A Web Browser Approach to the Design of Fractals and Multifractals

Experiments with Fractals

Dr. E. Garcia
admin@miislita.com

Published: May 28, 2010 Copyright © E. Garcia

Keywords: row primary, fractal patterns, web browsers, iterated layouts, fractal layouts

Abstract: This paper describes an approach for constructing fractal patterns on a Web browser by using CSS and XHTML. This simplifies the construction of fractals in a Web page as it is not necessary to use any additional software, the CSS Table Model, HTML tables, image files, complex mathematical algorithms, or the canvas tag introduced in HTML5.

1 Introduction

In a recent article (1), we described the design of CSS tableless layouts using a row primary approach. *A row primary layout* is one built with partitioned row divisions. This has several benefits. First, it forces a Web browser to render content from left to right and one-row-at-a-time. Second, the way content is rendered conforms to the way search engines and screen readers process documents. Third, a row primary layout allows one to associate an iteration level, therefore a row level, to the styling and nesting of markup elements.

In a follow up article (2), we explained why replacing table-based layouts with CSS tableless design not necessarily produces table-equivalent layouts. In two additional articles (3, 4) we described a tool that automates the creation of Web page layouts, including fractal layouts.

The traditional way of constructing fractals is with computer algorithms (5). Some of these use pixel-by-pixel drawing techniques wherein the output of a recursive function is evaluated against a predefined condition. Other strategies or combination of techniques use HTML tables, image files, VML, SVG, or the canvas tag introduced in HTML5 (6 - 8). In most cases, implementing these strategies and techniques involves a learning curve, requires of a large number of iterations, and consumes server resources.

The purpose of this article is to present an alternative to the design of fractals that does not require of any additional software or special resources, but of a Web browser that supports CSS and XHTML. Fractals are then generated with our row primary approach. Since this requires of a small number of iterations, the patterns can be coded by hand. Coding can also be automated by writing a JavaScript tool.

This paper is organized as follows. In section 2 we describe our experimental setup. The procedure is discussed in section 3. Results are presented in section 4. Findings and drawbacks are discussed in section 5. Finally, in section 6 we draw some conclusions.

2 Setup

An Acer computer with Windows 7 and at a screen resolution of 1366 x 768 pixels, landscape orientation, was used. All iterated patterns were tested with versions of Internet Explorer 8 and 7 (IE 8, IE 7), Firefox 3 (FF 3), Opera 10 (OP 10), Safari 4 (SF 4), and Google Chrome 4 (GC 4). The Fractal CSS Design Studio 2 (FCDS 2) tool described in Reference 3 was modified in order to iterate the markup instructions to be discussed and renamed FCDS 3.

3 Procedure

To create a pattern, a canvas division of length **L** and area $A = L^2$ is defined. Unlike the HTML5 canvas tag which addresses pixels, this division holds markup. Next, we declare row divisions; i.e.

```
<div class=”canvas”><div class=”row”>…</div>…</div>
```

Each row will accommodate cell divisions, to be left-floated or positioned according to design requirements. Cell divisions are styled by defining their sizes and background colors. CSS borders are optional, but if used, one should understand that these consume additional pixels, affect the area occupied by a cell, and impact the shape of a pattern. Upon recursion, bordered cells can also end touching neighboring borders. If these are styled with the background color of a cell, the layout will look disconnected.

To give the illusion of using bordered cells, wrapper divisions (i.e., <div class=”wp”></div) are used within cells and their borders styled. It is not necessary to declare all four borders of a wrapper, but the top (or bottom) and left (or right) borders. To avoid the cumulative effect of CSS borders, the iteration level used is also assigned to a style instruction. An example follows.

3.1 Associating an iteration level to style instructions

Let

```
<div><div class=”wp”></div></div>
```

be a markup fragment. Its first iterate (**i = 1**) is

```
<div><div class=”wp”><div><div class=”wp”></div></div></div></div>
```

To style the top and left borders of the internal wrapper without styling the external wrapper, the following CSS rule is used.

```
div.wp div.wp{border-top:1px solid #000;border-left:1px solid #000;}
```

Thus for **i** iterations, the div.wp is repeated **i + 1** times in the CSS rule. This is an example of associating an iteration level to the style rule. With the FCDS 3 tool, this is accomplished with a FOR-TO loop. If bordered cells are not needed, the wrappers and rule can be removed altogether.

3.2 A word on rows and cells

The numbers of rows and cells per rows to be declared depend on the type of fractal to be generated. In general, the following markup is used:

```
<div class=”row”>
<div class=”c11”><div class=”wp”></div></div>
<div class=”c12”><div class=”wp”></div></div>
⋮                    ⋮                    ⋮
<div class=”bk”></div>
</div>
```

For subsequent rows, the markup is defined in a similar way. A breaker division (i.e., <div class=”bk”></div>) is also declared to prevent repositioning (refloating) of cells upon any resizing of the browser window. This division is styled with the div.bk{clear:both;height:0px;} style rule. Any iteration of the markup occurs within the wrapper divisions.

