SEAS SM84 SESSION REPORT

Session Title: GML on Micros to Mainframes

Speaker: R. Watt   Installation Code:

Session Number: 1.70   Date: 09.04.84 Time: 16.50

Session Chairman: A. Berglund

Number of persons present:

GML at Waterloo, From Micros to Mainframes

Roger W. Watt

Associate Director (Systems)
Department of Computing Services
University of Waterloo
Waterloo, Ontario, Canada

April, 1984

ABSTRACT

This paper is a report on work in progress in three different projects, all of which are in support of the use of "Generalized Markup Language" (GML) for the preparation and composition of documents.

The Waterloo GML Word Processor: A portable interactive what-you-see-is-what-you-get "word processor" with an internal knowledge of GML document components and the ability to transform the user's document into an external file that is encoded with the appropriate GML tags. (Riel Smit, Carl Durance, Eric Mackie; Computer Systems Group, University of Waterloo.)

The WATCOM GML Processor: A portable GML tag-and-layout processor whose layout-control language is GML itself, with no reliance on an underlying document-composition system. (Dave McKee and Jim Welch; WATCOM Systems Inc., Waterloo.)

The Waterloo SCRIPT/GML Processor: The addition of new Control Words and Set Symbols to Waterloo SCRIPT, and a total re-design of the tag-to-macro-invocation process, to improve GML-processing performance and simpify the task of the text programmer in defining a GML and writing the underlying SCRIPT macros. (Roger Watt, Bruce Uttley; Department of Computing Services, University of Waterloo.)

BACKGROUND

Automated document preparation and composition was first introduced at the University of Waterloo in 1969 using IBM's Administrative Terminal System under OS/MVT on an IBM 360 Model 75 computer. ATS was a marvel in its day, but pure drudgery by today's standards. It existed in a "closed" environment that required its own userids and managed its own file space for documents, and it forced the user to concentrate on low-level layout-control actions rather than on the document's information-content architecture.

In 1974, we acquired a derivative of IBM's CP/67 SCRIPT, which provided a tool that all users of the operating system could access simply by preparing an OS dataset and running a batch job. SCRIPT's macro-writing capability allowed us to create, package, and give the user sets of higher-level document-component commands; the first such set, SYSPUB, was initially created in 1975.

VM/CMS was introduced at Waterloo in 1973. It rapidly demonstrated its many benefits as a personal-computing productivity aid, and when the SCRIPT processor was implemented there, most ATS users started abandoning ATS in favour of SCRIPT.

Between 1978 and 1980, all users of ATS and other OS/MVT-based facilities moved to VM/CMS. The SCRIPT processor had evolved to become Waterloo SCRIPT, and now supported a wide variety of output devices, including typesetters and proportionally-spacing daisywheel terminals.

In 1981, two events occurred at Waterloo that form the foundation for the three projects described in this paper:

- o the first microcomputer-network environment for student computing was implemented; the current implementation of this network has now become known as The Waterloo PC Network, a product recently announced by IBM in the United States and Canada
- o work began on the implementation, in Waterloo SCRIPT, of an equivalent to the GML Starter Set described by IBM with its SCRIPT/VS DCF product.

The use of GML has proven to be a substantial productivity aid in enabling the document-preparation user to create documents without being burdened by the need to learn a layout-control-oriented low-level language. The use of microcomputer networks has proven to be a substantial productivity aid for student computing, and is now spreading to use by administrative and professional users as well. Providing increased support for both the GML user and the GML text programmer in a variety of computing environments has become an obvious necessity.

THE PROJECTS

Each of these three projects has a very different set of objectives. Those objectives, in turn, lead to different design peculiarities that make the three projects interesting from different points of view.

The Waterloo GML Word Processor and the WATCOM GML Processor are both written in a system-independent portable language called WSL, the Waterloo Systems Language, which produces a machine-independent target language for which code generators have been written for a variety of microcomputer, minicomputer, and mainframe hardware/software architectures. Included in these are the IBM PC, the Series/1, and VM/CMS. Code generators have also been developed for non-IBM architectures. Both of these projects result in support for the use of GML in a wide variety of machine/system environments, and they have, by intent, avoided reliance on any existing underlying document-composition application software.

