Graphite2 Manual

Martin Hosken

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 resilient 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. == Building Graphite 2 ==

Graphite 2 is made available as a source tarball from these urls: https://github.com/silnrsi/graphite/releases or http://sf.net/projects/silgraphite/files/graphite2

While Graphite 2 is written in C++11, it is written in a subset that ensures no runtime dependency on the language runtime.

Graphite 2 uses cmake for its build system. The basic build procedure is to create a directory in which to build the library and executable products. Then cmake is run to generate build files and then the build is run.

1.1 Linux

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

This will do a default build of Graphite with minimal dependencies on other packages. There are various option settings see Generator configuration options>.

On amd64 architecture if you wish build and test 32 bit binaries this is possible using the following cmake invocation:

```
CFLAGS=-m32 CXXFLAGS=-m32 cmake ..
make
make test
```

You will need g++-multilib support see Limitations

It is possible to use clang to build and test Graphite. Use this build command:

```
CC=clang CXX=clang++ cmake ..
make
```

You will need libc `libc`-abi see clang-asan section of Limitations.

1.2 Windows

1. Create your build directory

   ```
   mkdir build
   cd build
   ```

2. Generate project files for your build system

   You need to specify the CMAKE_BUILD_TYPE as some Windows generators require it.

   ```
   cmake -DCMAKE_BUILD_TYPE:STRING=Release ..
   ```

   CMake will automatically detect your build system and generate a project for that. The options passed above will do a default build of Graphite with minimal dependencies on other packages. You may wish to specify a build system other than the automatically detected one, for examples if you have multiple versions of Visual Studio installed or other toolchains such as MinGW you wish build under. To do this pass the `-G <generator name>` option to the initial cmake configuration call, for example for Visual Studio 8:

   ```
   cmake -G "Visual Studio 12 2013" -DCMAKE_BUILD_TYPE:STRING=Release ..
   ```

   or for MinGW

   ```
   cmake -G "MinGW Makefiles" -DCMAKE_BUILD_TYPE:STRING=Release ..
   ```

   Tip
   You can get a list of generators CMakes supports with `cmake --help`.

3. Build graphite binaries

   ```
   cmake --build .
   ```

When building with using the Visual Studio generator you will need to append `--config Debug` or `--config Release` for you debug and release builds respectively to the end of above command. Depending on your chosen generator the next step varies, for MS Visual Studio projects you will need to run:

```
cmake --build . --target RUN_TESTS
```

for everything else:

```
cmake --build . --target test
```

4. Rebuilds

   You can clean the project with:

   ```
   cmake --build . --target clean
   ```

   Or just delete the build directory and start again.

1.3 Generator configuration options

There are various option settings that can be passed to cmake when generating. They are described here, along with their type and possible and default values. Boolean values may take the value OFF or ON. Options may be set using the -Doption=value command line option. For example: -DGRAPHITE2_COMPARE_RENDERER:BOOL=ON

BUILD_SHARED_LIBS:BOOL
Specifies that libgraphite2 should be built as a shared library, setting this to OFF will cause it built as a static library. When OFF this will also disable a few tests which either test aspects of a shared library or require one (such as any test case with a python driver).
The default is ON.

CMAKE_BUILD_TYPE:STRING
Specifies which type of build to do. It is a string and may take the values: Release, RelWithDeb, Debug.
The default is Release. This must be specified on Windows.

GRAPHITE2_COMPARE_RENDERER:BOOL
Specifies whether to build the comparerenderer program that may link to silGraphite or harfbuzz, if libraries of those packages are installed.
The default is OFF.

GRAPHITE2_NFILEFACE:BOOL
Turns off FileFace support to save code space.
The default is OFF.

GRAPHITE2_NTRACING:BOOL
Turns off tracing support to save code space. Tracing support allows debug output of segment creation.
The default is ON.