4 Results

In this section, we describe the construction of the fractal patterns shown in Figures 1 through 13. These were generated with IE 8. Source code instructions for these figures are available online from Mi Islita.com (http://www.miislita.com/fractals/fractal-source-codes.html) or by email by contacting the author. Due to limitation of space, in this article we only discuss the style and markup instructions for the Sierpinski Gasket.

4.1 Sierpinski Gasket

This is one of the best known fractals, sometimes also called the Sierpinski Triangle. To construct this pattern, two row divisions are declared within the canvas division. The area of each row is ½ of the whole canvas.

The first row spans across three cells, c11, c12, and c13. The area of c11 and c13 is ¼ of the whole first row while the area of c12 is ½ of this row. The second row spans across two cells, c21 and c22. The area of each cell is ½ of the whole second row.

The initial layout at i = 0 and sometimes also called *generator*, *motif*, or *blueprint* is shown in Figure 1. Its CSS and markup instructions are listed in Table 1. Comment instructions are in red. Note that the generator consists of a three- and a two-column layout. These types of layouts are commonly used in Web page design (1 - 4).

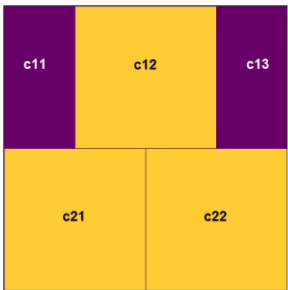


Figure 1. Sierpinski Gasket generator, resembling a Web page layout with three and two columns.

Table 1. Style and markup instructions for the Sierpinski Gasket generator.

CSS	XHTML
html,body{font-size:100%;height:100%;width:100%;	<div class="r">
margin:0px;padding:0px;border:0px;}	<div class="c11"><div class="wp"></div></div>
div{height:100%;margin:0px;padding:0px;border:0px;	<div class="c12"><div class="wp">
overflow:hidden;}	**<!-- Iterated layout goes here -->**
div.canvas{margin:0 auto;width:25em;height:25em;	</div></div>
border:1px solid #000;}	<div class="c13"><div class="wp"></div></div>
div.r{height:50%;background:#606;}	<div class="bk"></div>
div.bk{clear:both;height:0px;}	</div>
/*Repeat ‘div.wp’ i + 1 times. See Procedure section.*/	<div class="r">
div.wp{border-top:1px solid #606;border-left:1px solid #606;}	<div class="c21"><div class="wp">
div.c11,div.c13{float:left;width:25%;}	**<!-- Iterated layout goes here -->**
div.c12,div.c21,div.c22{float:left;background:#fc3;width:50%;}	</div></div>
/*Add next line for IE 7.*/	<div class="c22"><div class="wp">
div.c13,div.c22{float:none;width:auto;}	**<!-- Iterated layout goes here -->**
	</div></div>
	<div class="bk"></div>
	</div>

To construct the Sierpinski Gasket, this markup is iterated by pasting it back into each cell except cells c11 and c13. Figure 2 shows the emergence of the Sierpinski Gasket after six iterations.