Waterloo SCRIPT is written in IBM 360 Assembler Language, and is a full-function document-composition and text-programming language. Those attributes lend a different set of peculiarities ... while the GML language remains constant, the underlying layout-control capabilities require the skills of a SCRIPT-experienced text programmer when a layout must be created or modified.

The Waterloo GML Word Processor

The design objective of the Waterloo GML Word Processor is to produce a portable interactive "what you do, you see immediately" document-preparation tool that understands the document-component structure of the GML "Starter Set" tags defined by IBM, in a manner that can handle arbitrarily large documents, even when the processor is running on a microcomputer.

There are three major benefits to this design objective. Since the processor is written in the Waterloo Systems Language, it can be ported to a variety of environments; it will be and behave the same in every environment. The user of the GML Word Processor obtains immediate visual feedback that the document component just created is structurally correct and is what the user intended, and therefore can correct component and structural mistakes immediately. Because the processor works with GML Starter Set document components, it maintains the user's document in an internal structure that enables the processor, on user command, to produce an external file that is encoded with GML tags. This enables the user to process that GML-encoded file using other GML processors, and to transmit the file to other computing environments that provide GML processors.

Externals: The user of the GML Word Processor is provided with an interactive input/editing and help-menu capability driven from the keyboard of the workstation. On the IBM PC, the 10 Function Keys are used in "normal", "ALT", and "SHIFT" modes to enable selection of document-component types. Function Keys are also provided for various "word processing" functions, such as copying/moving blocks of text.

When the user needs to create a heading, for example, the user presses the Function Key that corresponds to the "heading" component type. Another Function Key enables the user to alter the level of the heading. Since the GML Word Processor keeps track of where the user is, in terms of the relationships between GML component types, it validates the level of the heading and will not let the user create a heading level that is structurally invalid (such as a level-two heading that has not been preceded by a level-one heading). It automatically (re)numbers the various heading levels throughout the document as the user adds/deletes headings. Heading-level numbers are displayed as protected fields, so that the user can alter a heading-level's text but not alter its number, since the heading-level number is not part of the user's information content. Other protected fields include the annotation-symbol text of items in ordered and un-ordered list structures.

The document can also be printed, on whatever hardcopy printing devices are provided by the user's machine environment.

Internals: The GML Word Processor maintains the user's document as a forward-and-backward linked list of document component types or "text blocks". Since the user always sees the information content in its "formatted" form, with automatic "line fill" and "word wrap" as text is added or deleted, the display of text lines is maintained as a linked list in which each display line is indexed to the component-type list of text blocks.

The GML Word Processor operates on three different types of files. The display-formatted version of the document is maintained in a disk file of type "WP". Layout-control specifications are maintained in disk files of type "WPL". The output that results from transforming the display-format version of a document into the corresponding GML-encoded version is written to a disk file of type "GML".

Layout Control: Currently, the extent to which the user can alter the layout of the document is minimal; layout control is provided by means of an editable file that allows specification of vertical/horizontal spacing and positioning parameters for each of the component types. From within the GML Word Processor, the use can issue a command that will cause a particular layout-control file to be used, instead of the default layout-control file.

Document Components: Currently, only a subset of the document components are implemented: heading (Hn), paragraph (P and PC), example (XMP), long quotation (LQ), and lists (OL, UL, and SL). However, a "transparent component" type is also supported. This allows the user to add a text block that the GML Word Processor will treat as a special "leave this exactly as entered" block. To remind the user that it was created as a transparent-block component, it is always displayed on the workstation screen in double intensity. By using transparent blocks, the user can add "foreign" objects to the document, including GML-encoded passages for component types not yet directly supported by the GML Word Processor.

