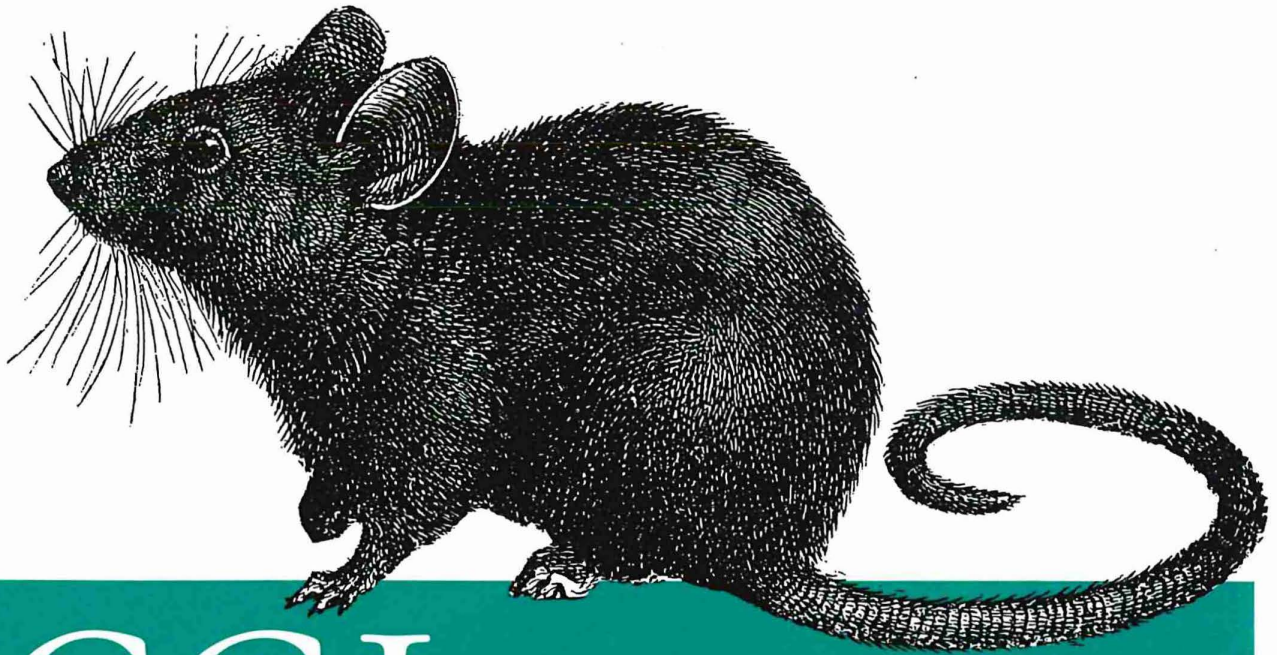


On-the-Spot Information

**Examples
in Perl**



CGI Programming

on the World Wide Web

ANUTSHELL®



HANDBOOK

Shishir Gundavaram

O'Reilly & Associates, Inc.

FreeWheel, Ex. 1082
FreeWheel v. AlmondNet
IPR2024-00419

CGI Programming *on the World Wide Web*

CGI Programming

on the World Wide Web

Shishir Gundavaram

O'Reilly & Associates, Inc.

Bonn • Cambridge • Paris • Sebastopol • Tokyo

CGI Programming on the World Wide Web

by Shishir Gundavaram

Copyright © 1996 O'Reilly & Associates, Inc. All rights reserved.
Printed in the United States of America.

Published by O'Reilly & Associates, Inc., 103 Morris Street, Suite A, Sebastopol, CA 95472.

Editors: Andy Oram and Linda Mui

Production Editor: Jane Ellin

Printing History:

March 1996: First Edition

Nutshell Handbook and the Nutshell Handbook logo are registered trademarks of O'Reilly & Associates, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly & Associates, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.



This book is printed on acid-free paper with 85% recycled content, 15% post-consumer waste. O'Reilly & Associates is committed to using paper with the highest recycled content available consistent with high quality.

ISBN: 1-56592-168-2

[4/96]

output, and deleted. If you like, you can use the algorithm in Chapter 3 to return the GIF image in small pieces.

Figure 6-6 shows the image created by this script.