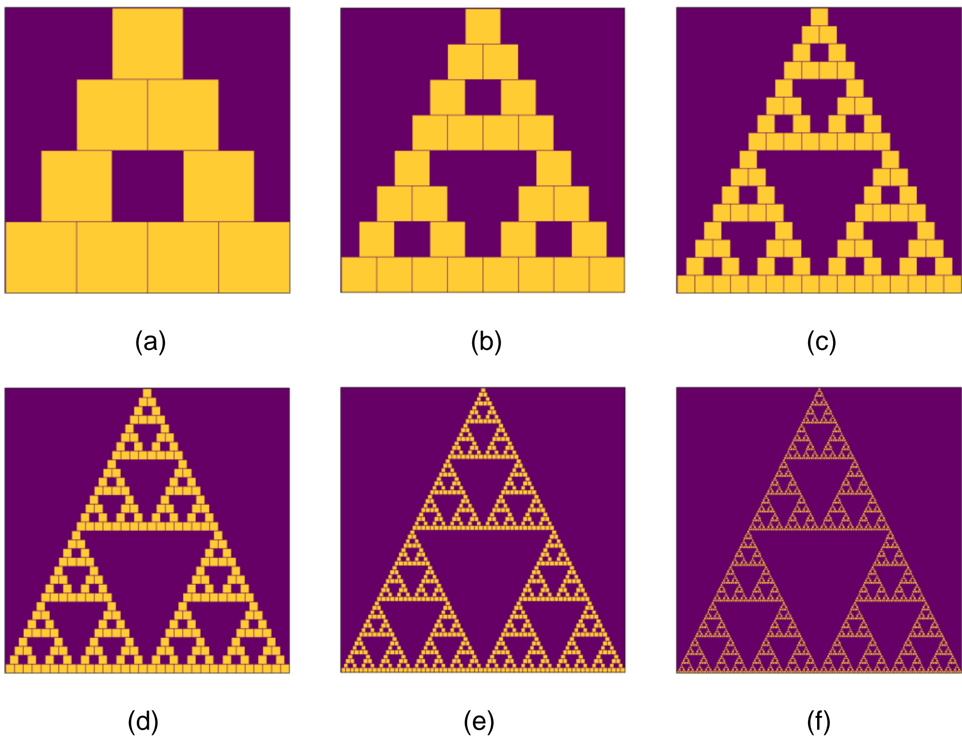


Figure 2. Sierpinski Gasket, obtained after iterating six times the row primary layout described in Figure1 and Table 1.

4.2 Left-Skewed Sierpinski Gasket

This is a modification of the previous pattern. Two rows are defined on the canvas, each spanning two cells: c11, c12, c21, and c22. Essentially each row is partitioned into a two-column layout.

The area of each cell is ½ the area of a row. Since the canvas occupies the area of four cells, the area of each cell is, therefore, ¼ the area of the whole canvas. This markup is iterated on each cell, except c12. Figure 3 shows the generator (a) and attractor (b) obtained after six iterations.

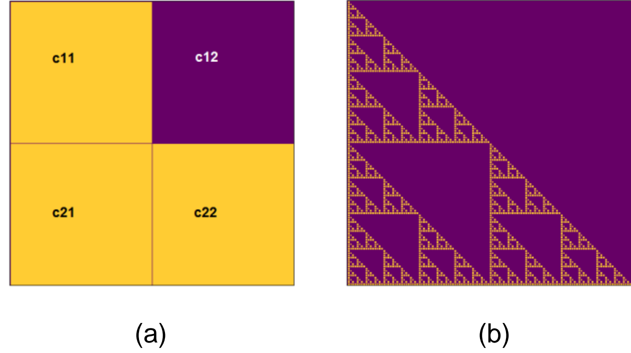


Figure 3. Left-Skewed Sierpinski Gasket, showing its generator (a) and attractor (b) after six iterations.

4.3 Sierpinski Carpet

According to Peitgen, et. al (9), unlike the Sierpinski Gasket the Sierpinski Carpet is a universal object that houses any one-dimensional object in the plane in the topological sense. This includes lines, squares, and other fractals like modified versions of the Sierpinski Gasket, Koch Curve, and Cantor Set.

The Sierpinski Carpet is constructed by defining three row divisions on the canvas. The area of each row is 1/3 of the whole canvas. Each row spans across three cells. The area of each cell in a row is 1/3 of the whole row or 1/9 of the whole canvas.

The markup for rendering the carpet is iterated on each cell, except cell c22. Figure 4 shows its generator (a) and attractor after three iterations; with (b) and without (c) declaring borders. Definitively, borders impact the end result.

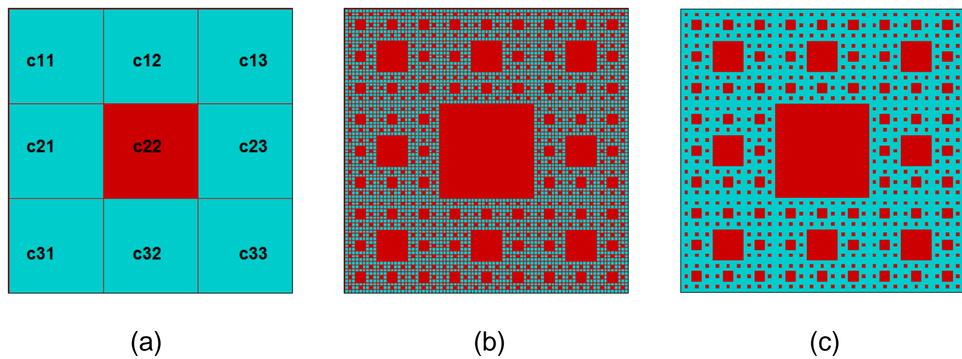


Figure 4. Sierpinski Carpet, showing its generator (a) and attractor after three iterations, with (b) and without (c) declaring borders.

4.4 Vicsek Fractal

Also known as Vicsek Snowflake or Box Fractal, this pattern arises from a construction similar to that of the Sierpinski Carpet, except that it can be constructed from two equivalent generators. These are shown in Figure 5 (a) through (d) and after four iterations.

