Graphite2 Manual

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

1 Introduction

Graphite2 is a reimplementation of the SIL Graphite text processing engine. The reason for such a project has grown out of the experience gained in integration the Graphite engine into various applications and frameworks. The original engine was designed with different use cases in mind and optimised towards those. These optimisations get in the way of optimising for the actual use case requirements that the integration projects required. The Graphite2 engine, therefore, is designed for use where a simple shaping engine is required, much akin to the simpler OpenType engine interfaces that exist. Graphite2 has the following features over the original engine:

- Faster
- Smaller memory footprint
- More resiliant to font corruption
- Smaller code base

What is lost is:

- Selection support
- Line end contextuals
- Integrated line breaking to paragraph rendering

What is Graphite? Graphite is a *smart font* technology designed to facilitate the process known as shaping. This process takes an input Unicode text string and returns a sequence of positioned glyphids from the font. There are other similar *smart font* technologies including AAT and OpenType. While OpenType implementations are more prevalently integrated into applications than Graphite, Graphite still has a place. Graphite was developed primarily to address the generic shaping problem where current OpenType shaping engines do not address the specific needs of a font developer and the lead time on any changes to address those needs become prohibitive. This is a particular issue when creating solutions for some minority languages. In effect OpenType addresses the 80% problem and Graphite the 20% problem (or is that the 98% problem and the 2% problem?)

There are a number of reasons why someone might want to add Graphite smarts to their font:

- There is no consistent shaping across OpenType engines for the script and writing system that a font designer wants their font to support. Not all OpenType engines support all scripts in the same way. In addition, some writing system requirements do not fit with the shaping of the script that OpenType engines support.
- The font designer would like to implement more complex shaping and positioning than OpenType supports. For example, in Graphite one can position glyphs based on the positions and sizes of other glyphs.
- Graphite supports user defined features. The font designer may create and support any features they want and these can be presented to the user in a standardised way.

Graphite allows font implementors to implement their font their way. It does not require them to fit within an, often poorly specified, interface between the shaper and the font. This allows for quicker debugging and results. Graphite supports font debugging to identify what the shaper is doing all the way from input Unicode to output glyphs and positions, giving font designers better control over their font processing.

2 Building GraphiteNG

GraphiteNG uses cmake for its build system. The basic build procedure is to create a directory in which to build the library and produce the products. Then cmake is run to generate build files and then the build is run.

2.1 Linux

```
mkdir build
cd build
cmake -G "Unix Makefiles" ..
make
make test
```

3 Calling Graphite2

3.1 Introduction

The basic model for running graphite is to pass text, font and face information to create a segment. A segment consists of a linked list of slots which each correspond to an output glyph. In addition a segment holds charinfo for each character in the input text.

```
#include <graphite2/Segment.h>
#include <stdio.h>

/* usage: ./simple fontfile.ttf string */
int main(int argc, char **argv)
{
    int rtl = 0;                /* are we rendering right to left? probably not */
    int pointsize = 12;         /* point size in points */
    int dpi = 96;               /* work with this many dots per inch */

    char *pError;               /* location of faulty utf-8 */
    gr_font *font = NULL;
    size_t numCodePoints = 0;
    gr_segment * seg = NULL;
    const gr_slot *s;
    gr_face *face = gr_make_file_face(argv[1], 0);                          /*❶*/
    if (!face) return 1;
    font = gr_make_font(pointsize * dpi / 72.0f, face);                     /*❷*/
    if (!font) return 2;
    numCodePoints = gr_count_unicode_characters(gr_utf8, argv[2], NULL,
                (const void **)(&pError));                                  /*❸*/
    if (pError) return 3;
    seg = gr_make_seg(font, face, 0, 0, gr_utf8, argv[2], numCodePoints, rtl);  /*❹*/
    if (!seg) return 3;

    for (s = gr_seg_first_slot(seg); s; s = gr_slot_next_in_segment(s))        /*❺*/
        printf("%d(%f,%f) ", gr_slot_gid(s), gr_slot_origin_X(s), gr_slot_origin_Y(s));
    gr_seg_destroy(seg);
    gr_font_destroy(font);
    gr_face_destroy(face);
    return 0;
}
```

❶ The first parameter to the program is the full path to the font file to be used for rendering. This function loads the font and reads all the graphite tables, etc. If there is a fault in the font, it will fail to load and the function will return NULL.

❷ A font is merely a face at a given size in pixels per em. It is possible to support hinted advances, but this is done via a callback function.

❸ For simplification of memory allocation, graphite works on characters (Unicode codepoints) rather than bytes or gr_uint16s, etc. We need to calculate the number of characters in the input string (the second parameter to the program). Very often applications already know this. If there is an error in the utf-8, the pError variable will point to it and we just exit. But it is possible to render up to that point.

If your string is null terminated, then you don't necessarily have to calculate a precise number of characters. You can use a value that is greater than the number in the string and rely on graphite to stop at the terminating null. It is necessary to pass some value for the number of characters so that graphite can initialise its internal memory structures appropriately and not waste time updating them. Thus for UTF-16 and UTF-32 strings, one could simply pass the number of code units in the string. For UTF-8 it may be preferable to call gr_count_unicode_characters.