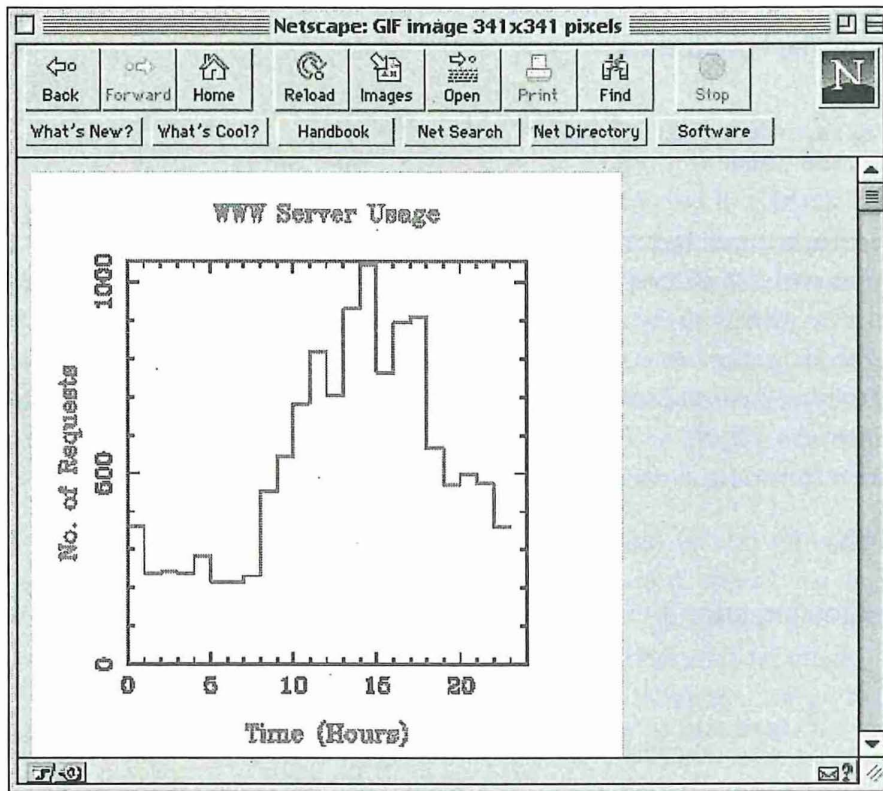


Figure 6-6. *pgperl* graph

Animation

Although Java is being touted as the best way to do animation on the Web, you can also write CGI programs to produce animation. There are two mechanisms for creating animation: client pull and server push. In client pull, a new HTTP connection is opened every time a document is requested. In server push, however, the connection is kept open until all the data is received by the client. That is the main difference between the two mechanisms. As a result, you can have an animation in an otherwise static document by using the HTML `<IMG>` tag to access the CGI program instead of a URL to an image, as introduced in the "Inserting Multiple Dynamic Images" section at the beginning of this chapter.

Client pull requires a special directive either in the HTML document header or as a part of the CGI program's HTTP response. This directive instructs the client to retrieve a specified document after a certain amount of time. In other words, the

client opens a new connection to the server for each updated image (see Figure 6-7).

