Graphite Compiler Debug Files

Sharon Correll

Version 3

1 Introduction

The Graphite compiler outputs several text files that can be helpful in the process of developing a Graphite font. Specifically, if the renderer is giving unexpected results, the GDL programmer can examine these debugger files to see whether the compiler is generating the expected output. Together these files create a textual representation of the bulk of the information in the Graphite font tables.

The compiler generates these files when the /d switch is passed as the first argument.

2 Parse tree: dbg_parsetree.txt

This file shows a parse tree reflecting the syntax of the GDL program.

The line numbers refer not to the original source code but to the file '$_temp.gdl', an integrated source file resulting from running the preprocessor over the program and handling the #define and #include statements.

3 Glyph attributes: dbg_glyphattrs.txt

This file contains the value of all the glyph attributes for all the glyphs in the font.

The first section contains a list of the internal IDs assigned to all the glyph attributes. This list includes the system-defined glyph attributes (breakweight, directionality, *actualForPsuedo*) as well as the programmer-defined attributes.

The second section lists all the glyph attributes whose values are non-zero. Any unlisted attribute can be assumed to have the value 0.

The "*actualForPsuedo*" attribute is a special system-defined attribute that contains the ID number of the glyph that should be used for the final rendering of any pseudo-glyph. This attribute is zero for all glyphs that are not psuedo-glyphs.

All values that are glyphs metrics are based on the design units of the font (the font's em-square).

Implementational note: any gpoint attribute whose value is actually 0 will have a special value recorded, since the value 0 is used to mean "not defined." This special value is shown in the 'dbg_glyphattrs.txt' file as "zero."

Debugging Tip: the dbg_glyphattrs.txt file shows only glyph attributes, not glyph metrics. To include glyph metrics in the file, define a glyph attribute that is exactly equal to the glyph metric of interest.

4 Character/glyph mappings: dbg_cmap.txt

This file shows the effect of mapping a Unicode character to a glyph ID. This is equivalent to the mapping in the font's *cmap,* with the exception of pseudo-glyphs which are given a mapping generated by the compiler.

The first section of the file contains the character-to-glyph mapping, and the second section lists the reverse mapping. In both sections pseudo-glyphs are marked. All values are shown in hex.

Items marked "[auto-pseudo]" are pseudo-glyphs that are created by virtue of the fact that there are duplicate mappings to a glyph in the font's cmap. (This behavior can be turned off by the directive `AutoPseudo = false` in your GDL program; see sections 3.2.1 and 7.3 in the GDL documentation.) Any pseudo-glyphs that are mapping Unicode values not present in the font's cmap are labeled "[pseudo; not in cmap]".

For pseudo-glyphs, this file reflects only the initial mapping from Unicode character to glyph. The *actualForPseudo* glyph attribute contains the final glyph ID to use for rendering the pseudo-glyph; the value of this attribute is shown in the 'dbg_glyphattrs.txt' file.

The second section of the file shows two special glyphs that are used for internal processing. The item labeled [line-break] is a special glyph that is used to indicate the line-break "character"; it corresponds the "#" item in GDL rules. The item labeled [phantom] is a non-existent glyph that is created by the compiler to facilitate rule mapping.

5 Finite state machines: dbg_fsm.txt

This file shows the finite state machines that are generated for each pass. The finite state machines are used to match the input and determine which rules to fire.

For each pass, the file first indicates the column to which each glyph is assigned. Following that is the FSM table itself. The rows in the table are the states, and each element in the table is the state the machine transitions to when an item assigned to the given column is encountered. A value of zero means that no match occurred; the machine has "jammed."

The state number is followed by a count of the number of slots that have been matched at this state. Each state also contains an indication of which rules are in the process of being matched, and which have been fully matched (the "Success" states). When the FSM runs, it keeps a list of all the success states that have been encountered. When the machine "jams," the rules associated with these states are candidates for firing. The longest rules are tried first, in the order they appeared in the source code; the first one whose constraints succeed is the rule that is fired.

The rules for the pass are listed following the FSM table. Keep in mind that there is not a one-to-one correspondence between the rules in the source code and those generated by the compiler. Specifically if a source code rule contains optional items, there will be multiple corresponding rules generated by the compiler. The overall order of the rules, however, is unchanged. You may need to carefully compare the final rules shown in this file with the source code to determine the exact correspondences.

The rules may be shown in a slightly different form than they were expressed in the GDL program. In particular, some rules may have ANY classes inserted at the beginning of the context. When ANY classes are inserted, you will notice that the effect is that each rule in a given pass has the same number of items before the first underscore in the context. (This is an implementational trick to ensure that the engine has a consistent reference point from which to begin matching the

rules.) The rules as written in the ‘dbg_fsm.txt’ file are also abbreviated.

If you are having trouble determining why a rule is not firing as expected, this file may be helpful. On a piece of scrap paper, write out the underlying characters in the problematic text, and below them write the glyph IDs (from the ‘dbg_cmap.txt’ file) followed by the column assignments for each. Starting at the critical spot in the text, step through the FSM table, transitioning to the next state in the table for each successive column in your input. (The slots-matched counter should increase by one at each state.) Keep track of each successful rule you encounter. When you hit a zero element in the table, the machine has “jammed.” Compare the order and length of the successful rules to make sure that an undesirable rule isn’t being fired instead of the one you expected (use the ‘dbg_ruleprec.txt’ file to determine the precedence of the matched rules).

