CHAPTER 21

Animating Text

OVERVIEW

In this chapter, we will outline several ways to animate text using the non-parsed header (NPH) method of controlling the interaction between the server and the browser. Surely, newer technologies, such as Java, or plug-ins, such as Shockwave, are better suited for complex animation. However, the ability to use NPH scripts in creative ways is an important part of any CGI programmer's box of tools. NPH scripts add another dimension to multimedia and can make your site more diverse. They can also be used when the big guns of Java or Shockwave are not appropriate. Because they use Perl, NPH scripts are more accessible to the majority of Web administrators, who are not necessarily advanced programmers.

Non-parsed header scripts are used when we want to bypass the server. Usually, when we use the line

```
print "Content-type: text/plain\n\n";
```

we count on the fact that the server that executes the script will fill in the rest of the HTTP protocol lines, such as the "200 OK" status codes, the date and time, and other information defined in the protocol. With NPH scripts, we must generate those HTTP protocol messages internally, bypassing server parsing. In other words, we output directly to the browser without Web server intervention.

Most of the information that you might pass to the browser is optional. However, you should at least return the MIME content type of the data, the HTTP protocol revision, the status of the program, and the server name and version.

By bypassing the server, you can communicate directly with the browser. As long as the browser listens, you can continue to feed it more information. In this way, you can use cell animation to create an animated series of text or images.

However, the use of NPH scripts requires that all scripts begin with the characters `nph-`. This is the convention used by servers to recognize an NPH script. If you rename the script, the server will not know to treat it as an NPH script and instead will run it as a normal script.

Because you are bypassing the server, you need to be careful that your NPH scripts do not run forever, because they will run to completion whether or not the browser has already moved to another page. If the script loops infinitely, it may never know to stop and may eat up your server resources.

Because all these animation scripts are fairly short, we have decided to group them in this chapter and discuss them one at a time. The following scripts are meant to build upon one another, exploring NPH-based animation from the simplest examples to the more complicated ones.

The first script, **nph-countdown1.0.cgi**, uses an NPH script to produce an animated countdown from 10 to 0. The client sees this countdown in real time in the browser window. Figure 21.1 shows the beginning, middle, and end frames of a sample session for **nph-countdown1.0.cgi**.

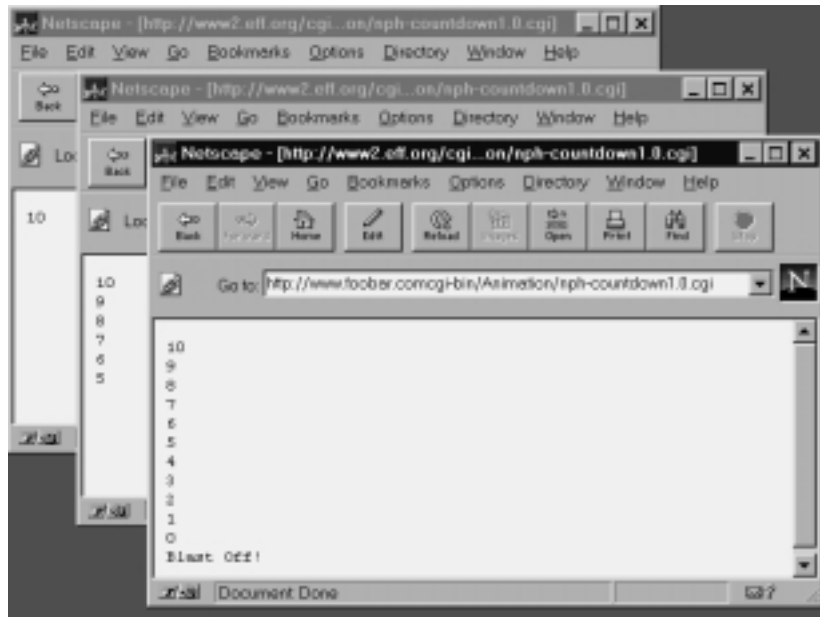

Figure 21.1 *nph-countdown 1.0 example.*

nph-countdown2.0.cgi takes the countdown to a new level. It counts up from 1 to 5, and instead of displaying the countdown as a chain of numbers, it erases the previous number and writes the new number where the old one used to be. Figure 21.2 shows the beginning, middle, and end frames of a sample session for **nph-countdown2.0.cgi**.