Producing a GML-Encoded File: When the user issues the command to the GML Word Processor to produce a GML-encoded external file containing the document's information content, the GML Word Processor automatically generates front-matter tags to request a table of contents, the body tag, the tags and text from the user's document, and the end-of-document tag. Transparent blocks are copied "as entered" to the external file. that an external GML-encoded file be produced from the document. To enable the user to work on a large document as separate GML Word Processor files, and then generate separate GML-encoded external files, the GML Word Processor never generates the start-of-document and front-matter tags if the document begins with a transparent-block component, and never generates the end-of-document tags if the document ends with a transparent-block component.

Future Considerations: Efforts are currently being concentrated on improving the user interface from the workstation keyboard and display, since the current method of Function Key selection will become cumbersome as the set of supported document-component types expands to encompass the complete GML Starter Set. The concept of display-area function menus that "open up" when requested by the user is being considered. After that, consideration will also being given to improving the richness of the layout-control parameters, to enabling the GML Word Process to absorb an external GML-encoded file as input, so that the user can use the GML Word Processor to work on a document created in other GML-processing environments.

Summary: Results to date have proven the merit of the concept. The GML Word Processor is a good productivity aid for the task of creating the types of documents for which the GML Starter Set was designed. The ability to generate a GML-encoded equivalent of a document allows the result to be processed by GML processors with richer layout-control and output-device capabilities.

The WATCOM GML Processor

The design objective of the WATCOM GML Processor is to produce a portable GML processor that supports the GML Starter Set and provides an easy-to-use layout-control language that is itself GML-encoded.

There are three major benefits to this design objective. Since the processor is written in the Waterloo Systems Language, it can be code-generated for a variety of hardware/system architectures; it will be and behave the same in every environment. It avoids any dependence on the existence of an underlying (possibly complex, and probably not portable) document-composition or text-formatting language processor for its formatting abilities and layout-control specifications. As a consequence, it's performance characteristics are totally under the control and capabilities of its developers, thereby avoiding any resource-intensive processing characteristics that may be inherent in an underlying text formatter.

Externals: The user prepares a GML-encoded document-source file, by using whatever editor the user prefers from among those provided in the computing environment being used. To process that "GML" file, the user invokes the WATCOM GML Processor by issuing the "GML" command. The user specifies the GML filename as the operand, and can specify any combination of three different types of options: the name of a "LAYOUT" control file to be used by the processor, the type of output device for which the composed result is to be produced, and a "verbose processing" parameter that gives the user visual feedback about the status of processing by displaying the document headings at the user's workstation as the processor progresses through the GML source file.

Internals: The WATCOM GML Processor is written in WSL. The WSL code includes a built-in table of GML tagnames, and a built-in table of tag attribute names and value lists. It also contains a built-in "state table", so that the processor knows, at all points during processing of the user's GML file, which tags are now valid and which tags may not now be specified. This allows the processor to enforce the structural integrity of the GML document-component types, and to do so in a manner that enables it to generate meaningful diagnostic messages when the user attempts to (mis)use GML tags in inappropriate places.

Most of the "application processing function" details are also written in WSL, as "subroutines".

Layout Control: Implementing a document-composition processor for the GML Starter Set without the services of an underlying document-composition processor also requires implementing all of the otherwise-unnecessary text-formatting capabilities as part of the GML processor. This approach therefore requires that some simple-to-use mechanism be provided by which the GML user can create new layouts and/or modify existing layouts without having to acquire the skill-level of an experienced application-development "text programmer".

The WATCOM GML Processor supports a special set of non-standard GML tags for defining layout-control parameters for the Starter Set tags.

The nature of the layout-control tags is shown in Figure 1. A number of "global formatting parameter" tags also exist within the "LAYOUT/eLAYOUT" block, to control things such as the size of the output page and the placement of text and running-title banners at the tops and bottoms of pages, and the extent of widow-prevention. This allows a new layout to be created as a pseudo-GML file. The

```
:LAYOUT
   ...
:fig
          indent=5
          skip=2
          spacing=1
:hl
          indent1=0
          page=yes
          skip=3
          spacing=1
          font=3
          style=h
          number=prop
          position=left
   ...
:SAVE
:eLAYOUT
```

Figure 1: The Layout-Control Tags

"SAVE" tag causes the internal-storage representation of the layout-control parameters to be written to an external file of type "LAYOUT", for future use. The result is a pre-compiled layout-control file that can be loaded rapidly when the user invokes the WATCOM GML Processor.