GRAPHITE2_VM_TYPE:STRING
This value can be `auto`, `direct` or `call`. It specifies which type of virtual machine processor to use. The value of `auto` tells the system to work out the best approach for this architecture. A value of `direct` tells the system to use the direct machine which is faster. The value of `call` tells the system to use the slower but more cross compiler portable call based machine.
The default is auto.

GRAPHITE2_SANITIZERS:STRING
This turns on compile time support for the specified sanitizers. This works with both gcc and clang, though there are some differences in which sanitizers each offers. The fuzzer sanitizer causes libFuzzer based fuzzing targets to be built as well, but this currently only works under clang.
The default is an empty string.

Bear in mind that ASAN will not work with ulimit constraints so running the fuzztest may result in problems.

1.4 Limitations

There are some hard build dependencies:

python
: To run the make test and make fuzztest, the build system requires python v2.7 or later.

Microsoft Visual C++
: You will need Microsoft Visual Studio 12 2013 or later as we use some C++11 features.

Other configuration related dependencies:

fonttools
: This python library supports truetype font reading.

g++-multilib
: If building 32bit binaries under a 64bit Linux host this is required for successful linking. These are the `g++-multilib` and `libc6-dev-i386` packages on Debian and derivatives and `glibc-devel.i686`, `glibc-devel` and `libstdc++-de` on Redhat OSs

clang
: To build with clang under linux you will need to ensure you have installed libc `and libc`abi packages. The easiest way to do that on Debian & derivatives is to install the libc`-dev and libc`abi-dev packages. == Calling Graphite2 ==

1.5 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;
}
```

1. 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.

2. 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.

3. 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 -lgraphite2 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.

This example is pretty simple and uses a convenient way to load fonts into Graphite for testing purposes. But when integrating into real applications, that is rarely the most appropriate way. Instead the necessary font information comes to Graphite via some other data structure with its own accessor functions. In the following example we show the same application but using a FreeType font rather than a font file.

```
#include <graphite2/Segment.h>
#include <stdio.h>
#include <stdlib.h>
#include "ft2build.h"
#include FT_FREETYPE_H
#include FT_TRUETYPE_TABLES_H

const void *getTable(const void *appHandle, unsigned int name, size_t *len)
{
    void *res;
    FT_Face ftface = (FT_Face)appHandle;
    FT_ULong ftlen = 0;
    FT_Load_Sfnt_Table(ftface, name, 0, NULL, &ftlen);     /* find length of table */
    if (!ftlen) return NULL;
    res = malloc(ftlen);                                 /* allocate somewhere to hold it ↩
        */
    if (!res) return NULL;
    FT_Load_Sfnt_Table(ftface, name, 0, res, &ftlen);      /* copy table into buffer */
    *len = ftlen;
    return res;
}

void releaseTable(const void *appHandle, const void *ptr)
{
    free((void *)ptr);          /* simply free the allocated memory */               /*❶*/
}

float getAdvance(const void *appFont, unsigned short glyphid)
{
    FT_Face ftface = (FT_Face)appFont;
    if (FT_Load_Glyph(ftface, glyphid, FT_LOAD_DEFAULT)) return -1.;  /* grid fit glyph */
    return ftface->glyph->advance.x;                    /* return grid fit advance */
}

/* usage: ./freetype 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;
    FT_Library ftlib;
    FT_Face ftface;
    gr_face_ops faceops = {sizeof(gr_face_ops), &getTable, &releaseTable};          /*❷*/
    gr_font_ops fontops = {sizeof(gr_font_ops), &getAdvance, NULL};
    /* Set up freetype font face at given point size */
    if (FT_Init_FreeType(&ftlib)) return -1;
    if (FT_New_Face(ftlib, argv[1], 0, &ftface)) return -2;
    if (FT_Set_Char_Size(ftface, pointsize << 6, pointsize << 6, dpi, dpi)) return -3;