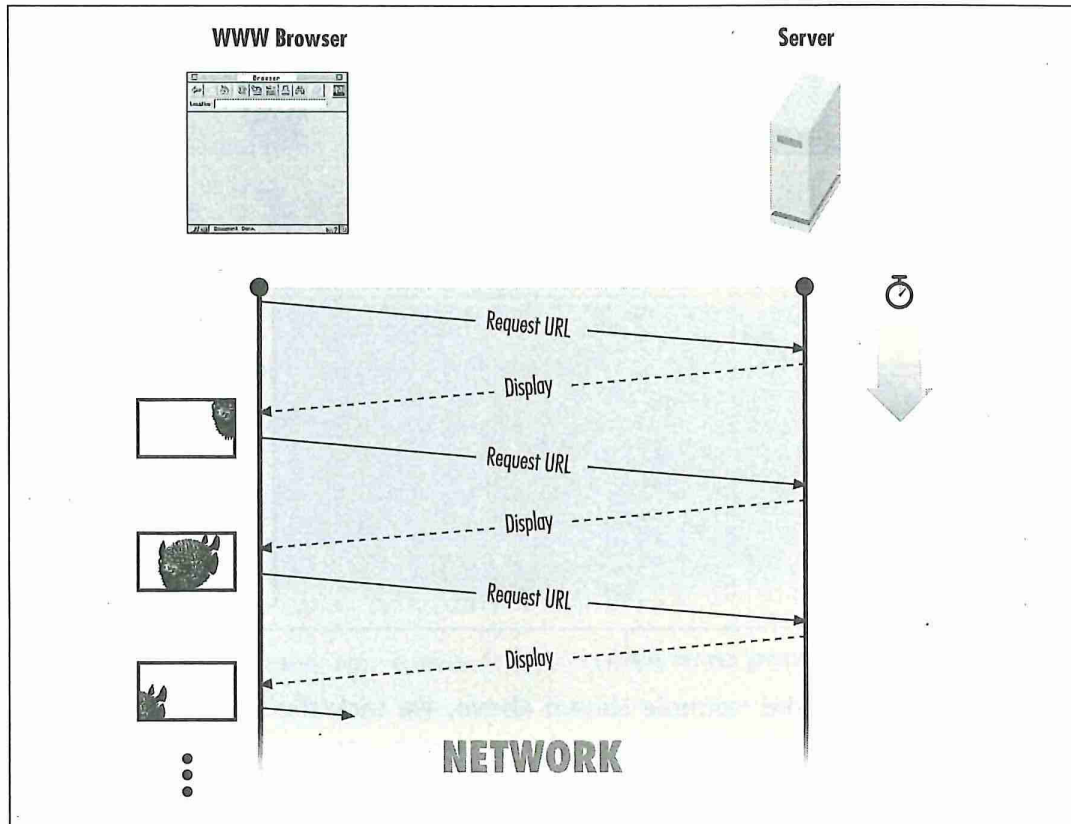


Figure 6-7. Animation using client pull

Server push involves sending packets of data to the client periodically, as shown in Figure 6-8. The HTTP connection between the client and the server is kept open indefinitely. Server push can be implemented in a CGI program through the use of the multipart/x-mixed-replace MIME type.

Both client pull and server push are supported only by Netscape Navigator (version 1.1 and higher) and Internet Explorer.

Client Pull

Here is a simple example of an HTML document that displays the time continuously:

```
<META HTTP-EQUIV="Refresh" CONTENT=5>
<!--#echo var="DATE_LOCAL"-->
```

Animation depends on updating the browser's window at regular intervals with new material from the server. Browsers provide a way to update their windows

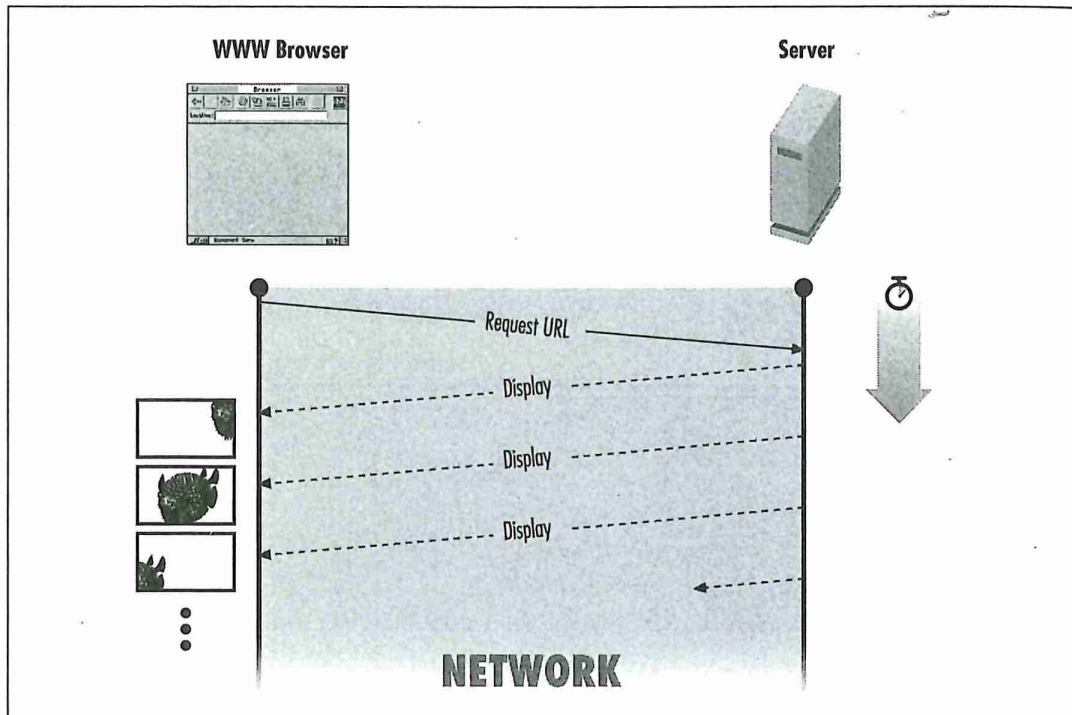


Figure 6-8. Animation using server push

called *refreshing*. In the example shown above, we trick the browser into issuing its refresh command every five seconds, so that it retrieves the document. The document simply uses server side includes to display the current time. (See Chapter 5 for more information on Server Side Includes.)

