$2.50

MUUGlines

June 1996

Volume 8, Number 8

This Month's Meeting

To celebrate the arrival of summer temperatures (at last), the MUUG regular meeting will be in the form of a barbeque held at Assiniboine Park. Join us at the south BBQ pits, adjacent to the parking lot that's closest to the Countess of Dufferin locomotive. As there may not be sufficient picnic tables for all of us, consider bringing lawn chairs or blankets to sit on.

The barbeque will start at 6:30, an hour before our regular meeting time to give us more time to soak up the warm sunshine (rather optimistic of me, perhaps!). The date is the usual second Tuesday of the month, June 11. If you attend, bring your choice of 'barbequeable' – MUUG will provide the soft drinks, condiments and other trimmings including dessert!

Other Tidbits

This is the last meeting of the MUUG 'season', as we take a break for the summer months. Our next meeting will be Tuesday, September 10th. The next issue of this newsletter will arrive in your mailbox in advance of that, to remind you and to let you know about all the exciting programs planned for the upcoming year.

The MUUG SIGs will also resume in September, with the opening meeting scheduled for the 17th. Stay tuned for details on SIG presentations.

To contact the MUUG board for membership information or anything else, send email to `board@muug.mb.ca`. Don't forget, we have a home page too, found at http://www.muug.mb.ca/, where you can find all kinds of information, including details of upcoming and past meetings and presentations, and references of interest related to them.

To contact the newsletter editor (and I know that you want to shower him with article submissions over the summer months), email `editor@muug.mb.ca`.

Have a great summer!

Form & Function - Using Forms On the Web

Part 2 - CGI Programs

By Gilbert Detillieux, Computer Science, University of Manitoba

In the last part, we looked at how fill-out forms are coded in HTML. We also looked at how the contents of forms are submitted by means of a MIME-format message.

In this article we'll look at the Common Gateway Interface (CGI), which lets you set up scripts to do server-side processing. Such scripts are usually the best way to process form input.

HTTP Transactions

Before getting right into the details, it helps to understand how your web browser (an HTTP client) talks to the HTTP server. HTTP transfers are based on transactions. For each document to be fetched, the client looks at the URL, decides on the protocol to use, the server to contact, and the document to fetch. The client then opens a connection to the server, issues a request, and receives a reply. The request will consist of a single command line, possibly followed by MIME-format headers, a blank line, and possibly followed by an encoded message body.

In the simplest case, the request consists of a single GET command, followed by a blank line:

```
GET /~gedetil/form/form.html HTTP/1.0
```

The server's response has a similar structure: status line, MIME-format headers, blank line, and message body. For a typical GET request, the server responds with the requested document:

```
HTTP/1.0 200 Document follows
Date: Mon, 25 Mar 1996 20:32:23 GMT
Server: NCSA/1.5
Content-type: text/html
Last-modified: Mon, 25 Mar 1996 18:48:57 GMT
Content-length: 1010
... document content ...
```

All of these transactions are normally transparent to the user, except when an error occurs. However, when dealing with CGI programs, it's important to know about the structure of transactions.

The Common Gateway Interface

Since an HTTP server only delivers static documents of various known types, it is limited in the information it can provide. To make it capable of delivering other types of information, particularly information that is obtained dynamically, possibly based on user input, the server calls external programs. The Common Gateway Interface (or CGI) was developed to allow such programs to communicate with the server in a standard way.

The server still handles the communication with the client, but when a request is received for a CGI program the server invokes the program and passes it the information it needs. Since the server has already read in the client's command and any MIME headers, this information is passed along via environment variables.

Continued on page 2

The REQUEST_METHOD environment variable will be set to either GET or POST, depending on the method used to submit the request. In the case of a GET request, the QUERY_STRING variable will contain the URL-encoded string that was submitted (which was sent as part of the URL on the request. In the case of a POST request, CONTENT_TYPE and CONTENT_LENGTH will contain the values of the corresponding MIME headers. CONTENT_TYPE should be set to "application/x-www-form-urlencoded" for a normal form submission. CONTENT_LENGTH will indicate how many bytes of data must be read from the standard input, in order to obtain the URL-encoded form data.