    face = gr_make_face_with_ops(ftface, &faceops, gr_face_preloadAll);              /*❸*/
    if (!face) return 1;
    font = gr_make_font_with_ops(pointsize * dpi / 72.0f, ftface, &fontops, face);  /*❹*/
    if (!font) return 2;
    numCodePoints = gr_count_unicode_characters(gr_utf8, argv[2], NULL,
                (const void **)(&pError));
    if (pError) return 3;
    seg = gr_make_seg(font, face, 0, 0, gr_utf8, argv[2], numCodePoints, rtl);
    if (!seg) return 3;

    for (s = gr_seg_first_slot(seg); s; s = gr_slot_next_in_segment(s))
        printf("%d(%f,%f) ", gr_slot_gid(s), gr_slot_origin_X(s) / 64,
                             gr_slot_origin_Y(s) / 64);                             /*❺*/
    gr_seg_destroy(seg);
    gr_font_destroy(font);
    gr_face_destroy(face);
    /* Release freetype face and library handle */
    FT_Done_Face(ftface);
    FT_Done_FreeType(ftlib);
    return 0;
}
```

1. We cast the pointer to remove its const restriction. Since when the memory was allocated, it was passed to Graphite as a read only memory block, via const, it gets passed back to us as a read only memory block. But we are the owner of the block and so can mess with it (like freeing it). So we are free to break the const restriction here.

2. The structure of an operations structure is to first hold the size of the structure and then the pointers to the functions. Storing the size allows an older application to call a newer version of the Graphite engine which might support more function pointers. In such a case, all those newer pointers are assumed to be NULL.

3. Pass the function pointers structure for creating the font face. The first function is called to load a table from the font and pass it back to Graphite. The second is called by Graphite to say that it no longer needs the table loaded in memory and will make no further reference to it.

4. Pass a function pointers structure for fonts. The two functions (either can be NULL) return the horizontal or vertical advance for a glyph in pixels. Notice that usually fractional advances are preferable to grid fit advances, unless working entirely in a low resolution graphical framework.
   The code following is virtually identical to the fileface code, apart from some housekeeping at the end.

5. Note that freetype works with fixed point arithmetic of 26.6, thus 1.0 is stored as 64. We therefore multiply the pointsize by 64 (or shift left by 6) and divide the resulting positions down by 64 to true floating point values.

Building and running this example gives similar, but not identical, results to the simple case. Notice that since the advances are grid fit, all the positions are integral, unlike the fractional positioning in the simple application:

```
43(0.000000,0.000000) 72(9.000000,0.000000) 79(17.000000,0.000000) 79(20.000000,0.000000)  ↩
    82(23.000000,0.000000) 3(31.000000,0.000000) 58(37.000000,0.000000)  ↩
    82(51.000000,0.000000) 85(59.000000,0.000000) 79(64.000000,0.000000)  ↩
    71(67.000000,0.000000) 4(75.000000,0.000000)
```

1.6 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.

glyph_string.png

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()`.

1.7 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.

1.8 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.

1.9 Caching

Graphite2 had the capability to make use of a subsegmental cache. Each sub run was then looked up in the cache rather than calculating the values from scratch. While the cache could be effective when similar runs of text were being processed, it often didn't fit with target applications exisiting framework and was a source of several bugs, it was marked as deprecated in 1.3.7. In order to avoid an API change `gr_make_face_with_seg_cache`, `gr_make_face_with_seg_cache_and_ops` and `gr_make_file_face_with_seg_cache` now simply alias their non-caching counterparts and ignore the cacheSize paramter.

1.10 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:

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

typedef struct cluster_t {
    size_t base_char;
    size_t num_chars;
    size_t base_glyph;
    size_t 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;
    size_t lenstr = strlen(argv[2]);
    gr_segment * seg = NULL;
    cluster_t *clusters;
    size_t 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);      /*❶*/
    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++)
    {
        size_t before = gr_cinfo_base(gr_seg_cinfo(seg, gr_slot_before(is)));
        size_t after = gr_cinfo_base(gr_seg_cinfo(seg, gr_slot_after(is)));
        int    nAfter;
        size_t cAfter;
        while (clusters[ci].base_char > before && ci)                               /*❷*/
        {
            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)       /*❸*/
        {
            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;

        nAfter = gr_slot_after(is) + 1;
        cAfter = nAfter < numCodePoints ? gr_cinfo_base(gr_seg_cinfo(seg, nAfter)) : lenstr ↩
            ;
        if (clusters[ci].base_char + clusters[ci].num_chars < cAfter)             /*❹*/
            clusters[ci].num_chars = cAfter - 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)                                         /*❺*/
        {
            fprintf(log, "[%zd+%zd]\n", clusters[ci].base_char, clusters[ci].num_chars);
            ++ci;
        }
    }
    fclose(log);
    free(clusters);
    gr_seg_destroy(seg);
    gr_font_destroy(font);
    gr_face_destroy(face);
    return 0;
}
```