In (a), the generator is iterated on all cells except c12, c21, c23, and c32. In (c), the generator is iterated on all cells except c11, c13, c31, and c33. The two attractors and fractal dimensions are equivalent, except that one attractor (d) is rotated by 45 degrees.

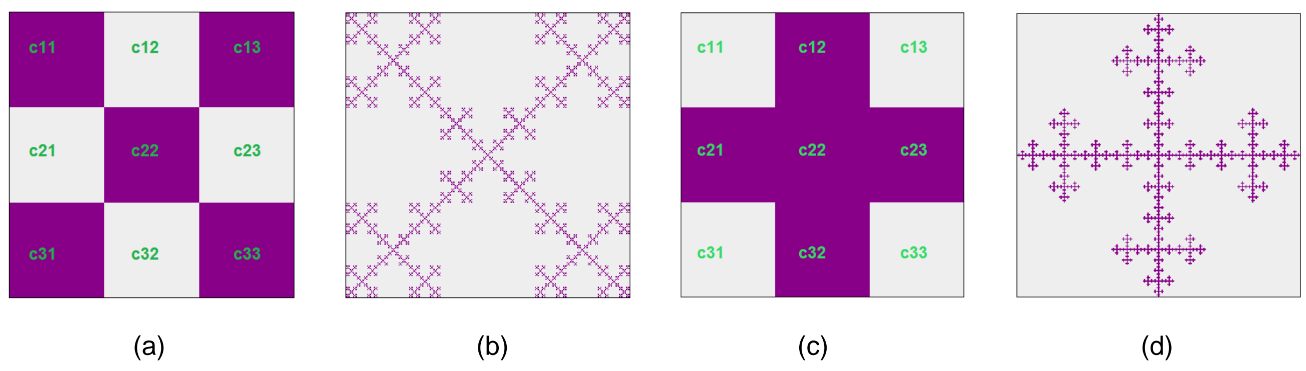


Figure 5. Vicsek Fractal, showing its generators (a, c) and attractors (b, d) after four iterations.

4.5 New Carpets

Rani and Kumar (10) have documented a new class of carpets within the Sierpinski Carpet. In the next subsections these are reproduced with our row primary approach.

4.6 Dhan Carpet

The Dhan Carpet is constructed as described by Rani and Kumar (10). Four row divisions are defined in the canvas division. The area of each row is 1/4 of the whole canvas. Each row spans across four cells. The area of each cell in a row is 1/4 of the whole row or 1/16 of the whole canvas. The HTML markup for rendering the carpet is iterated on each cell, except cells c11, c14, c41, and c44. Figure 6 shows its generator (a) and expected attractor after just two iterations.

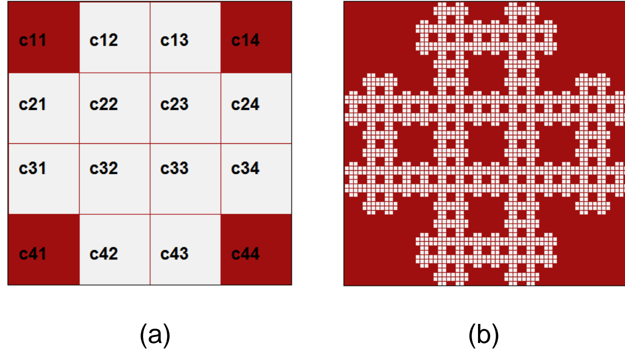


Figure 6. Dhan Carpet, showing its generator (a) and attractor (b) after two iterations.

4.7 Black and Red Carpet

Constructing this carpet consists in defining three row divisions in the canvas division. The area of each row is 1/3 of the whole canvas. Each row spans across three cells. The area of each cell in a row is 1/3 of the whole row or 1/9 of the whole canvas. The HTML markup for rendering the carpet is iterated on each cell, except cell c31. Figure 7 shows its generator (a) and expected attractor after three iterations.

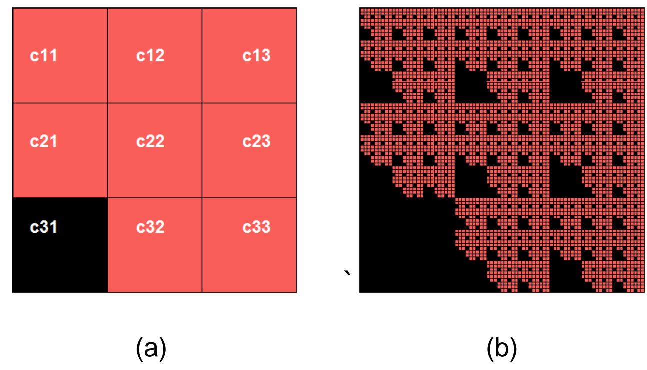


Figure 7. Black and Red Carpet, showing its generator (a) and attractor (b) after three iterations.