So much for the input side. For output, the CGI program is more in control. The server will usually worry about sending the initial status line, but the CGI program must output the MIME headers, a blank line, then a message body. At the very least, a "Content-type:" header must be sent, so that the client will know what sort of message is to follow. (The server doesn't know what the CGI program will send as output, so it's up to the program to say so itself.) The message body can be a simple text message, or a more elaborate document, either pulled from a file or generated on the fly.

A Trivial Example

The following example, a UNIX shell script, is about as simple as a CGI program can get:

```
#!/bin/sh

# Start with MIME-format message header:
echo "Content-type: text/plain"
echo ""

# Process information, and output
message:
date
```

When the server calls this script, it simply outputs the Content-type header, and a blank line, then it invokes the UNIX date command to output a single line of text, containing the system date.

As far as the HTTP transactions are concerned, the request is the same as for a normal document file, except that the CGI program name is given:

```
GET/~gedetil/setupwww/cgi-bin/date.shHTTP/1.0
```

The server's reply will consist of a few lines of its own, followed by the CGI program's output, verbatim:

```
HTTP/1.0 200 Document follows
Date: Mon, 25 Mar 1996 23:09:06 GMT
Server: NCSA/1.5
Content-type: text/plain

Mon Mar 25 17:09:06 CST 1996
```

If we wanted to pass input to the script, we could have added it to the GET request, by adding a string such as "?query-string" on to the end of the file name. Everything after the "?" character would have been passed to the script in the QUERY_STRING environment variable.

Using the Querysh Wrapper

The tough part of handling input in scripts, whether it's a simple query string or the content of a complex form, is decoding all of the URL-encoded data. This is why such CGI programs are often written in a programming language like C or in a scripting language like perl, both of which provide the needed string processing capabilities. However, we can get by with simple UNIX shell scripts if we leave the nitty-gritty job of decoding the data to a simple wrapper program, such as Querysh. [http://www.cs.umanitoba.ca/~gedetil/setupwww/querysh/]

Querysh will take care of determining whether a GET or POST method was used, and the URL-encoded data from the appropriate source, decode it, split the fields, and set the value of each field in an environment variable (which is easy to process in a script).

For example, given our earlier sample form, the URL-encoded string would be something like this:

```
status=New&membnum=&fullname=Gilbert+
Detillieux&address=123+Mulberry+Lane
%0D%0AWinnipeg%2C+MB%0D%0A
R3R+3R3&phone=%28204%29+555-1212
&email=gedetil@cs.umanitoba.ca
```

Querysh would decode this and set the following variables for the script to use:

```
QSH_Fields="QSH_status
QSH_membnum QSH_fullname
QSH_address QSH_phone QSH_email"
QSH_status="New"
QSH_membnum=""
QSH_fullname="Gilbert Detillieux"
QSH_address="123 Mulberry
Lane[CRLF]Winnipeg, MB[CRLF]R3R 3R3"
QSH_phone="(204) 555-1212"
QSH_email="gedetil@cs.umanitoba.ca"
```

Here, "[CRLF]" is used to denote an actual CR/LF pair of characters in the string. With the form data in this format, we're all set to write a script.

Processing the Sample Form

Using querysh to do the grunt-work, our CGI script takes on the following basic structure:

```
#!/usr/local/etc/httpd/querysh
#?/bin/sh

if [ -z "$QSH_Fields" ]
then
        # No form fields given — send
empty form
        cat <<!
Content-type: text/html

... HTML for an empty form ...
!
        exit
fi

... process the form data ...

cat <<!
Content-type: text/html

... HTML for a status message ...
!
```

The first couple of lines are to let the system know where to find querysh so it can be run, and to let querysh know

what shell it should then run to interpret the rest of the script. The next part of the script checks to see if there are any fields that were set. (It's possible that a blank form was submitted or that the script was invoked directly, without any form data being passed.) In that case, a sensible thing to do might be to simply return an HTML document containing a blank form to be filled out.