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

2. 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.

3. If this slot starts after the end of the current cluster, then create a new cluster for it. You can't start a new cluster with a glyph that cannot take a cursor position before it. Also don't make a new cluster the first time around (i.e. at the start of the string).

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

5. Output a line break between each cluster.

1.11 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>

/* calculate the breakweight between two slots */
int breakweight_before(const gr_slot *s, const gr_segment *seg)
{
    int bbefore = gr_cinfo_break_weight(gr_seg_cinfo(seg, gr_slot_after( ↩
        gr_slot_prev_in_segment(s))));
    int bafter = gr_cinfo_break_weight(gr_seg_cinfo(seg, gr_slot_before(s)));

    if (!gr_slot_can_insert_before(s))
        return 50;
    if (bbefore < 0) bbefore = 0;
    if (bafter > 0) bafter = 0;
    return (bbefore > bafter) ? bbefore : bafter;
}

/* 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 = (float)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))                 /*❸*/
    {
        sprev = NULL;
        if (gr_slot_origin_X(s) > lineend)                                     /*❹*/
        {
            while (s)
            {
                if (breakweight_before(s, seg) >= gr_breakWord)                /*❺*/
                    break;
                s = gr_slot_prev_in_segment(s);                                /*❻*/
            }
            lineslots[numlines++] = s;
            gr_slot_linebreak_before((gr_slot *)s);                            /*❼*/
            lineend = gr_slot_origin_X(s) + width;                             /*❽*/
        }
    }

    printf("%d:", width);
    for (i = 0; i < numlines; i++)
    {
        gr_seg_justify(seg, (gr_slot *)lineslots[i], font, width, 0, NULL, NULL); /*❾*/
        for (s = lineslots[i]; s; s = gr_slot_next_in_segment(s))             /*❿*/
            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_seg_destroy(seg);
    gr_font_destroy(font);
    gr_face_destroy(face);
    return 0;
}
```

❶ Create a segment as per the example in the introduction

❷ 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.

❸ A simplistic approach would scan forwards using `gr_slot_next_sibling_attachment`, thus walking the bases. The bases are guaranteed to be ordered graphically, and so we can chop when we pass the line end. But in some cases, particularly Arabic, fonts are implemented with one base per word and all the other base characters are attached in a chain from that. So we need to walk the segment by slot, even considering attached slots. This is not a problem since attached slots are not going to have a wildly different position to their base and if one leaks over the end of the line, the breakweight considerations will get us back to a good base.

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

❺ We use 15 (word break) as an appropriate break value.

❻ Scan backwards for a valid linebreak location.

❼ Break the line here.

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

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

10. 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.

1.12 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
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.

2 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};                              /*1*/
    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);             /*2*/
    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)                           /*❻*/
            {
                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;
}
```

❶ The easiest way to turn a char[4] into a string is to append a nul, hence we make a char[5].

❷ 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).

❸ Iterate over all the features in a font querying for the featureRef.

❹ Get the label in US English, for the feature name.

❺ Get the id for the feature name so that applications can refer to it. The id may be numeric or a string tag.