5. Discussion

5.1 Fractal Dimension Calculations

In a strict sense, construction of a fractal requires defining an initiator, a generator, and a scaling factor (11, 12). For fractals of the Sierpinski family, the initiator is a square of unit length (**L = 1**) and unit area ($A = L^2 = 1$). This is partitioned into b^2 smaller squares of length $L(1/b) = L*1/b = 1/b$ and area $A(1/b) = A*(1/b)^2 = (1/b)^2$, where **b** = 2, 3, 4... From here that **b**, therefore **1/b**, is a scale factor.

The generator consists of **n** smaller squares to be removed and $b^2 - n$ smaller squares to be retained. The generator is then iterated within the retained squares and the fractal dimension **D** is computed as

$$D = \log(b^2 - n)/\log(b) \qquad \text{(Eq 1)}$$

where the probability that $b^2 - n$ parts remain in the fractal is $p = (b^2 - n)/ b^2 = 1 - n/ b^2$.

For instance for the Sierpinski Carpet, b = 3, n = 1, and p = 8/9 = 0.88…; so, **D** = log(8)/log(3) ≈1.89. Table 2 lists **D** values computed for several fractals.

Table 2. Patterns and their fractal dimensions.						
Fractal	Sierpinski Gasket	Left-Skewed Sierpinski Gasket	Sierpinski Carpet	Vicsek Fractal	Dhan Carpet	Black & Red Carpet
D ≈	1.58	1.58	1.89	1.46	1.79	1.89

5.2 Mann Iteration Method

Rani and Kumar (10) have documented a new class of carpets within the Sierpinski Carpet using different scaling factors on a given generator. These scaling factors are computed with Mann Iteration Method (10, 13, 14) which works as follows. For a function **f** on a linear space **X** and an initial value x_0 (in **X**), define x_1 as a new value. In general, at a given iteration level **i**

$$x_{i+1} = \lambda * f(x_i) + (1 - \lambda) * x_i \qquad \text{(Eq. 2)}$$

where $0 < \lambda \leq 1$ and λ and $1 - \lambda$ are scaling factors.

Note that uniformly scaling a **k**-dimensional shape reduces to computing λ^k and $(1 - \lambda)^k$, where for a line **k = 1**, for a square **k = 2**, and for a sphere or cube **k = 3**. Thus for a square of area **A**, Mann's scaling factors define two scaled areas:

$$A(\lambda) = A * \lambda^2 \qquad \text{(Eq 3)}$$

$$A(1 - \lambda) = A * (1 - \lambda)^2 \qquad \text{(Eq 4)}$$

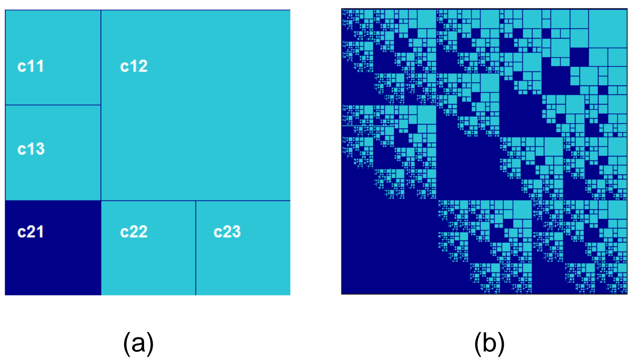
These two areas define the cells of the generators used by Rani and Kumar. In the rest of this section we reproduce their carpets with our row primary approach. The only drawback is the number of iterations that a Web browser can handle without crashing. Fortunately, our approach requires of a few numbers of iterations.

5.3 Kangri Carpet

This pattern, from Rani and Kumar (10), is shown in Figure 8. For its construction, we define two row divisions, r1 and r2, within the canvas. Row r1 contains rows r11 and r12, where r11 contains cells c11 and c13 and r12 includes c12. Row r2 contains cells c21, c22, and c23.

Initially, we tried to design this carpet with a simpler layout by placing c11, c12, and c13 all in one row, with smaller cells floated to the left and the larger cell floated to the right, but inconsistent results were obtained across browsers. This simpler layout, however, produced acceptable results with their Krishna and Vinod Carpets.

In the Kangri Carpet, the area of the smaller cells is $1/9 = (1/3)^2$ of the whole canvas while the larger cell occupies the area of 4 smaller cells or $4/9 = (2/3)^2$ of the whole canvas. These areas conform to Equations 3 and 4, but differ from those reported by Rani and Kumar. The markup of this layout is then iterated on each cell, except cell c21. Figure 8 shows its generator (a) and expected attractor after four iterations. It can be confirmed from Equation 1 that the fractal dimension of the carpet is the same as that of the Sierpinski Carpet.