The combination of these two capabilities allows the GML user to alter the effects of an existing layout by encoding only the needed re-specifications as a "LAYOUT/eLAYOUT" block at at the beginning of the GML file for a document, and, if the user wishes, to also "SAVE" the result as a new LAYOUT file.

Future Considerations: The capabilities of the layout-control tag set are currently under evaluation, since they do not yet provide a rich enough set of parameters to control all of the necessary aspects associated with a layout.

Some mechanism needs to be devised so that the WATCOM GML user can create new tags, since the application-processing-function code is WSL-source code that is only available to the processor's developers.

Also, output-device drivers for more attractive device types with richer capabilities need to be developed, since the only device currently supported is the "line" printer. Proportionally-spacing daisywheel terminals and the Xerox 2700 laser printer are currently being considered.

Summary: Efforts to date have already resulted in considerable success. The WATCOM GML Processor provides fast and machine-independent processing support for documents encoded with the tags of the GML Starter Set, and an easy-to-use mechanism for creating/modifying document layouts.

The Waterloo SCRIPT/GML Processor

Implementing the GML Starter Set by using an existing rich-function document-composition processor with a powerful set of text-programming capabilities as the underlying base eliminates many of the development tasks outlined with the above two projects. Whenever a new text-programming facility has been needed to support the GML implementation, the required facility has been encoded into the Waterloo SCRIPT processor, where it is also available to users and text programmers involved in non-GML applications. Device-driver support was added to Waterloo SCRIPT some years ago to enable documents to be composed for for a wide variety of output devices, including typesetters and proportionally-spacing daisy-wheel terminals. Recently, support has also been added for the Xerox 2700 laser printer, and a number of other output devices (such as the Xerox 87/9700 laser printers and a new 60-pages/minute "Delphax" printer) are being considered. Every advancement of Waterloo SCRIPT provides immediate benefit to the users of the Waterloo SCRIPT GML, as well.

The first full implementation of the GML Starter Set in Waterloo SCRIPT was completed late in 1982, as a set of SCRIPT-coded application-processing-function macros. Since then, the Waterloo SCRIPT version of the GML Starter Set has been extended considerably, with some additional tags and many additional attributes and values for the tags. There are now 17 different layouts that support the GML Starter Set, in a variety of paper, report, thesis, and manual styles, as well as 7 different "non-standard" layouts for memos, letters, minutes of meetings, contract agreements, and policy-and-procedure manuals.

Tools for the GML Text Programmer: Since the implementation of Waterloo SCRIPT's GML is in the Waterloo SCRIPT macro language, the services of an experienced Waterloo-SCRIPT text programmer are required in order to develop new GML tag sets or add tags to the existing sets, and to develop new layouts or modify existing layouts. The design objective of one recently-initiated Waterloo SCRIPT project is to provide the text programmer with a set of new SCRIPT control words that will simplify the process of defining a Generalized Markup Language tag set, and will simplify the nature of the coding needed in the corresponding application-processing-function SCRIPT macros.

There are three major benefits to this design objective. The new control words allow the SCRIPT processor to perform much of the validity-checking and error-diagnosis work to enforce the document-structure integrity intended by the text programmer; previously, that could only be achieved by the text programmer, by writing code in the underlying SCRIPT macros. The internal data structures that Waterloo SCRIPT creates from these new control words also allow it to by-pass most of the resource-intensive processing needed to initialize a macro's local Set Symbol dictionary before invoking the macro; the new code initializes far fewer local symbols, and also initializes some special-purpose local symbols that reduce the text programmer's need to resort to global symbols. Both of the above also reduce the skill level needed by the text programmer who is not yet thoroughly experienced at the intricacies of macro-writing application development.

