positions

Bidirectional Text Processing

Unicode Bidirectional Algorithm

Every character has a **Bidi_Class** property:

- **Strong LTR**: Latin letters (L)
- **Strong RTL**: Arabic, Hebrew letters (R, AL)
- **Neutral**: Punctuation, symbols (neutrally directional)

Processing Steps:

1. Assign embedding levels based on character properties
2. Create level runs of same directionality
3. Reorder glyphs within and between runs
4. Handle neutral characters based on context

Result: Text displays correctly regardless of storage order

Font Fallback

When Fallback Occurs

When primary font lacks required glyphs:

- Individual characters missing
- Entire clusters unsupported
- Language-specific glyph variants needed

Context Considerations

User Preferences:

- Language settings
- Input method indicators
- Markup language tags

Font Matching Criteria:

- **Classification**: serif, sans-serif, cursive, monospace
  - Some classifications are specified by fonts themselves
  - Some are determined by other means (□□)
- **Attributes**: weight, width, italic/oblique
  - Fallback font may not exactly match all attributes
  - Variable fonts can be responsive to some attributes

Available Font Selection:

- Platform-dependent font sets
- Application-specific font lists
- Privacy considerations (web fonts)

Display Emojis

Emoji processing requires:

- Character property analysis
- Known sequence recognition

- Variation selector handling
- Color font format support

Color Font Formats

- **Bitmapped**: sbix, CBDT/CBLC tables
- **Vector**: COLRv0/v1 tables
- **SVG**: SVG table

Not necessarily one glyph per emoji - complex emoji may use multiple glyphs with positioning

Multi-Line Layout

Line Breaking

Uses accumulated glyph width information to:

- Determine text that fits in available width
- Find appropriate break points
- Handle bidirectional content wrapping

Vertical Spacing

Font Metrics:

- **Ascent**: Distance above baseline
- **Descent**: Distance below baseline
- **Line Gap**: Additional spacing between lines

Applications may apply additional line spacing adjustments.

Common Display Problems

1. Invalid Clusters

Causes:

- Incorrect character sequences for script
- Components in wrong order
- Unicode normalization issues

Symptoms:

- Dotted circles indicating invalid combinations
- Missing or misplaced diacritical marks

2. Copy/Paste from PDF Issues

Problem: PDFs store glyph positions, not original text

- Advanced layout information lost
- Character-to-glyph mapping may be irreversible
- Copy/paste produces garbled text

Solution: Ensure PDFs embed proper text extraction data

3. Font Style Mismatches

Causes:

- Fallback font doesn't match original style
- Limited font selection available
- Font classification mismatches

Note: Fallback prioritizes legibility over style matching

4. Text Truncated Vertically

Causes:

- Text controls sized for specific scripts
- Different writing systems have different vertical requirements
- Font metrics not properly accounted for

5. Encoding Errors

Not Text Layout Issues - Upstream Problems:

Transcoding Failures:

- UTF-8 interpreted as legacy encoding
- Double-encoding artifacts
- Replacement characters (�) indicating conversion failure

Legacy Software:

- Non-Unicode capable applications
- Question marks for unsupported characters
- Incomplete UTF-16 surrogate handling

6. Incorrect Parsing of UTF-8 or UTF-16 Sequences

Software incorrectly assumes encoding format, leading to garbage text display

Implementation Implications

Performance Considerations

- Text display is extremely common operation
- Software optimized for efficient processing
- Incremental updates for document editing
- Constraint analysis to minimize re-layout

Complexity Management

- **Simple Scripts**: May use optimized basic layout paths
- **Complex Scripts**: Require full advanced layout pipeline
- **Modern Approach**: Apply advanced layout universally for consistent typography

Development Guidelines

1. **Don't rely on font fallback** for proper localization
2. **Test with target languages** early in development
3. **Understand platform differences** in text processing
4. **Plan for complex script requirements** from the beginning

Debugging Text Display Issues

Diagnostic Approach

1. **Identify the problem type:**
   - Layout/positioning issue
   - Font fallback problem
   - Encoding/conversion error
   - Platform/software limitation
2. **Gather information:**
   - What font was actually used?
   - What text processing occurred?
   - What are the original character codes?
   - What platform/software environment?
3. **Consult experts:**
   - Text layout engineers
   - Language/script experts
   - Platform documentation

Tools and Resources

- Unicode Character Database
- Script-specific documentation
- Platform text layout APIs
- Font inspection tools
- Text encoding validators

Key Takeaways

1. **Text display is complex** - What you see is the result of sophisticated processing
2. **Character ≠ Glyph** - One-to-many relationships are common
3. **Context matters** - Same characters may render differently based on surrounding text
4. **Scripts vary widely** - Solutions must accommodate diverse writing systems
5. **Font data drives behavior** - Advanced layout depends on font-provided rules
6. **Testing is crucial** - Problems often surface only with real-world multilingual content

Further Reading

- **Unicode Standard**: unicode.org
- **UAX #15**: Unicode Normalization Forms
- **UAX #29**: Unicode Text Segmentation
- **UAX #9**: Unicode Bidirectional Algorithm
- **OpenType Specification**: Microsoft Typography documentation
- **Platform APIs**: CoreText (Apple), DirectWrite (Microsoft), HarfBuzz documentation

Session recorded at Unicode Technical Workshop 2025
Notes compiled from presentation materials and transcript

Revision #2
Created 11 November 2025 17:28:56 by itsLittleKevin
Updated 11 November 2025 19:29:49 by itsLittleKevin