4 Here we create a segment. A segment is the results of processing a string of text with graphite. It contains all the information necessary for final rendering including all the glyphs, their positions, relationships between glyphs and underlying characters, etc.

5 A segment primarily consists of a linked list of slots. Each slot corresponds to a glyph in the output. The information about a glyph and its relationships is queried from the slot.

Source for this program may be found in tests/examples/simple.c

Assuming that graphite2 has been built and installed, this example can be built and run on linux using:

```
gcc -o simple -lgraphiteng simple.c
LD_LIBRARY_PATH=/usr/local/lib ./simple ../fonts/Padauk.ttf 'Hello World!'
```

Running simple gives the results:

```
43(0.000000,0.000000) 72(9.859375,0.000000) 79(17.609375,0.000000) 79(20.796875,0.000000)  ↩
    82(23.984375,0.000000) 3(32.203125,0.000000) 58(38.109375,0.000000)  ↩
    82(51.625000,0.000000) 85(59.843750,0.000000) 79(64.875000,0.000000)  ↩
    71(68.062500,0.000000) 4(76.281250,0.000000)
```

Not very pretty, but reassuring! Graphite isn't a graphical rendering engine, it merely calculates which glyphs should render where and leaves the actual process of displaying those glyphs to other libraries.

3.2 Slots

The primary contents of a segment is slots. These slots are organised into a doubly linked list and each corresponds to a glyph to be rendered. The linked list is terminated at each end by a NULL. There are also functions to get the first and last slot in a segment.

In addition to the main slot list, slots may be attached to each other. This means that two glyphs have been attached to each other in the GDL. Again, attached slots are held in a separate singly linked list associated with the slot to which they attach. Thus slots will be in the main linked list and may be in an attachment linked list. Each slot in an attachment linked list has the same attachment parent accessed via `gr_slot_attached_to()`. To get the start of the linked list of all the slots directly attached to a parent, one calls `gr_slot_first_attachment()` and then `gr_slot_next_attachment()` to walk forwards through that linked list. Given that a diacritic may attach to another diacritic, an attached slot may in its turn have a linked list of attached slots. In all cases, linked lists terminate with a NULL.

The core information held by a slot is the glyph id of the glyph the slot corresponds to (`gr_slot_gid()`); the position relative to the start of the segment that the glyph is to be rendered at (`gr_slot_origin_X()` and `gr_slot_origin_Y()`); the advance for the glyph which corresponds to the glyph metric advance as adjusted by kerning. In addition a slot indicates whether the font designer wants to allow a cursor to be placed before this glyph or not. This information is accessible via `gr_slot_can_insert_before()`.

3.3 CharInfo

For each unicode character in the input, there is a CharInfo structure that can be queried for such information as the code unit position in the input string, the before slot index (if we are before this character, which is the earliest slot we are before) and the corresponding after slot index.

3.4 Face

The gr_face type is the memory correspondance of a font. It holds the data structures corresponding to those in a font file as required to process text using that font. In creating a gr_face it is necessary to pass a function by which graphite can get hold of font tables. The tables that graphite queries for must be available for the lifetime of the gr_face, except when a gr_face is created with the faceOptions of gr_face_preloadAll. This then loads everything from the font at gr_face construction time, leaving nothing further to be read from the font when the gr_face is used. This reduces the required lifetime of the in memory font tables to the gr_make_face call. In situations where the tables are only stored for the purposes of creating a gr_face, it can save memory to preload everything and delete the tables.

3.5 Caching

Graphite2 has the capability to make use of a subsegmental cache. What this does is to chop each run of characters at a word break, as defined by the linebreaking pass. Each sub run is then looked up in the cache rather than calculating the values from scratch. The cache is most effective when similar runs of text are processed. For raw benchmark testing against wordlists, the cache can be slightly slower than uncached processing. But most people use real text in their documents and that has a much higher level of redundancy.

To use the cache, one simply creates a cached face, specifying the size of the cache in elements. A cache size of 5,000 to 10,000 has produced a good compromise between time and space.

In the example above, point 1 becomes:

```
    gr_face *face = make_file_face_with_seg_cache(argv[1]);
```

3.6 Clustering

It is common for applications to work with simplified clusters, these are sequences of glyphs associated with a sequence of characters, such that these simplified clusters are as small as possible and are never reordered or split in relation to each other. In addition, a cursor may occur between simplified clusters.