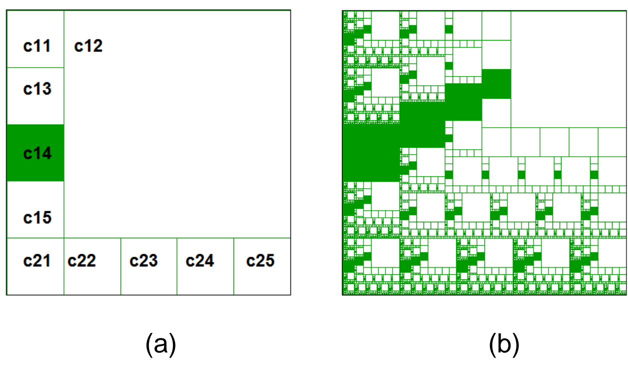


(a) (b)

Figure 8. Kangri Carpet, showing its generator (a) and attractor (b) after four iterations.

5.4 Krishna Carpet

To design this carpet (10), we define two rows in the canvas. Each row spans across five cells. The area of the first row is 4/5 of the whole canvas and that of the second row is 1/5 of the whole canvas. The largest cell (c12) is CSS-floated to the right of c11 and occupies the area of 16 smaller cells. Its area is $16/25 = (4/5)^2$ of the whole canvas and that of each of the smaller cells is $1/25 = (1/5)^2$ of the canvas. These areas differ from the ones reported by Rani and Kumar. The markup is iterated on each cell, except cell c14. Figure 9 shows its generator (a) and attractor after three iterations.



(a) (b)

Figure 9. Krishna Carpet, showing its generator (a) and attractor (b) after three iterations.

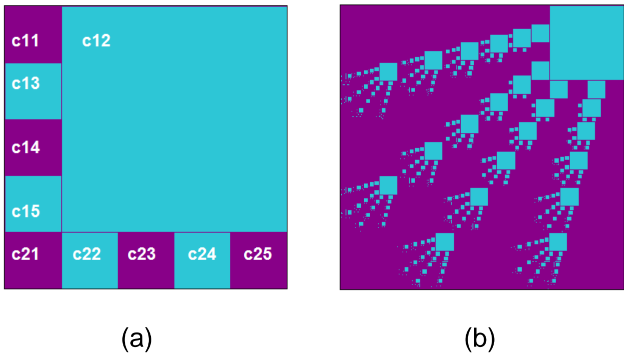
5.5 Vinod Carpets 1 and 2

To create their Vinod Carpet, Rani and Kumar said (10):

> "To design a Vinod carpet, divide the whole square into ten squares. Out of all the squares, the area of the largest one is 4/5 and that of the remaining squares is 1/5 of the whole square. Drop the alternate small squares starting from the left top square as shown in the blueprint. By repeating this process on each square again and again, after enough iterations, we get Vinod carpet."

We disagree with this procedure for three reasons:

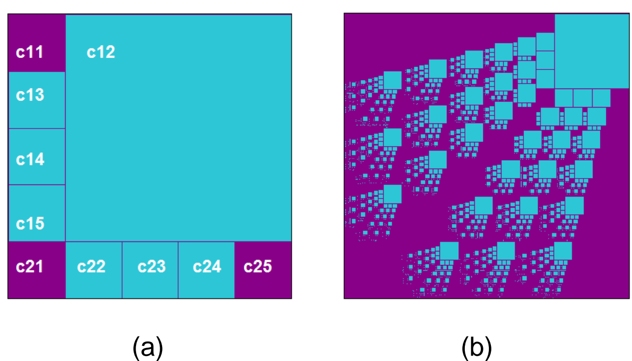
- First, the largest cell in their reported blueprint occupies the area of 16 smaller cells, being $16/25 = (4/5)^2$ of the whole canvas. So the area of each of the smaller cells is $1/25 = (1/5)^2$ of the whole canvas.
- Second as depicted in Figure 10, their generator does not produce the expected carpet.
- Third, since the generator was not the right one, the fractal dimension obtained from the generator does not correspond to that of the carpet.



(a) (b)

Figure 10. Vinod Carpet 1, showing its generator (a) and attractor (b) after five iterations.

To reproduce their Vinod Carpet, we modified the previous generator by iterating the markup five times on each cells, except c11, c21, and c25. The new generator and carpet (herein termed Vinod Carpet 2) are shown in Figure 11.



(a) (b)

Figure 11. Vinod 2 Carpet, showing its generator (a) and attractor (b) after five iterations.

These results now agree with those of Rani and Kumar. The fractal dimension of Vinod Carpet 1 is $D = \log(22)/\log(5) \approx 1.92$ while that of Vinod Carpet 2 is $D = \log(20)/\log(5) \approx 1.86$.

