webFractal – Documentation

Introduction

A fractal is a geometric shape which is made up of smaller parts which are similar in shape to the original. These smaller parts in turn are made up of parts that resemble the original. This pattern continues recursively. Besides being interesting mathematically, fractals can also be visual beautiful shapes. The methods of generating fractals varies. Many fractals, including the Mandelbrot and Julia sets, are rendered by plotting an algorithm on the complex plane.

Overview

webFractal is a browser-based fractal exploring program written in Java and JavaScript. As a web-based application, it consists of two components: client and server. The server consists of several Java servlets served through Apache Tomcat. The client is run in any supported web browser. Features of webFractal include support for most major browsers, and a unique algorithm for improving the speed of fractal rendering.

Requirements

Server:

Apache Tomcat 5.5

Java Runtime Environment 1.5.0

Client:

Supported web browser (see below)

Development:

JUnit 4 is required for unit testing

Eclipse IDE is required to open project files

Installation

- Copy all the files in the /webapps/ directory of the zip file to the /webapps/ directory of Tomcat.
- Add the following line to the `<Host>` node of Tomcat's server.xml file (found in /conf/):

  ```
  <Context path="/fractal" reloadable="false" docBase="[webapps path]/fractal" workDir="[webapps path]/fractal/work" />
  ```

  (Replace [webapps path] with the path to your /webapps/ directory)
- Open the web.xml file found in /fractal/WEB-INF and set the ColormapDir parameter to the location of the colormap directory on your server.
- Restart the Tomcat server
- In your browser, enter the base URL of your Tomcat server and add /fractal/ to load the Fractal Explorer.

Supported Browsers

Platform	*Browser*	*Supported*	*Notes*
Windows	Internet Explorer 7	Works fully	Tested on Windows Vista
Windows	Internet Explorer 6	Works fully	Tested on Windows XP
Windows	FireFox 1.x / 2.0	Works fully	Tested on Windows XP
Windows	Opera 9.0	Small display bug, fully functional*	Tested on Windows XP, Windows Vista
Windows	Netscape 8.1.2	Works fully	Tested on Windows Vista
Mac	Firefox 1.0	Works fully	Tested on Mac OS 10.4
Mac	Camino	Works fully	Tested on Mac OS 10.4
Mac	Sarari 2.0	Not supported	Froze browser**
Mac	Internet Explorer	Not supported	Causes script error
Mac	Netscape 7.2	Works fully	Tested on Mac OS 10.4
Mac	Opera 9.0	Small display bug, fully functional*	Tested on Mac OS 10.4
Linux	Firefox 1.5	Works fully	
Linux	Opera 9.0	Small display bug, fully functional*	
Linux	Konqueror 3.4.0	Not supported	Froze Browser**
Linux	Mozilla 1.7.12	Works fully	

* In Opera 9 on all platforms, when the "Options" menu is opened or closed, the display does not fully update to reflect this until the grid behind it is moved. Other than this, the options box is fully functional in Opera 9.

** This problem has been fixed, but these browsers have not since been tested.

Using webFractal

Panning

There are two ways to pan the fractal. The first way is to drag anywhere on the fractal with your mouse. You can also use the arrow keys on your keyboard.

Zooming

webFractal lets you zoom in and out up to 16 levels. Each level has a magnification factor of four times the last level. There are three ways to zoom. The first is by double-clicking anywhere on the fractal. The fractal will be zoomed in and centered on the area that was double-clicked. You can also use the plus and minus buttons near the top of the screen to zoom in and out. The screen will remain centered at the same point after the zoom. The current zoom level is given by the number shown in the box. Finally, you can use the keyboard's plus and minus keys to zoom in and out. Again, the screen is zoomed in so that the center of the screen remains the same.

Options

webFractal allows you to change several options that affect the fractal and how it is displayed. The options menu can be accessed by clicking the "Options" tab which can be found on the bottom of the screen.

The options are:

Fractal Type - Choose between a Mandelbrot and Julia fractal.