The following code gives an example algorithm for calculating such clusters:

```
#include <graphite2/Segment.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct cluster_t {
    unsigned int base_char;
    unsigned int num_chars;
    unsigned int base_glyph;
    unsigned int num_glyphs;
} cluster_t;

/* usage: ./cluster fontfile.ttf string */
int main(int argc, char **argv)
{
    int rtl = 0;                /* are we rendering right to left? probably not */
    int pointsize = 12;         /* point size in points */
    int dpi = 96;               /* work with this many dots per inch */

    char *pError;               /* location of faulty utf-8 */
    gr_font *font = NULL;
    size_t numCodePoints = 0;
    gr_segment * seg = NULL;
    cluster_t *clusters;
    int ic, ci = 0;
    const gr_slot *s, *is;
    FILE *log;
    gr_face *face = gr_make_file_face(argv[1], 0);
    if (!face) return 1;
    font = gr_make_font(pointsize * dpi / 72.0f, face);
    if (!font) return 2;
    numCodePoints = gr_count_unicode_characters(gr_utf8, argv[2], NULL,
                (const void **)(&pError));
    if (pError || !numCodePoints) return 3;
    seg = gr_make_seg(font, face, 0, 0, gr_utf8, argv[2], numCodePoints, rtl);      /*1*/
    if (!seg) return 3;

    clusters = (cluster_t *)malloc(numCodePoints * sizeof(cluster_t));
    memset(clusters, 0, numCodePoints * sizeof(cluster_t));
    for (is = gr_seg_first_slot(seg), ic = 0; is; is = gr_slot_next_in_segment(is), ic++)
    {
        unsigned int before = gr_slot_before(is);
        unsigned int after = gr_slot_after(is);
        while (clusters[ci].base_char > before && ci)                               /*2*/
        {
            clusters[ci-1].num_chars += clusters[ci].num_chars;
            clusters[ci-1].num_glyphs += clusters[ci].num_glyphs;
            --ci;
        }

        if (gr_slot_can_insert_before(is) && clusters[ci].num_chars
                && before >= clusters[ci].base_char + clusters[ci].num_chars)       /*3*/
        {
            cluster_t *c = clusters + ci + 1;
            c->base_char = clusters[ci].base_char + clusters[ci].num_chars;
            c->num_chars = before - c->base_char;
            c->base_glyph = ic;
            c->num_glyphs = 0;
            ++ci;
        }
        ++clusters[ci].num_glyphs;

        if (clusters[ci].base_char + clusters[ci].num_chars < after + 1)            /*4*/
            clusters[ci].num_chars = after + 1 - clusters[ci].base_char;
    }

    ci = 0;
    log = fopen("cluster.log", "w");
    for (s = gr_seg_first_slot(seg); s; s = gr_slot_next_in_segment(s))
    {
        fprintf(log, "%d(%f,%f) ", gr_slot_gid(s), gr_slot_origin_X(s),
                                   gr_slot_origin_Y(s));
        if (--clusters[ci].num_glyphs == 0)                                         /*5*/
        {
            fprintf(log, "\n");
            ++ci;
        }
    }
    fclose(log);
    free(clusters);
    gr_seg_destroy(seg);
    gr_font_destroy(font);
    gr_face_destroy(face);
    return 0;
}
```

1 Create a segment as per the example in the introduction.

❷ If this slot starts before the start of this cluster, then merge this cluster with the previous one and try again until this slot is within the current cluster.

❸ If this slot starts after the end of the current cluster, then create a new cluster for it.

❹ If this slot ends after the end of this cluster then extend this cluster to include it.

❺ Output a line break between each cluster.

3.7 Line Breaking and Justification

Whilst most applications will convert glyphs and positions out of the gr_slot structure into some internal structure, if graphite is to be used for justification, then it is necessary to line break the text and justify it within graphite's data structures. Graphite provides two functions to help with this. The first is gr_slot_linebreak_before() which will chop the slot linked list before a given slot. The application needs to keep track of the start of each of the subsequent linked lists itself, since graphite does not do that. After line breaking, the application may call gr_seg_justify() on each line linked list. The following example shows how this might be done in an application.