5.7 Multifractal Sierpinski Carpets

A multifractal is a pattern whose complexity cannot be characterized with a single fractal dimension. One way of constructing these patterns on a two-dimensional grid consists in color-coding cells. The following example was inspired from Perfect et al. (12), but with some modifications. As in their work, the generator is a square with 9 cells of equal sizes, without borders. These are colored in different shades of black and white according to a distribution function. Figure 12 shows the generator (a) and two different attractors (b, c) after four iterations. In (b) the generator was iterated on all cells except c33 while in (c) it was iterated on all cells. The new attractors house several fractal patterns.

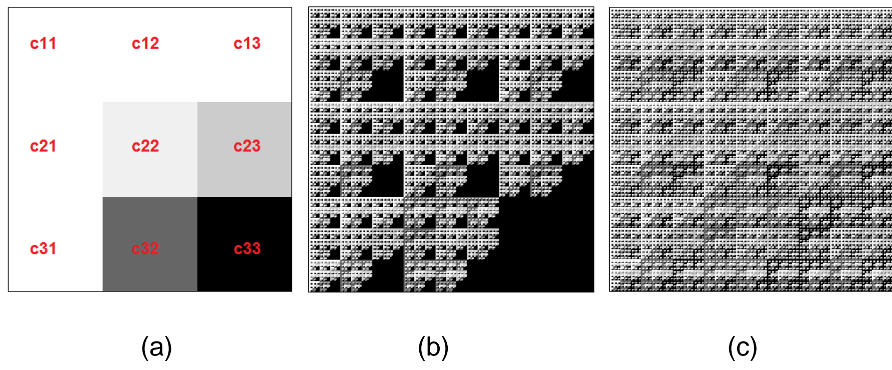


Figure 12. Multifractal carpets, showing the generator (a) and attractors (b, c) after four iterations.

The following example was motivated from Pallat et al. (15). As in Reference 12, they used a 3x3 square, but cells were color-coded representing statistical properties associated to a link graph network. We modified their generator by iterating it on all cells except c12 and c21. Figure 13 shows the generator (a) and new attractor (b) after four iterations. As in Figure 12, several fractal patterns are found in this carpet.

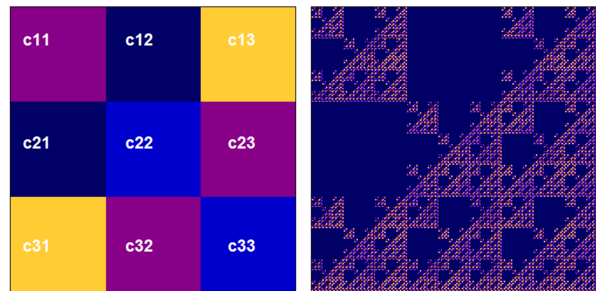


Figure 13. Multifractal carpet, showing its generator (a) and attractor (b) after four iterations.

5.8 Drawbacks: sub-pixel rounding problems across browsers

Resig (16) has shown that browsers introduce sub-pixel resolution errors because of the different approach used when, for instance, rounding widths calculated from percentages. He found that OP 10 and SF4 tend to round sub-pixels down, IE 7 and 6 tend to round up, and Firefox 3 tends to round alternately up and down.

To the best of our understanding, the effect of browsers sub-pixel rounding errors on iterated patterns is not well documented; and we haven't found a benchmark on this. The topic is important as small errors can lead to unexpected results. Conversely, iterated layouts obtained from Web browsers should shed some light into the rounding rules used by these. To test this idea, in Table 3 we have generated several Sierpinski Carpets with the above browsers. Their layout engines are also listed.

Table 3. Effect of browser sub-pixel rounding errors on the Sierpinski Carpet.

Browser	i = 1	i = 2	i = 3
Opera 10 (OP 10) **Engine: Presto**			
Safari 4 (SF 4) **Engine: WebKit**			
Google Chrome 4 (GC) **Engine: WebKit**			
Internet Explorer 7, 6 **Engine: Trident**			
Firefox 3 **Engine: Gecko**			
Internet Explorer 8 **Engine: Trident 4.0**			

Note that OP 10 and SF 4 leave gaps, confirming that these browsers round sub-pixels down. The carpet is ruined after few more iterations.

The table also reveals that IE 7, like IE 6, rounds up, causing elements to overflow containers, and fragmenting the Sierpinski Carpet. These browsers were not tested directly, but emulated. For example, IE 7 was emulated by placing the following tag in the head section of the generated markup:

```
<meta http-equiv="X-UA-Compatible" content="IE=EmulateIE7" />
```

In the case of FF 3, this browser was capable of correctly drawing the Sierpinski Carpet, confirming that it rounds sub-pixels alternately up and down in order to make elements fit completely.