The script then goes on to do some processing based on the submitted data, and finally it should output something back to the client such as a document, a status message or some output based on the request.

Error Checking

It's important to remember that once your CGI script is set up, it is accessible by anyone on the web. The input it receives may not even come from a form you set up, but may have been entered directly by someone else, or using a modified form. For this reason, it's important to carefully check all input fields before using them.

This is particularly important in shell scripts, where this input may be passed along to other commands, or interpreted by the shell itself. A malicious user could pass along input containing special characters for the shell, and possibly even commands to be run by this shell. So, be careful how you use input data in your script, and make sure you check all fields thoroughly before you make use of them.

Further Reading

A very good description of the Common Gateway Interface can be found at NCSA. [http://hoohoo.ncsa.uiuc.edu/cgi/overview.html] Ian Graham's Introduction to HTML also has a good description of CGIs. [http://www.utirc.utoronto.ca/HTMLdocs/NewHTML/serv-cgi-bin.html]

Doom on Linux

by Steve VanDevender <stevev@jcomm.uoregon.edu>

Last time, we left off the discussion of getting DOOM to run on Linux with "Prerequisites and Installation". We carry on with sound and libraries:

If you want sound effects, you will need:

> The Linux sound driver (Voxware) 2.90 or above (cat /dev/sndstat to find out your version)
>
> A 16-bit soundcard (such as the Sound Blaster 16, Pro Audio Spectrum 16, or Gravis Ultrasound). You may be able to get your 8-bit soundcard to work with the package doom_16to8bit_snd.tgz — see the section titled "I have an 8-bit soundcard, how can I get sound?".

Here's the output of ldd (which gives dynamic linking information for executables) for linuxsdoom, linuxxoom and sndserver. If your shared libraries are older than these versions and you're having problems, you may want to consider upgrading them. Note that if you want to run the X version, you must have the X11R5 libX11.so.3.1.0 available somewhere on your system, since X11R6 uses a different major revision number for its shared libraries.

```
$ ldd linuxsdoom
libvga.so.1 (DLL Jump 1.1pl8) => /usr/local/lib/libvga.so.1
libc.so.4 (DLL Jump 4.5pl26) => /lib/libc.so.4.5.26
$ ldd linuxxdoom
libXt.so.3 (DLL Jump 3.1) => /usr/X386/lib/libXt.so.3.1.0
libX11.so.3 (DLL Jump 3.1) => /usr/X386/lib/libX11.so.3.1.0
libc.so.4 (DLL Jump 4.5pl26) => /lib/libc.so.4.5.26
$ ldd sndserver
libm.so.4 (DLL Jump 4.5pl26) => /lib/libm.so.4.5.26
libc.so.4 (DLL Jump 4.5pl26) => /lib/libc.so.4.5.26
```

Installation instructions

Obtain the file linux-doom-1.8.tar.gz from the FTP site listed above. You can also obtain the individual parts of the original distribution, but these instructions are geared toward the complete package. If you have DOOM for MS-DOS, then you can use doom1.wad from the shareware version, doom.wad from the registered version, or doom2.wad from DOOM II.

First, extract linux-doom-1.8.tar.gz with the command

```
zcat linux-doom-1.8.tar.gz | tar xvf -
```

(it will extract its files into a new directory doom-1.8). Do this as root so that the correct permissions will be placed on the SVGAlib DOOM executable, or see "Cannot get I/O permissions" below to find out how to set the permissions manually. If you already have registered DOOM or DOOM II, and want to use the WAD files that come with those, remove or rename doom1.wad and copy doom.wad or doom2.wad into the doom-1.8 directory.

Continued on page 4

If you want to use the SVGAlib version of DOOM (I highly recommend it), you will either need to have SVGAlib 1.1.8 or above, or install SVGAlib 1.2.0 from the kit. If you are installing the kit version, copy libvga.so.1.2.0 to /usr/local/lib, make sure /usr/local/lib is in /etc/ld.so.conf, and run ldconfig to make it available. Also copy README.config to /usr/local/lib/libvga.config and edit it to match your system's VGA card and mouse settings. If you already have SVGAlib and use a MouseSystems-type mouse, you may want to install the kit version or apply the patch given in the README file to fix a problem with SVGAlib handling of MouseSystems-type mice.