The META tag is part of the HTML 3.0 specification used to simulate HTTP response headers in HTML documents. In this case, it is used to simulate the "Refresh:" HTTP header with a delay of five seconds.

The "Refresh:" header is non-repeating; it does not load the document repeatedly. However, in this example, "Refresh:" is specified on each retrieval, creating a continuous display.

Here is an example of a CGI program that performs the same operation as the previous HTML code:

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

$delay = 5;
$date = "/bin/date";

print "Refresh: ", $delay, "\n";
print "Content-type: text/plain", "\n\n";
print ` $date `;

exit(0);
```

Remember, SSI directives cannot be included in a CGI program. So, the *date* command is used to output the current date and time.

Now, let's look at the directive used to load a different document after a specified time:

```
<META HTTP-EQUIV="Refresh" CONTENT="5; URL=http://your.machine/name.html">
```

This example loads the file specified by the URL after five seconds. If the file *name.html* does not contain another "Refresh:" header, there is no animation, because "Refresh:" is non-repeating. The corresponding CGI statement would be:

```
print "Refresh: 5; URL=http://your.machine/name.html", "\n";
```

As a final example of client pull, here's a CGI program that loads a document with a random fortune message every ten seconds.

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

$fortune = "/usr/local/bin/fortune";
$refresh_time = 10;

print "Refresh: ", $refresh_time, "\n";
print "Content-type: text/plain", "\n\n";

print "Here is another fortune...", "\n";
print ` $fortune `;

exit(0);
```

This is a repeating document, because a "Refresh:" header is specified every time the program is executed. The program uses the UNIX *fortune* command, which generates a random fortune each time it is involved.

Server Push

Server push animations can be created using the multipart/x-mixed-replace MIME type. The "replace" indicates that each data packet replaces the previous data packet. As a result, you can make smooth animations. Here is the format in which this MIME type is used:

```
Content-type: multipart/x-mixed-replace;boundary=End

--End
Content-type: image/gif

Image #1

--End
Content-type: image/gif

Image #2
```

```
--End
Content-type: image/gif
```

```
Image #3
```

```
--End--
```

In the first Content-type declaration, we declare the multipart/x-mixed-replace content types and establish “End” as the boundary string. We then repeatedly display new images (declaring new content types of image/gif), ending each image with the “--End” string. The result is that the images are displayed one after another.

Let’s look at an example that utilizes the server push mechanism.

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

$| = 1;
$webmaster = "shishir@bu.edu";

$boundary_string = "\n" . "--End" . "\n";
$end_of_data = "\n" . "--End--" . "\n";

$delay_time = 1;
```

First, we define the boundary strings that need to be sent to the client. We also set the delay time between images—in this case, one second.

```
@image_list = ("image_1.gif",
               "image_2.gif",
               "image_3.gif",
               "image_4.gif",
               "image_5.gif");
```

All of the images that will be used in the animation are stored in the *@image_list* array. In this simple example, we use only 5 images.

```
$browser = $ENV{'HTTP_USER_AGENT'};

if ($browser =~ m/^Mozilla/(1\.[^0][2-9])#) {
    print "Content-type: multipart/x-mixed-replace;boundary=End", "\n";
```

The name of the client browser is obtained using the environment variable HTTP_USER_AGENT. If the browser is Netscape version 1.1 or higher, the multipart MIME type is sent to it, along with the initial boundary string. (Netscape uses “Mozilla” as its user agent string.)

```
    for ($loop=0; $loop < scalar (@image_list); $loop++) {
        &open_and_display_GIF ($image_list[$loop]);
        print $boundary_string;
        sleep ($delay_time);
    }

    print $end_of_data;
```

A loop is used to iterate through the *image_list* array. Each image is displayed using the *open_and_display_GIF* subroutine. A boundary is then sent to the client, and the program proceeds to sleep for the specified amount of time. It is important to print the boundary after the image *and* before the *sleep* command to ensure that the server “pushes” the entire image to the client. The process is repeated for all the images in the array. Finally, the terminating boundary string is sent to the client.