Color Map - The list of colors used to map a fractal value to a color. The current fractal will be redrawn as soon as the color map is changed.

cx and **cy** - The X and Y coordinates (i.e., real and imaginary components) of the c value, a complex number used to calculate a Julia Fractal. Different cx and cy values can change the shape of a Julia fractal dramatically. These numbers are ignored when rendering Mandelbrot fractals.

Update Fractal - When you have changed the Fractal Type, cx, or cy, you must press the "Update Fractal" button to apply the changes.

Download Desktop Background - Downloads an image of the current fractal, resized to fit the desktop of the computer being used. The aspect ratio is preserved.

Code Overview

The server-side component of webFractal is written completely in Java. It consists of two servlets, **FractalServlet** and **ListColorsServlet**. The URL "/draw" (relative to the base URL of the application) is mapped to **FractalServlet**. **FractalServlet** is the servlet that creates the images of the fractal and sends them to the browser. It takes the parameters x1, y1, x2, y2, height, width, fractal, and colormap as a GET request. The image is sent back to the browser as a PNG.

The URL "/colors.js" is mapped to **ListColorsServlet**, which returns a JavaScript file that includes an array of the names of the color maps. This array is used to populate the drop-down box found in the fractal options form.

The **StartupContextListener** class contains a method which is run on server startup. This loads environment variables and colormaps into the ServletContext.

The main classes used in the application are:

- **ColorMap** - Loads a color map file and creates a list of possible values and the colors they correspond to.
- **ComplexNumber** - Stores a complex number and does simple complex number math, as well as generates the Julia or Mandelbrot value for that number.
- **FractalDrawer** and **PerformanceFractalDrawer** - Both implement the **FractalRender** interface, and do the work for **FractalServlet**. **PerformanceFractalDrawer** is a higher performance version of **FractalDrawer** using the "box and window" algorithm, which is described below.
- **Mandelbrot** and **Julia** - Both implement the **Fractal** interface. They hold the information for a Mandelbrot and Julia fractal respectively and are used to get the value of a given coordinate on the fractal's plane.

Other classes which are not essential to the application are:

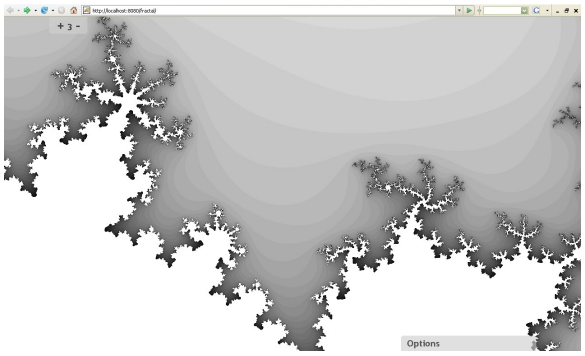
- **BoxedFractalDrawer** - Another implantation of the box and window algorithm that returns an image representing which squares were predicted by the algorithm and which ones had to be rendered.
- **ComplexNumberTest** - A JUnit4 unit test for the **ComplexNumber** class. It tests the mathmatical methods, but it does not test the fractal generating methods.

The client side HTML and JavaScript are sent to the browser through the Tomcat server as static pages. The fractal.js file holds all the JavaScript functions and classes.

- The **Mandelbrot** and **Julia** classes store information about the fractal, and provide the URL to an image sliced from that fractal given the coordinates and dimensions of that image slice. Although JavaScript does not support interfaces, pseudo-code for the **Fractal** interface is given

in the script's comments.

- The constructor of the **FractalGrid** class builds the grid that holds the images which make up the fractal. This class also moves the grid when it is told to, but the task of listening for user input is delegated to other classes.
- The **DragInput** class creates an invisible `<div>` element which spans the entire screen. When the mouse is held down, the `<div>` listens for mouse input, and calculates the number of pixels that the **FractalGrid** should be told to move.
- The **FractalOptions** class creates the "Options" menu that is shown at the bottom of the screen. It tells the **FractalGrid** to update when the options are changed.