Externals: Two new SCRIPT control words have been added. They are DEFINE TAG (.dt) and DEFINE ATTRIBUTE (.da), The DEFINE TAG control word is as follows:

```
.DT tagname ADD    macroname tagoptions
            CHANGE macroname
            ON/OFF
            ZERO
            DELETE/PRINT
     *      DELETE/PRINT
```

The "tagoptions" specify whether the tag causes "continued text" if it occurs inline, whether tagtext is required or not allowed, whether the tag may only be followed by another tag (a minimal form of "state table"), and whether the tag is to be processed or ignored during document composition. The DEFINE ATTRIBUTE control word is as follows:

```
.DA tagname attname (a) (b)
    *       attname
    *       *
where
(a) is any combination of:
    ON  OFF  UPPERCASE  REQUIRED
and
(b) is one of:
    AUTOMATIC 'value'
    LENGTH max
    RANGE min max <default>
    VALUE value <DEFAULT>
    VALUE value <USE 'value' <DEFAULT>>
    ANY <'value'>
```

The ON/OFF operands determine whether the atttibute is to be ignored during document composition (useful for attributes that provide information that can only be used with some output devices).

These new control words allow the GML text programmer to define the tags, attributes, and value lists that comprise a "tag set". SCRIPT builds an internal datastructure from these control words that allows it to perform most of the user-error validation and to invoke the underlying macros in a manner that by-passes much of the normal macro-invocation initialization processing.

```
.dt FIG add @FIGTAB attributes texterror
.da *   DEPTH range 0 200 0
.da *   FONT off upper
.da *   *    value MONO default
.da *   *    value TEXT
.da *   ID length 6
.da *   NAME automatic 'Figure'
```

Figure 2: Part of the definition for the FIG tag

Internals: Figure 2 shows part of the definition for the FIG tag, using these two new control words.

The coding two support these new control words and implement a new "tag scanner" comprises close to 4000 lines of Assembler code. The control-word code builds linked-list data structures. The primary list is the "tagblock" list; each tag defined occupies one block in this list, and points to a secondary list that contains the attributes for that tag; each attribute, in turn, points to a list of values valid for that attribute. The PRINT option of the DEFINE TAG control word allows the internal data structures to be displayed ... a useful debugging tool during the construction of a tag set, and a useful documentation aid as well.

From the internal data structures constructed by SCRIPT from these control-word definitions, SCRIPT is able to detect and diagnose the use of invalid attribute names, invalid values, missing attributes, and so on. It is also able to detect an attribute specified without a value and assign the default value for that attribute (if it has one) or generate an error message (if no default).

These data structures also enable SCRIPT to generate an internal local symbol dictionary for the macro that a tag is to invoke, and then invoke the macro directly, thus totally by-passing the "normal" process of macro initialization and invocation. The AUTO, DEFAULT, and specified attributes are provided to the macro as local symbols of the form "*attname". New "special" local symbols have also been introduced, to facilitate the task of the text programmer:

*_TAG	the name of the tag that caused the macro to be invoked (for example, both the FIG and TAB tags invoke the same macro)
*_N	the "usage count" for this tag (eliminates the need to maintain "counter" variables as global symbols)

Future Considerations: The DEFINE TAG and DEFINE ATTRIBUTE control words are being re-evaluated to include some mechanism for including a "state table" of tag-and-text relationships in the definition. Consideration is also being given to externalizing as much as possible of the "formatting environment" parameters, so that users of GML could achieve as much tailoring as possible without the need to first become SCRIPT-experienced text programmers.

Summary: The objectives of this current Waterloo SCRIPT project have been achieved, and work progresses on recoding the macros for the "Starter Set" tag set and the seven non-standard tag sets to exploit the new control words.

Substantial performance benefits have been realized, as well. "Test" documentation, including the Waterloo SCRIPT GML User's Guide, have experienced a 15-25% reduction in CPU-time processing.

The error diagnostics have been enhanced, also. There are 15 new SCRIPT-produced error messages, which demonstrates a considerable reduction in the things for which the GML text programmer formerly had to "program" to detect in the SCRIPT macros.