However, keep in mind that your list of input glyphs may not be in the state you expect, due to previous processing in this pass or a previous pass, and that may be the source of the problem. To do a really thorough debugging job, you may need to start with the beginning of the text at the first pass and work your way all the way through, modifying the glyph list as you go according to the rules that fire.

6 Rule precedence: dbg_ruleprec.txt

Since it is often difficult in a complicated pass with many rules to determine easily what the precedence for the rules are, the ‘dbg_ruleprec.txt’ is available to help with this task. This file lists the rules for each pass in the order of their precedence. When multiple rules are matched by the finite state machine, it is this precedence that determines which rule is actually fired.

Note that a single rule in the source code may correspond to multiple rules in this file, since rules are expanded by the compiler to account for optional items. The rule number (e.g., RULE 3.7) corresponds to the number in the ‘dbg_fsm.txt’ file and also in the ‘gr_xductn.log’ file that is output by the Graphite engine. The file indicates the line and file of the source code for each rule, and shows a simplified print-out of the rule.

7 Engine code: dbg_enginecode.txt

This file shows the internal code that is generated corresponding to the constraints and the effects of firing the rules. This internal code is in the form of a stack machine. See ‘Stack Machine Commands.doc’ for a description of the commands.

The file shows an abbreviated form of the rule, followed by the action code (the effects of running the rule) and the constraint code (the cumulative tests contained in the relevant –if– statements and in-rule tests).

Again, there is not a one-to-one correspondence between the rules as listed here and those in the source code (although the rule numbers are the same as what is shown in the ‘dbg_fsm.txt’ file). The numbers of the passes in this file, however, do correspond to the numbers in the source code. Pass 0 indicates the unnumbered pass, permitted when there is only one pass in a table.

Several of the engine code commands (PushGlyphAttr and PushAttToGlyphAttr) have the effect of pushing the values of glyph attributes on the stack. A translation between the glyph attribute internal ID number and the original name can be found in the ‘dbg_glyphattr.txt’ file.

Other commands (PutSubs and PutGlyph) have the effect of substituting one glyph for another. These commands use class numbers, which are specified in the ‘dbg_classes.txt’ file.

8 Substitution classes: dbg_classes.txt

This file shows classes that are used for substitution in the rules in the substitution table. Each class has a number, which corresponds to the numbers you can find in the dbg_enginecode.txt file, as arguments to the PutSubs and PutGlyph commands.

The classes are listed in two groups. The first group is labeled “LINEAR (OUTPUT) CLASSES.” These are the classes that appear in the right-hand or output side of the rules. These classes consist of simply a list of glyphs. They are organized into groups of 10 so that it is fairly easy for you to determine the index of any member of the class.

The second group of classes are labeled “INDEX (INPUT) CLASSES.” These classes appear in the left-hand or input side of the rules. You will notice that the list of glyphs for each class is sorted by glyph number and each glyph is followed by an index. When the given glyph is encountered in the input, the associated index is used to determine the corresponding glyph from the output class to substitute in place of the original glyph.

Note that some classes may be used in both the left- and right-hand sides of substitution rules, in which case they will appear in both the input and output class lists.

Example. Suppose you have a rule that replaces lower-case vowels (a, e, i, o, u) with upper-case equivalents (A, E, I, O, U). This rule might be represented in the dbg_enginecode.txt file something as follows:

```
Rule 1.0: clsLowerVowel  >  clsUpperVowel;

ACTIONS
PutSubs 0 10 6
Next
RetZero
```

Now suppose the font assigns glyphs as shown in the table below.

a	0x0021		A	0x0053
e	0x0025		E	0x0057
i	0x0029		I	0x005B
o	0x002F		O	0x0061
u	0x0035		U	0x0067

The dbg_classes.txt file would contain the following definitions for the classes:

```
LINEAR (OUTPUT) CLASSES

...

Class #6: clsUpperVowel
0:  0x0053  0x0057  0x005B  0x0061  0x0067

...

INDEXED (INPUT) CLASSES

...
```

```
Class #10: clsLowerVowel
0x0021 :  0    0x0025 : 1    0x0029 : 2    0x002F : 3    0x0035 : 4
```

Note that the second and third arguments of the PutSubs engine code command correspond to the numbers of the input and output classes, respectively. The glyph IDs are hypothetical, but this is in effect saying that “a” is assigned index 0, which corresponds to “A” in the clsUpperVowel class;“e” is assigned index 1, corresponding to “E”; etc. (In this case the input glyphs happen to be in sorted order, but keep in mind that this is not always the case.)

9 Revision History

1. 26-April-2000: First draft by Sharon Correll.
2. 26-Jan-2001. Revised to discuss ANY.
3. 26-Feb-2003. Added discussion of dbg_classes and dbg_ruleprec files.
4. 28-April-2003. Enhanced discussion of dbg_cmap file.

10 File Name

Compiler Debug Files.doc