Graphite Overview

Sharon Correll

Version 3

1 Overview

The Graphite engine provides "smart" rendering for complex writing systems. It allows for the following complex behaviors:

- contextual substitution, insertion, and deletion
- reordering
- creation of ligatures with defined components
- positioning based on attachment points or shifting and kerning
- accessing of glyph metrics
- rule-based line-breaking
- application of the Unicode bidirectional algorithm.
- selecting and editing in all of the above situations.

2 Project history

Graphite was originally developed under the code name “WinRend” because it was intended to provide complex rendering on the Windows system. Early documents refer to WinRend and also to RDL which is now known as GDL (Graphite Description Language). Design work began around 1997, requirements were formalized in the first half of 1998, and coding began in the middle of 1998. An alpha was made available for testing in mid-2000. A beta of WorldPad, the first Graphite-enabled application, was released in September, 2001. Open-sourcing efforts began in earnest in the second half of 2002, with a port to Linux starting in late 2002. Version 2 of the API was finalized in mid-2005.

A thorough rewrite of the Graphite engine, called “Graphite2” was completed in 2010. It contains a completely new API, but can render using the same font tables as the original Graphite engine. This documentation pertains only to the original engine.

3 Graphite’s application interactions

From the point of view of the calling application, there are two main classes that serve as the top-level interface. The Font class represents a font object, which incorporates the font face, boldness, and italicization. Various platforms and environments provide different versions of Font, depending on how they want to read tables out of the font file. The Segment class represents a range of text laid out for rendering, with properly positioned glyphs. The Font is passed as an argument to the method that creates Segments (the constructor). Normally a Segment would consist of one line of text, or if there are changes in font, writing system, or style, several Segments may be needed to make up one line. In no case does a Segment ever consist of text displayed on more than one line. The calling application is responsible for laying out the segments into a reasonable paragraph.

In addition to Font and Segment, a TextSource must be provided by the application to supply key information; it serves as a “call-back” class. An instance of TextSource represents the text to be

rendered—the range of characters and their associated style information. A rudimentary version of TextSource is included in the open-source code, but many applications will need to supply their own.

Another call-back class is optional, needed only in the case where justification is required. The GrJustifier class is implemented by the application and used by the Graphite engine to make decisions about stretching and shrinking to achieve justification. A default GrJustifier class is supplied as part of the open-source code.

The SegmentPainter class used to handle drawing and other operations that occur as part of the editing process. An application may make use of the default SegmentPainter class, may subclass methods to customize the behavior, or may implement drawing and selection behaviors directly without using a SegmentPainter at all.

4 Graphite fonts

The behavior of the Graphite engine for a given writing system is specified by special-purpose tables in a TrueType font. The application creates a Font object based on this TrueType font, and as this object is passed to the Graphite engine, the tables in the font are used to perform the layout of glyphs.

5 Graphite processing

Processing in Graphite occurs in a series of passes. The first pass converts Unicode characters from the underlying string to glyph IDs; all the remainder of the processing happens in terms of glyph IDs. Each pass takes the output of the previous pass as its own input, looks for patterns, and applies rules to make modifications to the stream of glyphs. The final pass performs the final positioning of all glyphs. The output of the engine is a “segment,” a group of well-positioned glyphs that fits properly in the available space, and also understands its relationship to the original underlying string.

A class called GrTableManager is an important helper class to the engine. It keeps a list of passes and manages the process of calling each pass. An important point to be aware of is that processing occurs in small chunks. We *don’t* run the whole string through the first pass, then run all of the result through the second pass, etc. Instead, we process just little bit of the first pass, and send the results on the second pass, which does its processing and sends the results to the third pass, and so on. The mechanism is driven by the final pass attempting to do its positioning, and repeatedly requesting a little more input from the previous passes until it has filled up the available space or successfully processed all the input.

The reason for using this approach is to allow us to avoid doing more work than necessary in the case where we have a long string that will not fit on the line, and we need to insert a line-break. We gradually process a little bit of each pass until the final pass notices that the available space has been exceeded, and then it initiates a prcoess called “backtracking.” Backtracking involves finding a place to insert a linebreak, and then "unwinding" the subsequent glyph streams so that we can redo the processing while taking into account the inserted line-break. The trick is to unwind as little as possible but enough that we retain the context for each pass. This requires careful bookkeeping, and is managed by keeping track of “chunks.” (It may also be necessary to backtrack more than once before finding a line-break that will allow the segment to fit on the line.) See ‘WR Data Transform Engine.pdf’ (an early design document) for more details on this process.

5.1 Passes and streams

There are five kinds of passes: the glyph-generation pass, line-break passes, substitution passes, the bidi pass, and positioning passes—implemented by corresponding subclasses of GrPass. They occur in the order just listed. The first pass is the glyph-generation pass, and simply creates a stream of glyphs corresponding to characters. Substitution passes have the ability to substitute, insert, delete, and reorder glyphs; positioning and line-break passes do not. The bidi pass (if any) performs the Unicode bidi algorithm. (Many Graphite fonts will not have a bidi pass or any line-break passes.)

Each pass takes a stream of glyphs as input and generates a stream as output. That output stream then serves as input to the following pass. The first pass, which is always the glyph generation pass, is numbered 0 and generates stream #0. Stream #0 serves as input to pass #1 which outputs stream #1, etc.

The streams are implemented by instances of class GrSlotStream, containing the glyphs as they are being processed. Each stream holds a sequence of GrSlotStates, where each slot contains one glyph. The slots have pointers to slots in the previous streams to help keep track of the relationships between the output and the input. They also have a number of instance variables that represent "slot attributes" that are modified by the rules.

5.2 Matching and running rules

Pattern matching—determining which rules to fire—is achieved using finite state machines (FSMs). Each pass has its own finite state machine. The columns in the FSM correspond to classes of glyphs that are considered equivalent for the purposes of matching, and the rows are the states to transition to. Each FSM has a table assigning each glyph ID to a column in the table, and each final state indicates which rules are considered to be matched. See ‘WR FSM Design.pdf’ for more details.

The effect of firing of the rules is done though a stack machine mechanism. There are commands to perform subsitutions, look up glyph attributes, make mathematical calculations, etc. See ‘Stack Machine Commands.doc’ for a complete list.

5.3 Other details

Another tricky aspect of the engine relates to cross-line-boundary contextualization. In other words, the way a segment is rendered may be affected by the characters on the previous or following line. In order to make this happen, there is a block of information that is passed in when starting to create a new segment, which is information from the previous segment. It tells the engine how much to "back up" in order to take into account the stuff from the previous segment that will have an effect. ‘WR Data Transform Engine.pdf’ also discusses this process.

To get a complete overview of the capabilities of the Graphite system, refer to ‘GDL.pdf’.

6 Revision History

1. 15-May-2003: based on an earlier plain-text document.
2. 25-April-2006: updated to discuss API v2.
3. 3-Oct-2011: mention Graphite2

7 File Name

GraphiteOverview.rtf