❻ 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)
```

3 Font Topics

3.1 Guard Space

Graphite introduces guard space around diacritics. Sometimes a diacritic is wider than its base and the diacritic is in danger of crashing into clusters on either side of it. To stop that happening, graphite allows the font designer to signal that they want guard space to be ensured around a diacritic. For example, if a diacritic glyph is designed with a positive left side bearing and a positive right side bearing, the graphite engine will ensure that the cluster honours those side bearings for the diacritic. Of course, if the base character is wider than the diacritic, then no guard space is needed.

The basic principle for cluster adjustment is that if a diacritic has a non-zero advance and after positioning, the advance of the diacritic is greater than the advance of the base then the advance of the base is increased to be the same as the positioned advance of the diacritic. Likewise if the origin of the diacritic is to the left of the origin of the base character, then the cluster is adjusted so that the origin of the diacritic is now where the origin of the base character was and the base character is moved to the right. Notice that this only happens if the origin of the diacritic is to the left of where it is attached or the advance is non-zero.

In the following image, we use a dotless i with a combining tilde over it, which is wider than the dotless i. The four scenarios show how positioning the tilde and setting its advance controls the origin and advance of the attached cluster:

guardspace.png

Each line shows the two glyphs as they are designed with the origin and advance (if any). The third element on the line is the combined cluster. Again the origin and advance for the cluster is shown with solid lines and any subglyph origins and advances that don't coincide with a solid line, are shown dotted. Notice that we don't show the implied attachment point used to attach the tilde to the dotless i.

The first line shows the diacritic as if it were a full base character, with positive left and right side bearings. When the glyphs attach the origin and the advance of the dotless i (shown as dotted lines) are pushed out to the origin and advance of the diacritic. Notice that graphite uses the wider advance and origin regardless of which component of the cluster they come from.

The second line shows the other extreme. Here no guard space is inserted. The diacritic is to the left of the origin and the advance is zero. The cluster origin and advance are taken from the base glyph. The dotted line shows the origin and advance of the diacritic.

These two lines are the most common cases that people want to use for rendering diacritics and whether space is automatically inserted to avoid collisions. The next two are rarely used but help to explain how the mechanism works.

The third line has guard space on the left only. For this the diacritic is drawn to the right of the origin but the advance width is set to 0. The effect is that guard space is inserted on the left, because there is a positive left side bearing (or more precisely the origin of the diacritic is to the left of the origin of the base when the two glyphs combine). Thus the dotless i origin (shown as a dotted line) is pushed out. Actually the whole cluster is pushed to the right so that the origin of the diacritic is aligned with where the origin of the base glyph was.

The fourth line gives guard space only after the diacritic. In this case, the diacritic is drawn to the left of the origin, and so no left guard space can occur, since the origin of the diacritic is to the right of the dotless i. The diacritic has also been drawn so that it finishes at the origin. This ensures that the guard space to the right is the same as the advance. It need not be. The cluster has the same origin as the base glyph. The base glyph advance is shown as a dotted line, which while not necessarily coinciding with the origin of the diacritic, will be close. Finally the advance for the cluster is the advance from the diacritic.

Since it is possible to change the advance width of a glyph (or at least for a particular instance of a glyph or slot), it is possible to dynamically control some of the guard space mechanism within GDL. It is possible to use a rule to change from both to or from left only. Likewise it is possible to use a rule to change from none to or from right only. But, unfortunately, it is not possible to shift glyphs within their drawn space and so switch between both and none, purely from GDL. == Hacking ==

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