Similarly, **nph-text_animator_1.0.cgi** uses the NPH method combined with array manipulation to produce an animated series of words. Figure 21.3 shows the beginning, middle, and end frames of a sample session for **nph-text_animator_1.0.cgi**.

As **nph-countdown2.0.cgi** did for **nph-countdown1.0.cgi**, so **nph-text_animator_2.0.cgi** does for **nph-text_animator_1.0.cgi**. This little script animates the words by replacing each word with the next one so that a visible list does not develop. Figure 21.4 shows the beginning, middle, and end frames of a sample session for **nph-text_animator_2.0.cgi**.

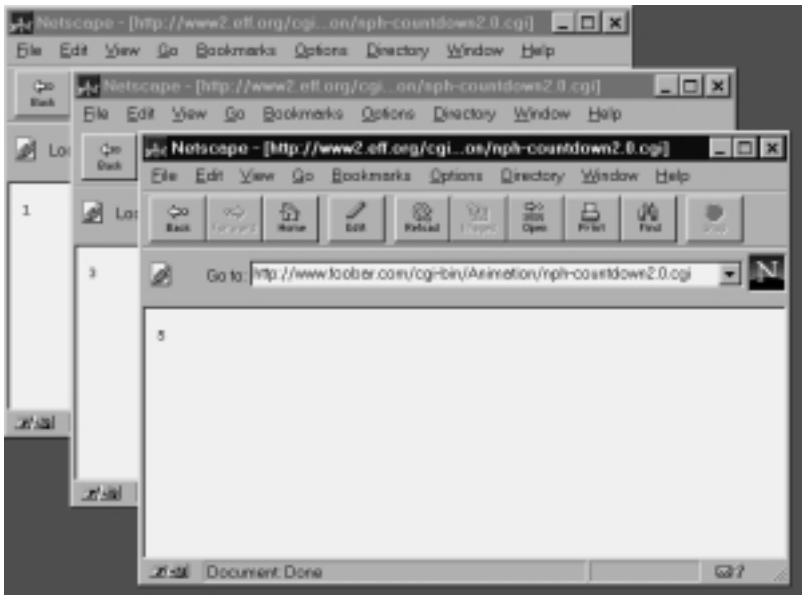

Figure 21.2 *nph-countdown 2.0 example.*

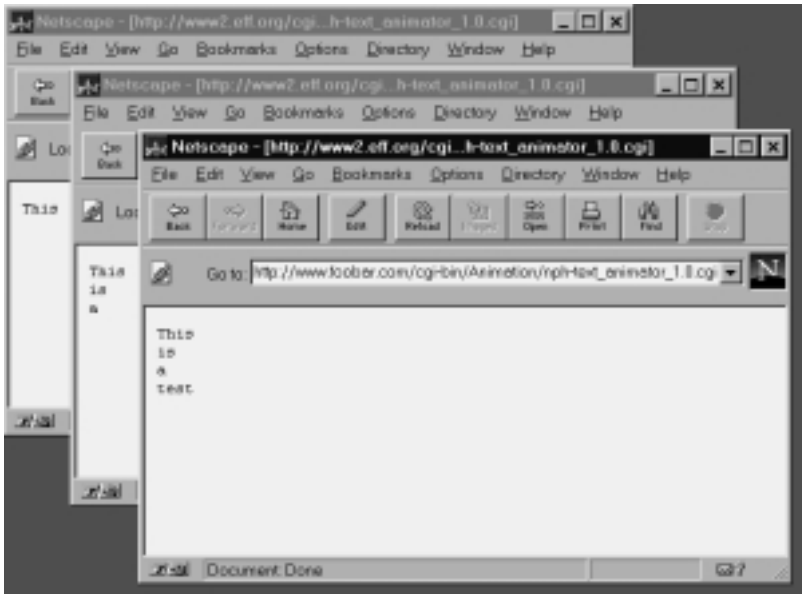

Figure 21.3 *nph-text_animator 1.0 example.*

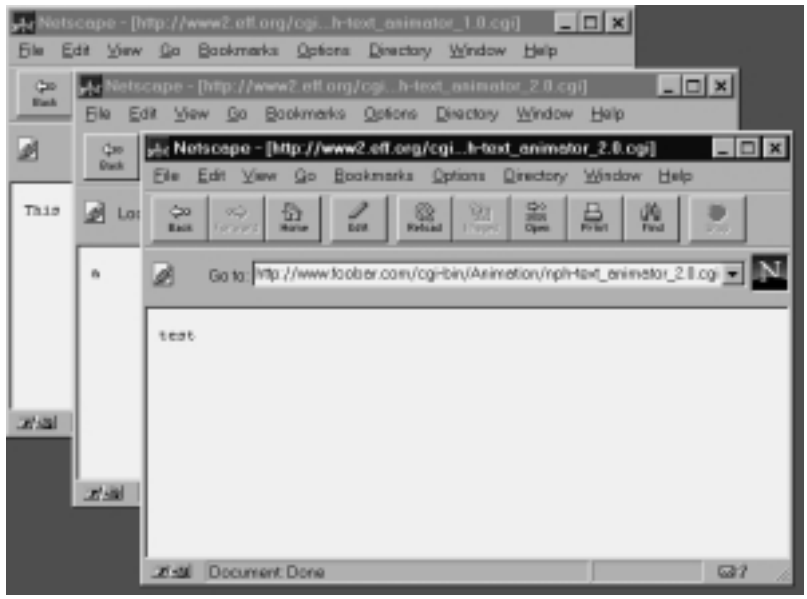

Figure 21.4 *nph-text_animator 2.0 example.*

nph-text_animator_3.0.cgi takes the animation one step further by integrating HTML output. Thus, we use HTML to affect the display of the animated words instead of simply outputting them as plain text. Figure 21.5 shows the beginning, middle, and end frames of a sample session for **nph-text_animator_3.0.cgi**.

In **nph-text_animator_4.0.cgi** we use the HTML <PRE> tag to introduce horizontal movement into the animation.

The <PRE> tag may not be supported by all browsers, but it is included here to show the method. You also can generate horizontal space by using transparent images that are invisible when displayed.

Figure 21.6 shows the beginning, middle, and end frames of a sample session for **nph-text_animator_4.0.cgi**.

Figure 21.5 *nph-text_animator 3.0 example.*

Figure 21.6 *nph-text_animator 4.0 example.*

Finally, in **nph-text_animator_5.0.cgi** we use the `srand` function to produce an HTML animation that uses random colors, size, and horizontal and vertical spacing. Figure 21.7 shows the beginning, middle, and end frames of a sample session for **nph-text_animator_5.0.cgi**.

Figure 21.7 *nph-text_animator 5.0 example.*

INSTALLATION AND USAGE

Each of these scripts should be placed in a directory from which the Web server is allowed to read and execute CGI scripts. Each script should be readable and executable by the Web server. Figure 21.8 depicts the directory structure with the correct permissions for each of the scripts.