```

    } else {
    &open_and_display_GIF ($image_list[0]);
    }

    exit(0);

```

If the browser is not Netscape version 1.1 or higher, only the first image stored in the array is displayed.

```

sub open_and_display_GIF
{
    local ($file) = @_ ;
    local ($content_length) = (stat($file))[7];

    print "Content-type: image/gif", "\n";
    print "Content-length: ", $content_length, "\n\n";

    if (open (FILE, "<" . $file)) {
        while (<FILE>) {
            print;
        }

        close (FILE);
    } else {
        &return_error (500, "File Access Error", "Cannot open graphic file
$file");
    }
}

```

This routine should be very familiar to you. First, it sends the image/gif MIME type, along with the length of the image. Then, the image is printed to standard output.

One final note: If you are using an NCSA server, it is better to create the CGI server push animation program as a non-parsed header (“nph”) script, as described in Chapter 3, *Output from the Common Gateway Interface*. That way the server will not parse the HTTP headers, and instead will send the information directly to the client. The main advantage of this is reduced “jerkiness” in the animation. Just to refresh your memory, you need to name the script with an “nph-” prefix, and the first lines that are output from your script should be:

```

print "HTTP/1.0 200 OK", "\n";
print "Content-type: multipart/x-mixed-replace;boundary=End", "\n";

```

In This Chapter:

- *Animated Clock*
- *Game of Concentration*
- *Introduction to Imagemaps*
- *Calendar Manager*

11

Advanced and Creative CGI Applications

In this final chapter of practical advice and code, we will look at three applications: a simple animated clock, the game of Concentration, and a Calendar Manager. All three of these examples utilize a combination of the various techniques presented up to this point.

Animated Clock

This example creates the effect of an animated digital clock by repeatedly generating dynamic GIF images and sending them to the browser using server push (see the discussion in Chapter 6, *Hypermedia Documents*). You can use the techniques presented in this example to create CGI programs that continuously display such information as system load averages, stock prices, or sports scores. However, programs like these can heavily tax the host machine, although they may be fun and entertaining. So you should use them only if there is an absolute need to do so.

To summarize the method used in this example: First we check that the browser is Netscape Navigator, version 1.1 or higher. That's because Netscape is the only browser that currently supports server push. We then generate a new image every few seconds and send it to the client. To create the image, we'll use the same *gd* extension to Perl that we showed in Chapter 6. We have to send the data as a special MIME type called `multipart/x-mixed-replace` so that the client replaces each old image with the new one. Following the MIME standard, we send an "--End--" string at the end of each image. Here is the code:

```
#!/usr/local/bin/perl5
```

```
use GD;
```

```

$| = 1;

$font_length = 8;
$font_height = 16;

$boundary_string = "\n" . "--End" . "\n";
$end_of_data = "\n" . "--End--" . "\n";

```

The program turns output buffering off by setting Perl's `$|` variable. The boundary strings for server push are defined.

```

$delay_time = 5;
$max_updates = 10;

```

The `$delay_time` variable reflects the time between image updates. The maximum number of updates performed by this program is set to 10. The reason for setting these variables is so that the user does not tax the system by watching the updates for an infinite amount of time.

```

print "HTTP/1.0 200 OK", "\n";

```

This CGI script outputs the complete HTTP header (see Chapter 3, *Output from the Common Gateway Interface*). Server push animation appears smooth only if buffering is turned off and a complete header is output.

```

$browser = $ENV{'HTTP_USER_AGENT'};

if ($browser =~ m#^Mozilla/(1\.[^0]|[2-9])#) {
    print "Content-type: multipart/x-mixed-replace;boundary=End", "\n";
    print $boundary_string;
}

```

This *if* block runs if the browser is Netscape Navigator, version 1.1 or higher.

```

    for ($loop=0; $loop < $max_updates; $loop++) {
        &display_time ();
        print $boundary_string;
        sleep ($delay_time);
    }

```

The `display_time` subroutine determines the current time, creates an image, outputs the image/gif MIME type, and displays the image. The boundary string is sent to the browser indicating the end of image data. The `sleep` command then waits for the specified amount of time.

```

        &display_time ("end");
        print $end_of_data;

```

Once the loop is terminated, the `display_time` subroutine is called one final time, with an argument. The “end” argument instructs the subroutine to draw the clock in a different way—as we will soon see. Finally, the last boundary string is sent to the browser.