Notice that this example does not take into considering whitespace hanging outside the right margin.

```
#include <graphite2/Segment.h>
#include <stdio.h>
#include <stdlib.h>

/* usage: ./linebreak fontfile.ttf width string */
int main(int argc, char **argv)
{
    int rtl = 0;                /* are we rendering right to left? probably not */
    int pointsize = 12;         /* point size in points */
    int dpi = 96;               /* work with this many dots per inch */
    int width = atoi(argv[2]) * dpi / 72;  /* linewidth in points */

    char *pError;               /* location of faulty utf-8 */
    gr_font *font = NULL;
    size_t numCodePoints = 0;
    gr_segment * seg = NULL;
    const gr_slot *s, *sprev;
    int i;
    float lineend = width;
    int numlines = 0;
    const gr_slot **lineslots;
    gr_face *face = gr_make_file_face(argv[1], 0);
    if (!face) return 1;
    font = gr_make_font(pointsize * dpi / 72.0f, face);
    if (!font) return 2;
    numCodePoints = gr_count_unicode_characters(gr_utf8, argv[3], NULL,
                (const void **)(&pError));
    if (pError) return 3;
    seg = gr_make_seg(font, face, 0, 0, gr_utf8, argv[3], numCodePoints, rtl);  /*❶*/
    if (!seg) return 3;

    lineslots = (const gr_slot **)malloc(numCodePoints * sizeof(gr_slot *));
    lineslots[numlines++] = gr_seg_first_slot(seg);                             /*❷*/
    for (s = lineslots[0]; s; s = gr_slot_next_in_segment(s))
    {
        sprev = NULL;
        if (gr_slot_origin_X(s) > lineend)                                      /*❸*/
        {
            for (sprev = gr_slot_prev_in_segment(s); sprev;                     /*❹*/
                                    s = sprev, sprev = gr_slot_prev_in_segment(sprev))
            {
                int bw = gr_cinfo_break_weight(gr_seg_cinfo(seg, gr_slot_before(s)));
                if (bw < -15)                                                    /*5*/
                    continue;
                bw  = gr_cinfo_break_weight(gr_seg_cinfo(seg, gr_slot_after(sprev)));
                if (bw > 15)
                    continue;
                break;
            }
            lineslots[numlines++] = s;
            gr_slot_linebreak_before((gr_slot *)s);                              /*6*/
            lineend = gr_slot_origin_X(s) + width;                               /*7*/
        }
    }

    printf("%d:", width);
    for (i = 0; i < numlines; i++)
    {
        gr_seg_justify(seg, (gr_slot *)lineslots[i], font, width, 0, NULL, NULL); /*8*/
        for (s = lineslots[i]; s; s = gr_slot_next_in_segment(s))               /*9*/
            printf("%d(%.2f,%.2f@%d) ", gr_slot_gid(s), gr_slot_origin_X(s),  ↩
                gr_slot_origin_Y(s), gr_slot_attr(s, seg, gr_slatJWidth, 0));
        printf("\n");
    }
    free((void*)lineslots);
    gr_font_destroy(font);
    gr_face_destroy(face);
    return 0;
}
```

1. Create a segment as per the example in the introduction

2. Create an area to store line starts. There won't be more line starts than characters in the text. The first line starts at the start of the segment.

3. Scan through the slots, if we are past the end of the line then find somewhere to chop.

4. Scan backwards for a valid linebreak location.

5. We use 15 (syllable break) as an appropriate break value. A negative value means the break constraint is before the slot, so we test that. Likewise for after we test for a positive value.

6. Break the line here.

7. Update the line width for the new line based on the start of the new line.

8. Justify each line to be width wide. And tell it to skip final whitespace (as if that whitespace were outside the width).

9. Each line is a complete linked list that we can iterate over. We can no longer iterate over the whole segment. We have to do it line by line now.

3.8 Bidi

Bidirectional processing is complex not so much because of any algorithms involved, but because of the tendency for applications to address bidi text processing differently. Some try to do everything themselves, inverting the text order, etc. While others do nothing, expecting the shaper to resolve all the orders. In addition, there is the question of mirroring characters and where that is done. Graphite2 adds the complexity that it tries to enable extensions to the bidi algorithm by giving PUA characters directionality. To facilitate all these different ways of working, Graphite2 uses the `rtl` attribute to pass various bits to control bidi processing within the Graphite engine.

gr_nobidi	gr_nomirror	Description
0	0	Runs the bidi algorithm and does all mirroring

gr_nobidi	gr_nomirror	Description
0	1	Runs the bidi algorithm and mirrors those chars that don't have char replacements. It also un/remirrors anything that ends up in the opposite direction to the stated text direction on input.
1	0	Doesn't run the bidi algorithm but does do mirroring of all characters if direction is rtl.
1	1	Doesn't run the bidi algorithm and only mirrors those glyphs for which there is no corresponding mirroring character.

4 Font Features

Graphite fonts have user features. These are values that can be set to control all kinds of rendering effects from choosing particular glyph styles for a group of languages to how bad sequences should be displayed to almost anything.

A font (strictly speaking a face) has a set of features. Each feature has an identifier which is a 32-bit number which can take the form of a tag (4 characters) or if the top byte is 0 a number. Also each feature can take one of a set of values. Each feature has a UI name from the name table. In addition each value also has a UI name associated with it. This allows an application to list all the features in a font and to show their names and values in a user interface to allow user selection.

Feature values are held in a FeatureVal which is a compressed map of feature id to value. The map is indexed via a FeatureRef which may be quered from a face given an id. It is also possible to iterate over all the FeatureRefs in a face.