Then simply point your Web browser to the script, sit back, and watch. The scripts are referenced like any other script, using the following example syntax:

```
http://www.foobar.com/cgi-bin/Animation/nph-countdown1.0.cgi
```

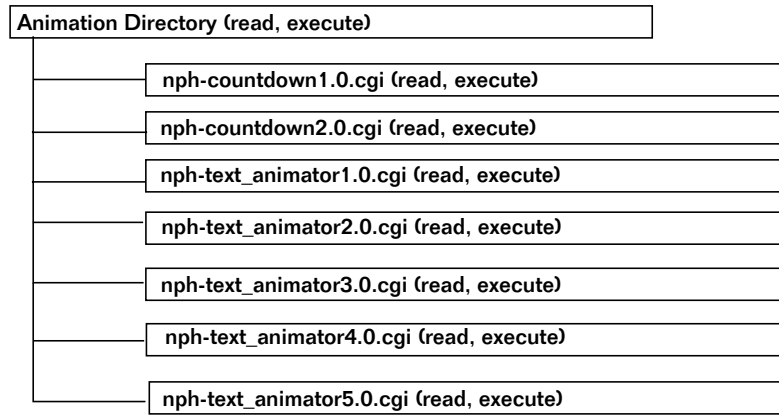

Figure 21.8 *Directory structure and permissions.*

Design Discussion

The logic is fairly similar for all the animation scripts. The script first outputs the HTTP header information. Next, it defines any animation variables, and then it creates the cell animation by looping. This logic is depicted in Figure 21.9.

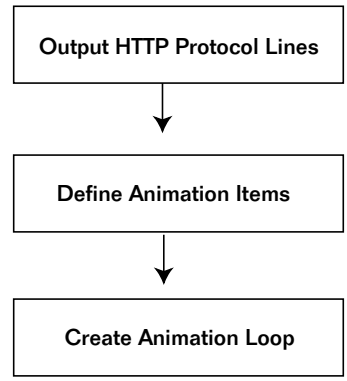

Figure 21.9 *Animation logic.*

nph-countdown1.0.cgi

nph-countdown1.0.cgi begins by starting the Perl interpreter and printing a complete HTTP header so that it can bypass the server and talk directly to the client.

```
!/usr/local/bin/perl
print "$ENV{'SERVER_PROTOCOL'} 200 OK\n";
print "Server: $ENV{'SERVER_SOFTWARE'}\n";
print "Content-type: text/plain\n\n";
```

Next, it tells the Perl interpreter to forgo buffering. We want to print straight to output so that the animation will be as smooth as possible. If we set `$|` to zero, the Perl interpreter would buffer the output. If we set it to 1, the script will "flush" the buffer continuously.

```
$| = 1;
```

Then the script creates the logic of the cell animation. First, it sets `$loop = 10`. Then it begins counting down (`$loop–`) one number at a time until it reaches zero (`$loop >= 0`). For each number, it prints the value to the browser and pauses (`sleeps`) for one second. The client sees a real-time countdown in the browser window.

```
for ($loop = 10; $loop >= 0; $loop–)
{
print "$loop\n";
sleep (1);
}
```

Finally, the script prints "Blast Off!" and exits.

```
print "Blast Off!\n";
exit (0);
```

nph-countdown2.0.cgi

As before, the Perl interpreter is started and the script outputs the standard HTTP header while bypassing the Perl buffer.

```
#!/usr/local/bin/perl
$| = 1;
print "$ENV{'SERVER_PROTOCOL'} 200 OK\n";
print "Server: $ENV{'SERVER_SOFTWARE'}\n";
```

However, this time the script tells the Web browser that it should not keep adding the numbers to a displayed list. Instead, the browser should continually replace the displayed number. Thus, instead of simply outputting a mime type of plain text or HTML, the script lets the browser know that it will be replacing each character and that every time the browser comes across –ARandomString\n\n, it knows that it is time to replace an old character with a new one.

```
print "Content-type: multipart/x-mixed-
replace;boundary=ARandomString\n\n";
```

The script begins sending characters by sending the flag string.

```
print "--ARandomString\n";
```

It then sets `$loop` to 1 and begins counting upward one number at a time (`$loop++`) until it hits five (`$loop <= 5`).

```
for ($loop = 1; $loop <= 5; $loop++)
  {
```