```

    } else {
        &display_time ("end");
    }

```

```

}

exit(0);

```

If the browser does not support server push, the *display_time* subroutine is called just once to display a static image of the current time.

The *display_time* subroutine does most of the work for the program:

```

sub display_time
{
    local ($status) = @_ ;
    local ($seconds, $minutes, $hour, $ampm, $time, $time_length,
          $x, $y, $image, $black, $color);

    print "Content-type: image/gif", "\n\n";

    ($seconds, $minutes, $hour) = localtime (time);

    if ($hour > 12) {
        $hour -= 12;
        $ampm = "pm";
    } else {
        $ampm = "am";
    }

    if ($hour == 0) {
        $hour = 12;
    }

    $time = sprintf ("%02d:%02d:%02d %s", $hour, $minutes, $seconds,
                    $ampm);

```

The current time is formatted and stored in the variable *\$time*. The output of this variable will look like this: 09:27:03 pm.

```

$time_length = length($time);
$x = $font_length * $time_length;
$y = $font_height;

```

The size of the image is calculated, based on the length of the *\$time* string multiplied by the font dimensions.

```

$image = new GD::Image ($x, $y);
$black = $image->colorAllocate (0, 0, 0);

```

A new image is created with black as the background color.

```

if ($status eq "end") {
    $color = $image->colorAllocate (0, 0, 255);
    $image->transparent ($black);
} else {
    $color = $image->colorAllocate (255, 0, 0);
}

```

If the argument passed to this script is "end," the color of the text is set to blue. In addition, black is set as the transparent color. In other words, black will not appear in the image, and as a result the blue text will appear without any image border. If an argument was not passed, the text color is set to red.

```
$image->string (gdLargeFont, 0, 0, $time, $color);  
print $image->gif;  
}
```

Finally, the image is displayed to standard output.

Game of Concentration

Up to this point, we have discussed reasonably useful applications. So it is time now to look at some pure entertainment: the game of Concentration (also called Memory). The game consists of an arbitrary number of tiles, where each tile exactly matches one other tile. The value (or picture) "under" each tile is hidden from the user. Figure 11-1 shows what the initial screen looks like.

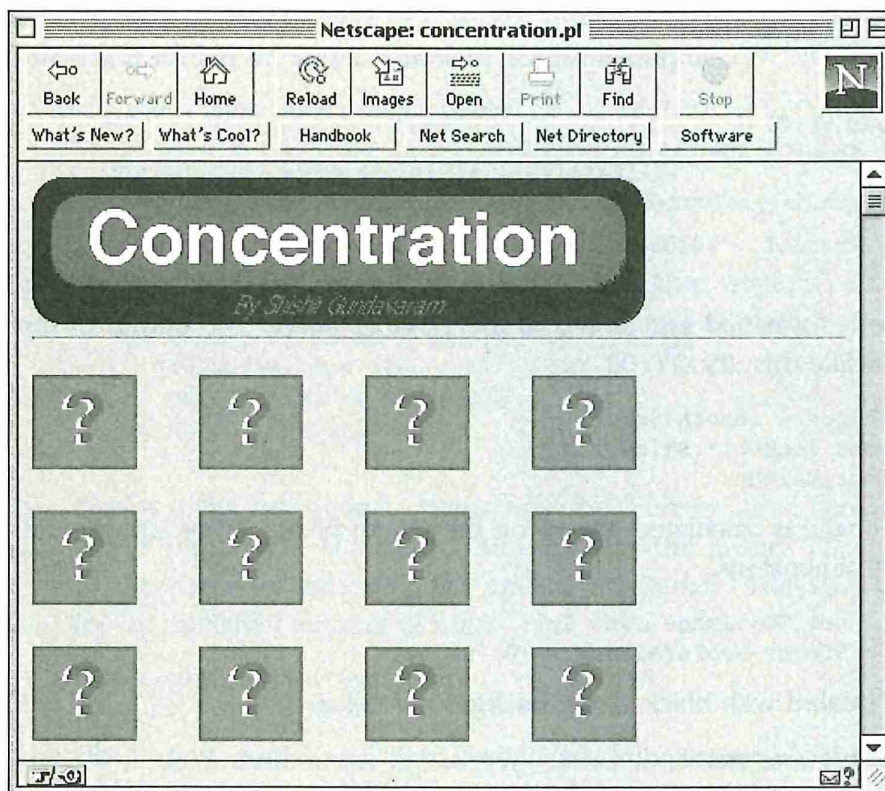


Figure 11-1. First game screen

When the user selects a tile, the value is displayed. The user can select two tiles at a time. If they match, the values behind the tiles remain displayed. The object