A face has a default featureVal corresponding to each language the face supports along with a default for other languages. A face may be asked for a copy of one of these default featureVals and then it may be modified to account for the specific feature settings for a run.

```
#include <graphite2/Font.h>
#include <stdio.h>

int main(int argc, char **argv)
{
    gr_uint16 i;
    gr_uint16 langId = 0x0409;
    gr_uint32 lang = 0;
    char idtag[5] = {0, 0, 0, 0, 0};                                /*❶*/
    gr_feature_val *features = NULL;
    gr_face *face = gr_make_file_face(argv[1], 0);
    int num = gr_face_n_fref(face);


    if (!face) return 1;
    if (argc > 2) lang = gr_str_to_tag(argv[2]);
    features = gr_face_featureval_for_lang(face, lang);             /*❷*/
    if (!features) return 2;
    for (i = 0; i < num; ++i)
    {
        const gr_feature_ref *fref = gr_face_fref(face, i);         /*❸*/
        gr_uint32 length = 0;
        char *label = gr_fref_label(fref, &langId, gr_utf8, &length);   /*❹*/
        gr_uint32 id = gr_fref_id(fref);                            /*❺*/
        gr_uint16 val = gr_fref_feature_value(fref, features);
        int numval = gr_fref_n_values(fref);
        int j;

        printf("%s ", label);
        gr_label_destroy(label);
        if (id <= 0x00FFFFFF)
            printf("(%d)\n", id);
        else
        {
            gr_tag_to_str(id, idtag);
            printf("(%s)\n", idtag);
        }

        for (j = 0; j < numval; ++j)
        {
            if (gr_fref_value(fref, j) == val)                          /*6*/
            {
                label = gr_fref_value_label(fref, j, &langId, gr_utf8, &length);
                printf("\t%s (%d)\n", label, val);
                gr_label_destroy(label);
            }
        }
    }
    gr_featureval_destroy(features);
    gr_face_destroy(face);
    return 0;
}
```

1. The easiest way to turn a char[4] into a string is to append a nul, hence we make a char[5].
2. Query the face for the default featureVal of the given lang or 0 for the default. The lang is a uint32 which has been converted from the string and is 0 padded (as opposed to space padded).
3. Iterate over all the features in a font querying for the featureRef.
4. Get the label in US English, for the feature name.
5. Get the id for the feature name so that applications can refer to it. The id may be numeric or a string tag.
6. Iterate over all the possible values for this feature and find the one the is equal to the value for the feature in the default featureVal. Then print out its details.

A sample run of.

```
./features ../fonts/Padauk.ttf ksw
```

Gives this output.

```
Khamti style dots (kdot)
        False (0)
Filled dots (fdot)
        False (0)
Lower dot shifts left (lldt)
        True (1)
Tear drop style washwe (wtri)
        True (1)
Long U with Yayit, long UU with Hato (ulon)
        False (0)
U and UU always full height (utal)
        False (0)
Insert dotted circles for errors (dotc)
        True (1)
Slanted hato (hsln)
        Sgaw style slanted leg with horizontal foot (1)
Disable great nnya (nnya)
        False (0)
Variant tta (vtta)
        False (0)
```

5 Hacking

In this section we look at coding conventions and some of the design models used in coding graphiteng.

5.1 Compiling and Integrating

To compile the graphite2 library for integration into another application framework, there are some useful utilities and also various definitions that need to be defined. The file src/files.mk will create three make variables containing lists of files that someone will want to build in order to build graphite2 from source as part of an application build. The variable names are controlled by setting _NS to a prefix that is then applied to the variable names _SOURCES, _PRIVATE_HEADERS and _PUBLIC_HEADERS. All files are listed relative to the variable _BASE (with its _NS expansion prefix). The _MACHINE variable with its _NS expansion prefix, must be set to *call* or *direct* depending on which kind of virtual machine to use inside the engine. gcc supports direct call, while all other compilers without a computed goto should use the call style virtual machine. See src/direct_machine.cpp and src/call_machine.cpp for details.

Various C preprocessor definitions are also used as part of compiling graphite2:

GRAPHITE2_EXPORTING
Needs to be set when building the library source if your are build it as a DLL. When unset the graphite header API will be marked dllimport for use in client code.

GRAPHITE2_STATIC
If set removes all dllimport or dllexport declspecs on the Graphtie2 API functions and makes them exportable for a clean static library build on Windows.

GRAPHITE2_NSEGCACHE
If set, the segment caching code is not compiled into the library. This can be used to save space if the engine is under space constraints both for code and memory. The default is that this is not defined.

GRAPHITE2_NFILEFACE
If set, the code to support creating a face directly from a font file is not included in the library. By default this is not set.

GRAPHITE2_NTRACING
If set, the code to support tracing segment creation and logging to a json output file is not built. However the API remains, it just won’t do anything.

GRAPHITE2_CUSTOM_HEADER
If set, then the value of this macro will be included as a header in Main.h (in effect, all source files). See Main.h for details.

5.2 Memory Allocation

While GraphiteNG is written in C++, it is targetted at environments where libstdc++ is not present. While this can be problematic since functions used in C++ may be arbitrarily placed in libc and libstdc++, there are general approaches that can help. To this end we use a mixed memory allocation model. For graphiteng classes, we declare our own new() methods and friends, allowing the use of C++ new() and all that it gives us in terms of constructors. For types that are not classes we use malloc() or the type safe version gralloc().

5.3 Missing Features

There are various facilities that silgraphite provides that graphite2 as yet does not. The primary motivation in developing graphite2 is that it be use case driven. Thus only those facilities that have a proven use case in the target applications into which graphite is to be integrated will be implemented.

Justification
Silgraphite has the ability to handle complex justification. This is not part of graphite2 yet. But then neither were any external applications making use of this facility within silgraphite.

Line End Contextuals
The cost of implementing line end contextuals is high and no fonts actually make use of it or need to make use of it. If the need were to rearise, line end contextuals could be integrated with a justification pass.