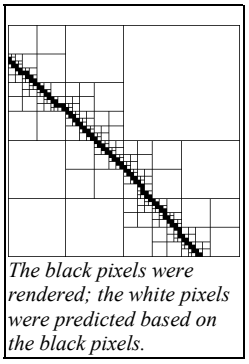


- The **ZoomControl** class creates the zoom buttons that are shown at the top of the screen. It tells the **FractalGrid** when the buttons are pressed, and listens to **FractalGrid** for changes in the current zoom level.
- The **KeyCatcher** class creates a function to listen for keyboard events. Because Internet Explorer does not have keyboard events on its window object, the `<body>`'s onmouse- events are used if possible.

Code Notes

Box-and-Window algorithm

In order to speed up the rendering of the fractals, I created an algorithm that is able to determine the color of the majority of the pixels without actually testing for them. The algorithm takes advantage of a property of Mandelbrot and Julia fractals that I noticed while working with the application. At any place on the fractal where you could draw a perimeter of a solid color, the area inside that perimeter will also be of that color. The only exception to this rule that I could find was if the perimeter encompasses the entire fractal.



The black pixels were rendered; the white pixels were predicted based on the black pixels.

Due to the nature of the application, FractalServlet is never asked to draw a complete fractal, it only draws one portion (slice) at a time. When a slice is requested, the Box method traces the perimeter of the slice to see if it is a solid color. If it finds that it is, it fills it with the solid color. If it is not, it sends it to the Window method. The Window method divides the square into four squares and sends each of these squares back to the Box method. By recursing until the box is quite small, the algorithm is able to predict the color of most pixels. This algorithm resulted in a dramatic speed increase of fractal rendering, and works flawlessly on both the Mandelbrot and Julia fractals.

To better demonstrate how the box-and-window algorithm works, I added a colormap option called "boxed". The resulting fractal is colored according to which pixels were rendered and which were predicted. The lighter blue shows where pixels did not need to be rendered. The black pixels were rendered individually. Dark blue pixels are the ones used to guess the contents of the box. They are rendered individually as well.

It is worth noting that the window shape used by the algorithm is itself a self-similar shape.

Since each window is made up of four more windows, each in turn made up of four more windows, etc. with the potential to go on forever, the window shape is self-similar.

Infinitely Scrollable Image

The infinitely scrollable image that makes up the fractal is actually a grid of images which are held together in an HTML `<table>`. As the table is moved, columns and rows are added or removed in order to fill the entire screen without keeping too many rows and columns in memory.

Browser Testing

The method used to test the application on each browser was:

- Load the Fractal Explorer in the browser and ensure that the page is rendered correctly
- Using the mouse, drag the fractal around
- Use the keyboard to pan the fractal
- Zoom in and out
- Try opening and closing the "options" menu
- Change the colormap and see if the fractal is updated appropriately
- Change the fractal type and update the fractal

It can be assumed that if the browser passes all these tests, the application will be fully functional. The "Download Desktop Background" button was not tested, as it was added after the testing was done.

Security

webFractal is a Java servlet application which does not access a database, so it is inherently fairly secure. However, some improvements could be made to make it safer for public use. One potential threat to the server would be a denial-of-service attack. Because the server uses a lot of resources to generate an image, and the height and width can be changed through the URL, it would be conceivable for someone to request hundreds of large images in a short time in order to slow down the server. Since the servlet container limits the access to system resources, this would not likely crash the server, but it may make the service unstable for other users. A possible solution to this would be to limit the number of requests per IP address as well as the size of the images requested.

Known Bugs and Limitations

- All caching is done on the client-side to save bandwidth when only one user has access to the application. If it were to be set up on a public server, it would be reasonable to cache the images on the server as well to reduce server load from re-rendering the same images.
- Not all browsers are supported.
- At high zoom levels, the image loses a lot of detail. This is probably due to rounding errors since the numbers being used at those zoom levels are very small.
- Due to the high number of requests to the server, some are occasionally dropped (The Tomcat log shows a "socket write error"). This causes an image to not load. The problem can be temporarily fixed by dragging the image outside of the screen and back in. Running the script on a server with more resources should reduce this problem.