Meanwhile, the script prints the value of `$loop` in plain text, pauses for a second, and then tells the browser to overwrite the old value of `$loop` with the new value by sending the new character flag.

```
print "Content-type: text/plain\n\n";
print "$loop\n";
sleep (1);
print "\n--ARandomString\n";
}
```

When the script is finished counting, it quits.

```
exit (0);
```

nph-text_animator_1.0.cgi

As before, the script starts the Perl interpreter, tells it to bypass the buffer, and prints a complete HTTP header so that it can bypass the server's buffer.

```
#!/usr/local/bin/perl
$| = 1;
print "$ENV{'SERVER_PROTOCOL'} 200 OK\n";
print "Server: $ENV{'SERVER_SOFTWARE'}\n";
print "Content-type: text/plain\n\n";
```

Next, a list of words to be animated is defined and set equal to `@words`.

```
@words = ("This", "is", "a", "test");
```

Then the script sets `$loop` to zero (because all arrays start with zero as their first value) and begins incrementing `$loop` by 1 (`$loop++`) until `$loop` is equivalent to the number of words in `@words` (`$loop <= @words;`).

```
for ($loop = 0; $loop <= @words; $loop++)
  {
```

For every value of `$loop`, the script prints the value in `@words` that corresponds to the value of `$loop`. So when `$loop` equals zero, the script prints the first word in the `@words` array; when `$loop` equals 1, it prints the second word, and so on. Then the script pauses a second and prints another word until it is finished with all the words.

```
print "$words[$loop]\n";
sleep (1);
}
exit (0);
```

nph-text_animator_2.0.cgi

The script uses the Perl interpreter, bypassing the Perl buffer, and, as before, prints a complete HTTP header. However, as it did in **nph-counter2.0.cgi**,

the script lets the browser know that it will be replacing each character and that every time the browser comes across the tag `--ARandomString\n\n`, it will know that it is time to replace an old word with a new one.

```
#!/usr/local/bin/perl
$| = 1;
print "$ENV{'SERVER_PROTOCOL'} 200 OK\n";
print "Server: $ENV{'SERVER_SOFTWARE'}\n";
print "Content-type: multipart/x-mixed-
replace;boundary=ARandomString\n\n";
```

The script then alerts the browser by sending the boundary flag.

```
print "--ARandomString\n";
```

Next, `@words` is filled with the list of words to be animated.

```
@words = ("This", "is", "a", "test");
```

Then `$loop` is set to zero and the script begins incrementing `$loop` by 1 until `$loop` is equivalent to the number of words in `@words`.

```
for ($loop = 0; $loop <= @words; $loop++)
{
```

For every value of `$loop`, the script prints the value in `@words` that corresponds to the value of `$loop`. It then pauses a second and prints the flag that notifies the browser to replace the next word with the current one until all the words have been displayed and the script exits.

```
print "Content-type: text/plain\n\n";
print "$words[$loop]\n";
sleep (1);
print "\n--ARandomString\n";
}
exit (0);
```

nph-text_animator_3.0.cgi

nph-text_animator_3.0.cgi begins as all the others have begun in this chapter.

```
#!/usr/local/bin/perl
$| = 1;
print "$ENV{'SERVER_PROTOCOL'} 200 OK\n";
print "Server: $ENV{'SERVER_SOFTWARE'}\n";
print "Content-type: multipart/x-mixed-
replace;boundary=ARandomString\n\n";
print "--ARandomString\n";
```

As before, `@words` is filled with the list of words to be animated. However, the `is` and `test` elements are modified with HTML `<CENTER>` tags; when displayed as HTML, they reflect the HTML formatting.

```
@words = ("This", "<CENTER>is</CENTER>", "a",
          "<CENTER>test</CENTER>");
```

Then the script sets `$loop` to zero and begins incrementing `$loop` by 1 until it is equivalent to the number of words in `@words`.

```
for ($loop = 0; $loop <= @words; $loop++)
{
```