Ligature Components
Graphite has the ability to track ligature components and this feature is used in some fonts. But application support has yet to be proven and this is an issue looking for a use case and appropriate API.

SegmentPainter
Silgraphite provides a helper class to do range selection, cursor hitting and support text editing within a segment. These facilities were not being used in applications. Graphite2 does not preclude the addition of such a helper class if it would be of help to different applications, since it would be layered on top of graphite2.

Hinted Attachment Points
No use has been made of hinted attachment points, and so until a real use case requirement is proven, these are not in the font api. They can be added should a need be proven.

5.4 Hunting Speed

Graphite2 is written based on experience of using SilGraphite. SilGraphite was written primarily to be feature complete in terms of all features that may be needed. Graphite2 takes the experience of using SilGraphite and starts from a use case requirement before a feature is added to the engine. Thus a number of features that have not been used in SilGraphite have been removed, although the design is such that they can be replaced in future.

In addition, a number of techniques were used to speed up the engine. Some of these are discussed here.

Slot Stream
Rather than copying slots from one stream to another, the slot stream is allocated in blocks of slots and the processing is done in place. This means that each pass is executed to completion in sequence rather than using a pull model that SilGraphite uses. In addition, the slot stream is held as a linked list to keep the cost of insertion and deletion down for large segments.

Virtual Machine
The virtual machine that executes action and condition code is optimised for speed with different versions of the core engine code being used dependent upon compiler. The interpretted code is also pre-analysed for checking purposes and even some commands are added necessary for the inplace editing of the slot stream.

Design Space Positioning
The nature of the new processing model is that all concrete positioning is done in a final finalisation process. This means that all passes can be run in design space and then only at finalisation positioned in pixel space.

6 Android

Included in the contribs section of the Graphite2 source code is a way of integrating the Graphite2 library into an android application. It is known to work for Android 2.2 - Android 2.3.1 (API levels 8-10).