If you don't want sound effects or don't meet the prerequisites above for sound hardware/drivers, remove or rename the file "sndserver".

When you're ready to try it out, change to a text-mode virtual console and give the command

```
sdoom -warp 1 1
```

If you want to play in X, get into X and give the command

```
xdoom -warp 1 1
```

Make sure your mouse is in the "No Name" window that DOOM comes up in, or click on the window if your window manager uses click-to-focus input focusing.

Press ESC to get the DOOM main menu, if you want to select something other than the default starting location and difficulty level. Your mouse may not work right away. See "Configuration, Tweaks, and Tricks" for more information about setting your mouse type and customizing the keyboard and mouse.

If you would like to install the binaries in one directory and keep the doom*.wad file in another, you can set the DOOMWADDIR environment variable to the name of the directory where you are keeping doom*.wad so that linux[xs]doom can find it. You can also make a symbolic link to a copy of doom*.wad in a mounted MS-DOS filesystem if you don't want to duplicate doom*.wad into a Linux filesystem (although the game will start much faster if doom*.wad is in an ext2 filesystem). If you can't get DOOMWADDIR to work, see the section titled "DOOMWADDIR doesn't work as advertised" below.

H. Peter Anvin <hpa@ahab.eecs.nwu.edu> has written a shell script that acts as a front-end for Linux DOOM as part of a package including the DOOM executables and the sndcvt program for 8-bit sound support. The package is available by anonymous FTP from eecs.nwu.edu in pub/linux/doom/doom.tgz . The script installs into /usr/local/bin and runs DOOM from /usr/local/lib/doom.

Read the README.linux[xs] and README.dos included in linux-doom-1.8.tar.gz. They contain most of the information here (in briefer form), complete documentation of the game commands and gameplay features, and also explain id's policy towards the Linux port — it exists because "Linux gives [David Taylor] a woody" and is not officially supported by id. Linux DOOM does not support all MS-DOS doom features (notably music and modem or IPX network play) and probably never will.

Problems and solutions

Most of the headers in this section contain key words from an error message output by DOOM when something doesn't work.

Cannot get I/O permissions (SVGA)

SVGAlib applications must run as root or setuid to root in order to obtain permission to write to the VGA I/O registers. Run SVGA DOOM as root, or say "chown root linuxsdoom; chomd 4755 linuxsdoom" to make SVGA DOOM setuid to root so it can be run by non-root users.

DISPLAY=(null) (X)

This means you are trying to run X DOOM without X. You need to start your X server before you can start X DOOM. If you try to set the environment variable DISPLAY without having an X server running, or the variable is set wrongly, then instead you'll probably get a message containing "Cannot connect to server".

If you have to install X, you will need to install the SVGA version of X or one of the versions tailored for various accelerated graphics chipsets, and run the display in 256-color mode (not monochrome or 16-color mode, or the 16- or 32-bit color modes available in XFree86 3.1). DOOM will not work with a display whose color depth is anything other than 8 bits.

Error: Demo is from different game version

This comes up if you give just the command "sdoom" or "xdoom" and wait for the game to play the prerecorded demos in the registered WAD files. At this time, the Linux DOOM executables think they're for version 1.8, while most likely you have WAD files for version 1.9 (if you're a real DOOM fiend) or 1.666 (if you're a wimp and haven't upgraded yet). If you are using the shareware WAD file, then it's still version 1.8 and actually will play its demos.

The way around this, of course, is to invoke sdoom or xdoom with the "-warp" command-line option to jump directly into the game and skip the demos:

```
sdoom -warp 1 1
```

This will place you in Episode 1, Level 1 of registered DOOM. If you are using DOOM II, then you give only one number for the level you want to jump into.

Next issue: more problems and their solutions to get you into the game!