Finally, the script displays the words as usual. The standard HTML header and footer are included, because we are now displaying an HTML document rather than just a text document.

```
print "Content-type: text/html\n\n";
print "<HTML><HEAD><TITLE>nph-demo</TITLE></HEAD><BODY>";
print "<H1>";
print "$words[$loop]\n";
print "</H1>";
print "</BODY></HTML>";
sleep (2);
print "\n--ARandomString\n";
}
exit (0);
```

nph-text_animator_4.0.cgi

Again, the script begins as usual.

```
#!/usr/local/bin/perl
```

```
$| = 1;
print "$ENV{'SERVER_PROTOCOL'} 200 OK\n";
print "Server: $ENV{'SERVER_SOFTWARE'}\n";
print "Content-type: multipart/x-mixed-
replace;boundary=ARandomString\n\n";
print "--ARandomString\n";
```

This time, each word in `@words` is offset by blank spaces so that we achieve the appearance of movement.

```
@words = ("This",
          "   is",
          "     a",
          "       test",
          "            of",
          "           animated",
          "              text");
```

As before, the script loops the array and prints each horizontally affected word until all the words have been displayed.

```
for ($loop = 0; $loop <= @words; $loop++)
{
print "Content-type: text/html\n\n";
print "<HTML><HEAD><TITLE>nph-demo</TITLE></HEAD><BODY>";
print "<PRE>";
print "$words[$loop]\n";
print "</PRE>";
print "</BODY></HTML>";
sleep (1);
print "\n--ARandomString\n";
}
exit (0);
```

nph-text_animator_5.0.cgi

nph-text_animator_5.0.cgi prints the standard script header that we have seen throughout this chapter.

```
#!/usr/local/bin/perl
$| = 1;
print "$ENV{'SERVER_PROTOCOL'} 200 OK\n";
```

```
print "Server: $ENV{'SERVER_SOFTWARE'}\n";
print "Content-type: multipart/x-mixed-
replace;boundary=ARandomString\n\n";
print "—ARandomString\n";
```

Next, the script revs up the Perl randomizer so that it can generate some random numbers.

```
srand;
```

`@words` is defined with the list of words to be animated.

```
@words = ("This",  "is",  "a",  "test");
```

`@spaces` is also defined so that the script can randomly offset the words horizontally when displayed.

```
@spaces = ("", " ", "   ", "   ", "      ", "       ",
           "       ");
```

Similarly, `@font_sizes` is defined with tags that the script uses to randomly size words for non-Netscape browsers.

```
@font_sizes = ("H1", "H2", "H3", "H4", "H5", "H6");
```

Next, `@font_colors` is defined to generate random text colors.

```
@font_colors = ("FFFFFF", "000000", "00CCCC", "CC1523",
                "B916CC", "1F0DCC", "11CCC1", "07CC24",
                "C1CC10", "#D98719", "#D9D919",
                "#5C3317", "#2F4F2F", "#FF2400");
```

Then `vertical_spacings` is defined to randomly place displayed words vertically.

```
@vertical_spacings = (
"",
"<FONT COLOR = \"C9C3CC\">.<P></FONT>",
"<FONT COLOR = \"C9C3CC\">.<P>.<P></FONT>",
"<FONT COLOR = \"C9C3CC\">.<P>.<P>.<P></FONT>",
```

```
"<FONT COLOR = \"C9C3CC\">.<P>.<P>.<P>.<P></FONT>",
"<FONT COLOR = \"C9C3CC\">.<P>.<P>.<P>.<P>.<P></FONT>",
"<FONT COLOR = \"C9C3CC\">.<P>.<P>.<P>.<P>.<P>.<P>.<P></FONT>",
"<FONT COLOR = \"C9C3CC\">.<P>.<P>.<P>.<P>.<P>.<P>.<P>.<P></FONT>",
"<FONT COLOR =
\"C9C3CC\">.<P>.<P>.<P>.<P>.<P>.<P>.<P>.<P>.<P></FONT>",
"<FONT COLOR
=\"C9C3CC\">.<P>.<P>.<P>.<P>.<P>.<P>.<P>.<P>.<P>.<P></FONT>",
"<FONT COLOR = \"C9C3CC\">.<P>.<P>.<P>.<P>.<P>.<P></FONT>");
```