3.2 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 and clang support 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 you are building 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 Graphite2 API functions and makes them exportable for a clean static library build on Windows.

GRAPHITE2_BUILTINS
If set this will use gcc or clang *builtin intrinsics where appropriate when compiling for some architectures (Intel with SSE4.2 or later and ARMv7a with a NEON capable FPU) this will cause the compiler to emit specialised instructions where it can or fall back to library code. Currently this is only used for* builtin_popcount and should only be turned when you know the instruction will be emitted as the gcc __builtin_popcount is 2x slower on Intel than Graphite's own version.

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.

3.3 Floating point maths

On some Intel 32 bit processors, gcc on Linux, will attempt to optimise floating point operations by keeping floating point values in 80 bit (or larger) float registers for as long as possible. In some of the floating point maths performed for collision detection this results in error accumulating which produces different results between 64 bit and 32 bit native code. The current work around is to pass either -ffloat-store (the slow but widely compatible option) or -mfpmath=sse -msse2 (the faster but not as generic), this causes float values to be rounded between every operation or the use of sse float operations which are more strictly specified and therefore more predictable across processor generations. This has not been observed to be an issue on 32 bit MSVC or Mac 32 bit compilers and is not an issue on ARM 32 bit either.

3.4 Thread Safety

The Graphite engine has no locking or thread safe storage. But it is possible to use the Graphite engine in a thread safe manner. Face creation must be done in one thread and if `gr_face_preloadAll` is set, all font table interaction will occur while the face is being created. That is no calls will be made to the `get_table` function or the `release_table` function during segment creation. References to the name table, made during calls to `gr_fref_label` use an internal copy of that table to ensure that all table interaction is completed after `gr_make_face` is called. Note that none of this precludes an application handling thread issues around font table querying and releasing and graphite being used in a lazy table query manner. Font objects must be created without hinted advances otherwise the application is responsible for handling the shared `AppHandle` resource during segment creation. Following this, the face is a read only object and can be shared across different threads, and so segment creation is thread safe. Following creation, segments may be shared across threads so long as they are not modified (using `gr_seg_justify` or `gr_slot_linebreak_before`).

Any use of logging will break thread safety. Face specific logging involves holding a file open for as long as logging is active, and so segments cannot be made from a shared face across different threads. Further, if gr_start_logging is called with NULL for the face, a non locked global library variable is written to which will impact across threads and all face creation. Logging should therefore only be used in a single threaded context.

Any other use of the graphite engine in a multithreaded context will involve the application in doing its own locking.

3.5 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().

3.6 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.

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.

3.7 Hunting Speed

Graphite2 is written based on the 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 imported 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 interpreted 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. == Testing Graphite2 ==

During development Graphite2 is regularly checked against its test suite of over a 100 test cases. This happens automatically on every check-in to the default branch thanks to our continuous build server for Windows and Linux. Prior to each major release it's tested using an automated fuzzing system, which checks for robustness in the face of corrupted font files. This set of fuzz tests uses valgrind to check for rogue memory accesses and runs several thousand tests on each of the 4 major test fonts and takes considerably longer to run. We also have a growing suite of fuzz test regressions culled from logs generated by the above fuzz test system, which can be run with a make command. These are intended to be run before bug fix release and other point releases.

3.8 Running the tests

3.8.1 Running the standard tests

The standard test suite, the same one run on every checkin to the graphite2 project repository, can be run with a simple:

```
make test
```

3.8.2 Runnging the fuzztest regressions

```
make fuzztest
```

These should be run before point and bug fix releases.

3.8.3 Running the full fuzz test script

```
full-fuzz-test.sh script [fuzztest options]
```

This script exercises graphite over 4 scripts Myanmar, Devangari, extended Latin and Arabic using 4 fonts and text in the Myanmar, Nepalese, Yoroba and Arabic languages. It uses the `fuzzcomparerender` script to fuzz every byte of each font with a random value and enforce generous enough runtime and memory resource limits to detect infinite loops or memory leaks. A successfull run of this script will produce four empty log files. Passing `--valgrind` to the script this is passed down to the `fuzztest` program which will run the test program with valgrind, this increases the runtime of script considerably. Normally the script can run all four tests within 24 hours, fully loading our 4 core hyperthreaded Xeon/Core i7 system. Using the valgrind option this takes approximately a week on the same system.

3.8.4 Running fuzzcomparerender

```
tests/fuzzcomparerender <font name> <text name> [-r] [fuzztest options]
```

The `font name` must be the basename of a font file under `tests/fonts` and the `text name` must be the basename of a text file from `tests/texts`. The `-r` option if present is passed to comparerenderer and tells it the text is right-to-left. Any further options are passed to the fuzztest program, this is typically one of `--valgrind` or `--input`, or less frequently `--passes`, `--jobs` or `--status`. See `tests/fuzztest --help` for more information. This script runs `fuzztest` so that it corrupts every byte of the required TrueType and Graphite specific tables with a random value, but excludes OpenType and AAT related tables. It also imposes a runtime limit of 10 seconds (most test should compete in a fraction of a second) and a memory limit 200MiB, again a normal run should only use a tiny fraction of that. If the `comparerenderer` test segfaults, exceeds those limits or returns an error value it is logged to a file named on the following pattern: `fuzzfont-<font name>-<text name>.log`, which is written to the script's current directory.

3.8.5 Running fuzztest

```
tests/fuzztest --font=font [options] -- <test harness>
```

A multiprocess fuzz test framework that generates corrupt fonts to run a user supplied test harness against. This will check each byte in every table of a TTF or subset of it's tables if specified, by overwriting with a random or user specified value. Using the `--input` option it can also re-run tests using previous output logs as input, it will ignore any line that doesn't match the format of a fuzz line generated by the program, so such input fuzzes can be well annotated. It is this facility that is used to drive the `make fuzztest` regression check. By default this will try to ensure there is always one test harness running on each OS reported core at all times, the `--jobs` option can be used to limit this if need to limit the load. Unless told otherwise the program will display a status line inidcating the percentage of the tests run so far, the rate of testing and an estimated time and date for completion. Once the status has updated 4-5 times the estimate usually settles down to a frequently accurate estimate.

3.9 Adding fuzz tests

Fuzz regression test files are simply copies of the log generated by the `fuzzcomparerender` script. Once you have a log that generates a test case, you can edit it to produce a more targeted set of fuzz lines (a log can generate several different bugs). You should try to produce a minimal set of fuzz lines for that particular test case, however the more fuzz lines that casue the same bug the better.

The test case should be placed as follows:

```
tests/fuzz-tests/<font name>/<text name>/<bug description>.fuzz
```

where:

font name
The name of a font, minus the .ttf extension from tests/fonts.

text name
The name of a text file, minus .txt extension from tests/texts.

bug description
A short description of the bug, this cannot contain any ":" characters or other characters forbidden by Windows or Unix filename schemes. For `class::member` references use an _ e.g. `class_member`.

3.10 LibFuzzer based fuzzing

We now build a number of Clang libFuzzer test targets in `<src>/tests/fuzz-tests` whenever `fuzzer` is one of the sanitizers found in `GRAPHITE2_SANITIZERS`. There are a series of seed inputs in the `<src>/tests/fuzz-tests/libfuzzer-co` directory each one coresponding to one of the existing unit tests pairing a font with text and parameters. These are binary files which consist of:

Table 1: libFuzzer target seed file-format for fuzz targets

Length	**Input parameter**
<variable>	TrueType file of font
utint16 * N	Feature values (N = number of feature ids in the font)
uint32_t	Text Script ID
uint8_t	UTF encoding form uint size (1,2 or 4) see `gr_encform`
uint32_t	Text directionality see `gr_bidirtl`
uint_8_t[128]	Text in the UTF encoding form specified

Run as follows:

```
cd <src>/build
cp ../tests/fuzz-tests/libfuzz-corpus corpus
tests/fuzz-tests/gr-fuzzer-font corpus
```

To rerun a test just do:

```
tests/fuzz-tests/gr-fuzz-font crash-test-case-produced-by-fuzzer
```

See the libFuzzer documentation for more advanced options standard to all libFuzzer produced fuzzing targets.