The sample application shows how it might be used.

```
/*  GRAPHITE2 LICENSING

    Copyright 2010, SIL International
    All rights reserved.

    This library is free software; you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published
    by the Free Software Foundation; either version 2.1 of License, or
    (at your option) any later version.
    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
    Lesser General Public License for more details.

    You should also have received a copy of the GNU Lesser General Public
    License along with this library in the file named "LICENSE".
    If not, write to the Free Software Foundation, 51 Franklin Street,
    Suite 500, Boston, MA 02110-1335, USA or visit their web page on the
    internet at http://www.fsf.org/licenses/lgpl.html.

    Alternatively, you may use this library under the terms of the Mozilla
    Public License (http://mozilla.org/MPL) or under the GNU General Public
    License, as published by the Free Sofware Foundation; either version
    2 of the license or (at your option) any later version.
*/

package org.sil.palaso.helloworld;

import android.app.Activity;
import android.graphics.Typeface;
import android.os.Bundle;
import android.webkit.WebView;
import android.widget.TextView;
import org.sil.palaso.Graphite;

public class HelloWorld extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        Graphite.loadGraphite();                                        // ❶
        Typeface tfp = (Typeface)Graphite.addFontResource(getAssets(),
                                "Padauk.ttf", "padauk", 0);             // ❷
        Typeface tfa = (Typeface)Graphite.addFontResource(getAssets(),
                                "Scheherazadegr.ttf", "Scheh", 1);

        TextView tv;
        WebView wv;
//      String s = "&#x1019;&#x1002;&#x1004;&#x103a;&#x1039;&#x1002;&#x101c;&#x102c;| ↩
    &#x1019;&#x1018;&#x1039;&#x1018;&#x102c;&#x104a; &#x1024;&#x1000;&#x1032;&#x1037;| ↩
    &#x101e;&#x102d;&#x102f;&#x1037;|&#x101b;&#x102c;|&#x1007;|&#x101d;&#x1004;&#x103a;| ↩
    &#x1010;&#x1004;&#x103a;|&#x1019;&#x100a;&#x103a;&#x1037; &#x1000;&#x103c;&#x1031;| ↩
    &#x100a;&#x102c;|&#x1001;&#x103b;&#x1000;&#x103a;|&#x1000;&#x102d;&#x102f;  ↩
    &#x1015;&#x103c;&#x102f;|&#x101c;&#x102f;&#x1015;&#x103a;| ↩
    &#x1015;&#x103c;&#x102e;&#x1038;|&#x1014;&#x1031;&#x102c;&#x1000;&#x103a;  ↩
    &#x1024;&#x100a;&#x102e;|&#x101c;&#x102c;|&#x1001;&#x1036;|&#x1021;| ↩
    &#x1005;&#x100a;&#x103a;&#x1038;|&#x1021;|&#x101d;&#x1031;&#x1038;| ↩
    &#x1000;&#x103c;&#x102e;&#x1038;|&#x1000;  ↩
    &#x1000;&#x1019;&#x1039;&#x1018;&#x102c;&#x1037;|&#x1000;&#x102f;|&#x101c;|&#x101e;| ↩
    &#x1019;&#x1002;&#x1039;&#x1002;|&#x1021;|&#x1016;&#x103d;&#x1032;&#x1037;| ↩
    &#x101d;&#x1004;&#x103a; &#x1014;&#x102d;&#x102f;&#x1004;&#x103a;|&#x1004;&#x1036;  ↩
    &#x1021;&#x102c;&#x1038;|&#x101c;&#x102f;&#x1036;&#x1038;|&#x1021;&#x102c;&#x1038;  ↩
    &#x1011;&#x102d;&#x102f;|&#x1000;&#x103c;&#x1031;|&#x100a;&#x102c;|&#x1005;&#x102c;| ↩
    &#x1010;&#x1019;&#x103a;&#x1038;|&#x1000;&#x103c;&#x102e;&#x1038;&#x104f;  ↩
    &#x1005;&#x102c;|&#x101e;&#x102c;&#x1038;|&#x1000;&#x102d;&#x102f;|&#x1021;| ↩
    &#x1019;&#x103b;&#x102c;&#x1038;|&#x1015;&#x103c;&#x100a;&#x103a;|&#x101e;&#x1030;| ↩
    &#x1010;&#x102d;&#x102f;&#x1037; &#x1000;&#x103c;&#x102c;&#x1038;|&#x101e;&#x102d;| ↩
    &#x1005;&#x1031;|&#x101b;&#x1014;&#x103a; &#x1000;&#x103c;&#x1031;|&#x100a;&#x102c;| ↩
    &#x1015;&#x102b;|&#x1019;&#x100a;&#x103a;&#x1037; &#x1021;| ↩
    &#x1000;&#x103c;&#x1031;&#x102c;&#x1004;&#x103a;&#x1038;|&#x1000;&#x102d;&#x102f;| ↩
    &#x101c;&#x100a;&#x103a;&#x1038;| ↩
    &#x1000;&#x1031;&#x102c;&#x1004;&#x103a;&#x1038;&#x104a;  ↩
    &#x1011;&#x102d;&#x102f;&#x1037;|&#x1015;&#x103c;&#x1004;&#x103a;| ↩
    &#x1014;&#x102d;&#x102f;&#x1004;&#x103a;|&#x1004;&#x1036;| ↩
    &#x1019;&#x103b;&#x102c;&#x1038;&#x104a; &#x101e;&#x102d;&#x102f;&#x1037;| ↩
    &#x1010;&#x100a;&#x103a;&#x1038;|&#x1019;|&#x101f;&#x102f;&#x1010;&#x103a;  ↩
    &#x1014;&#x101a;&#x103a;|&#x1019;&#x103c;&#x1031;| ↩
    &#x1019;&#x103b;&#x102c;&#x1038;&#x104f; &#x1014;&#x102d;&#x102f;&#x1004;&#x103a;| ↩
    &#x1004;&#x1036;|&#x101b;&#x1031;&#x1038; &#x1021;|&#x1006;&#x1004;&#x103a;&#x1037;| ↩
    &#x1021;|&#x1010;|&#x1014;&#x103a;&#x1038;|&#x1000;&#x102d;&#x102f;  ↩
    &#x101c;&#x102d;&#x102f;&#x1000;&#x103a;&#x104d; &#x1001;&#x103d;&#x1032;| ↩