The font color C9C3CC corresponds to the basic gray background of a browser. Thus, the periods will not be visible (as long as your browser isn't defaulted to a color other than gray) and will functionally offset the vertical placement. Alternatively, you could create invisible (transparent) images.

Finally, `@netscape_font_sizes` is defined to generate random font sizes for Netscape browsers.

```
@netscape_font_sizes = ("1", "2", "3", "4", "5", "6",
                        "7", "8", "9", "10");
```

Then the script sets `$loop` to zero and increments `$loop` by 1 until `$loop` is equivalent to the number of words in `@words`.

```
for ($loop = 0; $loop <= 20; $loop++)
{
```

As before, the script prints the words. But this time, it randomly affects each word in a number of ways. The script generates a random value for the word, its vertical and horizontal spaces, its size, and its color.

The script first chooses a random (`rand`) integer (`int`) between zero and the number of elements in each array and then sets the variable to the returned value. This value is used to reference the array again. If `$font_color` is set to 3, the random font color will be referenced by plugging 3 into the array, returning the value of `"00CCCC,"` according to the array definition.

```
$word = int(rand(@words));
$space = int(rand(@spaces));
$font_size = int(rand(@font_sizes));
$netscape_font_size = int(rand(@netscape_font_sizes));
$font_color = int(rand(@font_colors));
$vertical_spacing = int(rand(@vertical_spacings));
print "Content-type: text/html\n\n";
print "<HTML><HEAD><TITLE>nph-demo</TITLE></HEAD>";
print "<BODY TEXT = \"$font_colors[$font_color]\">";
print "<PRE><B>";
print "$vertical_spacings[$vertical_spacing]";
```

In the special case of font size, the script determines whether the browser understands `<FONT SIZE>` tags. If it does, the script displays the text as `<FONT SIZE>`. Otherwise, it displays it with `<H*>` settings.

`$ENV{'HTTP_USER_AGENT'}` is the environment variable that keeps track of the kind of browser accessing the CGI script. `=~ /^Mozilla/` means that if the value is something (`=~`) that begins with `Mozilla`, the script should use the `<FONT SIZE>` tag.

```
if ($ENV{'HTTP_USER_AGENT'} =~ /^Mozilla/)
  {
  print "<FONT SIZE =
        \"$netscape_font_sizes[$netscape_font_size]\">";
  }
else
  {
  print "<$font_sizes[$font_size]>\n";
  }
print "$spaces[$space]$words[$word]\n";
```

The script then adds the closing font tags.

```
if ($ENV{'HTTP_USER_AGENT'} =~ /^Mozilla/)
  {
  print "</FONT>";
  }
else
  {
  print "</$font_sizes[$font_size]>\n";
  }
print "</B></PRE>";
print "</BODY></HTML>";
```

Next, it prints the flag that notifies the browser to replace the next word with the current one.

```
sleep (0);
print "\n--ARandomString\n";
}
```

Finally, the script prints a final version of the sentence so that if viewers are confused by the randomness they can see the whole sentence.

```
print "\n--ARandomString\n";
print "Content-type: text/html\n\n";
print "<HTML><HEAD><TITLE>nph-demo</TITLE></HEAD>";
print "<BODY>";
print "<CENTER><H2><BLINK>";
```

For every word in the list of words, the script prints the word.

```
foreach $word (@words)
{
print "$word\n";
}
print "</BLINK></H2></CENTER></BODY></HTML>";
exit (0);
```