    &#x1001;&#x103c;&#x102c;&#x1038;|&#x1001;&#x103c;&#x1004;&#x103a;&#x1038; &#x1019;| ↩
    &#x1015;&#x103c;&#x102f;|&#x1018;&#x1032;|&#x1021;|&#x1013;&#x102d;|&#x1000;| ↩
    &#x1021;&#x102c;&#x1038;|&#x1016;&#x103c;&#x1004;&#x103a;&#x1037; &#x1005;&#x102c;| ↩
    &#x101e;&#x1004;&#x103a;|&#x1000;&#x103b;&#x1031;&#x102c;&#x1004;&#x103a;&#x1038;| ↩
    &#x1019;&#x103b;&#x102c;&#x1038;|&#x1014;&#x103e;&#x1004;&#x103a;&#x1037; &#x1021;| ↩
    &#x1001;&#x103c;&#x102c;&#x1038;|&#x1015;|&#x100a;&#x102c;|&#x101b;&#x1031;&#x1038;  ↩
    &#x1021;|&#x1016;&#x103d;&#x1032;&#x1037;|&#x1021;|&#x1005;&#x100a;&#x103a;&#x1038;| ↩
    &#x1019;&#x103b;&#x102c;&#x1038;|&#x1010;&#x103d;&#x1004;&#x103a;  ↩
    &#x1011;&#x102d;&#x102f;|&#x1000;&#x103c;&#x1031;|&#x100a;&#x102c;|&#x1005;&#x102c;| ↩
    &#x1010;&#x1019;&#x103a;&#x1038;|&#x1000;&#x103c;&#x102e;&#x1038;| ↩
    &#x1000;&#x102d;&#x102f; &#x1016;&#x103c;&#x1014;&#x103a;&#x1037;| ↩
    &#x1001;&#x103b;&#x102d; &#x101d;&#x1031;|&#x1004;&#x103e; &#x1005;&#x1031;| ↩
    &#x101b;&#x1014;&#x103a;&#x104a; &#x1019;&#x103c;&#x1004;&#x103a;|&#x101e;&#x102c;| ↩
    &#x1021;&#x1031;&#x102c;&#x1004;&#x103a; &#x1015;&#x103c;|&#x101e;| ↩
    &#x1011;&#x102c;&#x1038;|&#x1005;&#x1031;|&#x101b;&#x1014;&#x103a;&#x104a;| ↩
    &#x1016;&#x1010;&#x103a;|&#x1000;&#x103c;&#x102c;&#x1038;|&#x1005;&#x1031;| ↩
    &#x101b;&#x1014;&#x103a;|&#x1014;&#x103e;&#x1004;&#x103a;&#x1037; &#x1021;| ↩
    &#x1013;&#x102d;&#x1015;&#x1039;&#x1015;&#x102b;&#x101a;&#x103a;| ↩
    &#x101b;&#x103e;&#x1004;&#x103a;&#x1038;|&#x101c;&#x1004;&#x103a;&#x1038;  ↩
    &#x1016;&#x1031;&#x102c;&#x103a;|&#x1015;&#x103c;|&#x1005;&#x1031;| ↩
    &#x101b;&#x1014;&#x103a; &#x1006;&#x1031;&#x102c;&#x1004;&#x103a;| ↩
    &#x101b;&#x103d;&#x1000;&#x103a;|&#x1015;&#x102b;|&#x1019;&#x100a;&#x103a;&#x1037;  ↩
    &#x1021;|&#x1000;&#x103c;&#x1031;&#x102c;&#x1004;&#x103a;&#x1038;| ↩
    &#x1016;&#x103c;&#x1004;&#x103a;&#x1037; &#x101c;&#x100a;&#x103a;&#x1038;| ↩
    &#x1000;&#x1031;&#x102c;&#x1004;&#x103a;&#x1038; &#x1006;&#x1004;&#x103a;&#x1037;| ↩
    &#x1006;&#x102d;&#x102f; &#x101c;&#x102d;&#x102f;&#x1000;&#x103a;| ↩
    &#x101e;&#x100a;&#x103a;&#x104b;".replace("|", "\u200B");
        String s = "&#x644;&#x645;&#x651;&#x627; &#x643;&#x627;&#x646;  ↩
            &#x627;&#x644;&#x627;&#x639;&#x62a;&#x631;&#x627;&#x641;  ↩
            &#x628;&#x627;&#x644;&#x643;&#x631;&#x627;&#x645;&#x629;  ↩
            &#x627;&#x644;&#x645;&#x62a;&#x623;&#x635;&#x644;&#x629; &#x641;&#x64a;  ↩
            &#x62c;&#x645;&#x64a;&#x639; &#x623;&#x639;&#x636;&#x627;&#x621;  ↩
            &#x627;&#x644;&#x623;&#x633;&#x631;&#x629; ( ↩
            &#x627;&#x644;&#x628;&#x634;&#x631;&#x64a;&#x629;)  ↩
            &#x648;&#x628;&#x62d;&#x642;&#x648;&#x642;&#x647;&#x645;  ↩
            &#x627;&#x644;&#x645;&#x62a;&#x633;&#x627;&#x648;&#x64a;&#x629;\u221A  ↩
            &#x627;&#x644;&#x62b;&#x627;&#x628;&#x62a;&#x629; &#x647;&#x648;  ↩
            &#x623;&#x633;&#x627;&#x633; &#x627;&#x644;&#x62d;&#x631;&#x64a;&#x629;  ↩
            &#x648;&#x627;&#x644;&#x639;&#x62f;&#x644; \u06F1\u06F2\u06F3  ↩
            &#x648;&#x627;&#x644;&#x633;&#x644;&#x627;&#x645; &#x641;&#x64a;  ↩
            &#x627;&#x644;&#x639;&#x627;&#x644;&#x645;.";
        String w = "\uFEFF<html><body style=\"font-family: Scheh\">Test: "
                                    + s + "</body></html>";                                         // ❸

        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        tv = (TextView) findViewById(R.id.tv);
        tv.setText(s);
        tv.setTypeface(tfa, 0);                                                                      // ❹
        tv.setTextSize((float)(tv.getTextSize() * 1.2));
        wv = (WebView) findViewById(R.id.wv);
        wv.loadData(w, "text/html", "UTF-8");
    }
```

```
}
```

❶ This sets up the graphite redirection for the core graphics layer.

❷ The graphite integration needs to be informed of the fonts we intend to use The fonts are included in the assets/ directory and we give the filename, then an identifier to use for the font within web styling, followed by whether the font is presumed to be rendering right to left or left to right text.

❸ Here we package the simple text as HTML and set the font. Notice how the font-family: parameter corresponds to the font identifier in <2>.

❹ Registering a font also returns a Typeface for the font and this can be used to set the typeface for a widget.