In Table 3 we listed two additional browsers, not included in Reference 15: GC 4 and IE 8. The results suggest that GC 4 could be using rounding rules more similar (i.e., not necessarily identical) to those of SF 4 than to IE 7 while IE 8 could be using rounding rules more similar to those of FF 3 than to IE 7 or GC 4.

This educated guess can be reasoned in terms of the type of layout engines used by these browsers. GC 4 and SF 4 are built around the WebKit layout engine, so it is not entirely surprising that these browsers produce similar results. By contrast, IE 7 uses Trident while IE 8 uses Trident 4.0; thus, it is not equally surprising to see these browsers producing dissimilar results. (17 - 22).

We have retested these browsers with other patterns like the Vicsek Fractal 2. Our results are summarized in Table 4. The best results were obtained with IE 8. As before, the rounding rules used by the rest of the browsers ruined the final output.

We tried to find CSS workarounds to fix these results with limited success. For instance, for IE 7 we found that assigning

```
{float:none;width:auto;}
```

to the last cell of a row tends to prevents these from overflowing a row, while assigning a height slightly above 100% to the cells,

```
{height:101%;overflow:hidden;}
```

hides the extra horizontal gaps in IE 7 and FF 3, but causes not so obvious distortions. This second fix is not recommended for the other patterns like the Dhan Carpet. Actually, we were not able to find a simple fix that would work well 100% of the time across patterns and browsers. Accordingly, at the time of writing we can only recommend Web users to view our patterns with the latest versions of Internet Explorer or Firefox, with IE being the preferred option. Fortunately, most users prefer these over all other browsers (23).

Table 4. Vicsek Fractal 2 patterns after four iterations and obtained before and after CSS workarounds.		
Browser	**Before**	**After**
OP 10		
SF 4		
GC 4		
IE 7, 6		
FF 3		

6 Conclusion

We have demonstrated a simple technique for constructing fractal and multifractal patterns with a Web browser. The iterated layouts shed some light into the sub-pixel rounding rules used by these.

Rani and Kumar's carpets were also reproduced and their results corrected. However compared with their carpets, ours were of low resolution since a few numbers of iterations were used to prevent the Web browsers from crashing.

Fractal carpets have widespread applications in many areas: from graphic design, to physical carpets design; and from the study of complex electrodes, to the study of diffusion-limited aggregation phenomena (24). Even electronic devices like cell phones have fractal antennas shaped as Sierpinski carpets (25, 26).

Although not demonstrated in this work, our row primary approach can be used to speed up other Web-based tools, like the Bifurcation Diagram program described in Reference 6 and the Mandelbrot Viewer developed by Aliverti (27). In each case, img and td table elements can be replaced with styled division elements and the iterated output colored one row at a time.

References

1. Garcia, E. (2009). *Fractal CSS Design*.
   http://www.miislita.com/fractals/fractal-css-design.pdf
2. Garcia, E. (2010). *Fractal CSS Design as a Row Primary Technique*.
   http://www.miislita.com/fractals/fractal-css-design-row-primary-technique.pdf
3. Garcia, E. (2010), Fractal *Movies, CSS-only Backgrounds, and Two-Column Layouts.*
   http://www.miislita.com/fractals/fractal-movies.pdf
4. Garcia, E. (2010). *Fractal Art and Three-Column Iterated Layouts*.
   http://www.miislita.com/fractals/fractal-art-iterated-layouts.pdf
5. Barnsley, M. *Fractals Everywhere*.
   Academic Press, San Diego (1988).
6. Nakhimovsky, A., Myers, T. *JavaScript Objects.*
   Chapter 9, pages 306 – 308; Wrox Press, Chicago (1998).
7. Bromberg, P. A. *Javascript Animated Martin Fractals with IE and VML.*
   http://www.eggheadcafe.com/articles/20010908.asp
8. Sucan, M. *SVG or Canvas? Choosing between the two.*
   http://dev.opera.com/articles/view/svg-or-canvas-choosing-between-the-two/
9. Peitgen, H., Jürgens, H., and Saupe, D. *Fractals for the Classroom, Part One: Introduction to Fractals and Chaos*. New York: Springer-Verlag (1992).
10. Rani, M. and Kumar, V. *New Fractal Carpets.*
    The Arabian Journal for Science and Engineering, Volume 29, Number 2C (2004).
    http://ajse.kfupm.edu.sa/articles/292C_09P.pdf
11. Mandelbrot, B. *The Fractal Geometry of Nature*.
    Freeman, New York, (1983).
12. Perfect, E., Gentry, R.W., Sukop, M.C., and Lawson, J.E. *Multifractal Sierpinski carpets: Theory and application to upscaling effective saturated hydraulic conductivity.* Geoderma 134, 240–252 (2006).
    http://web.utk.edu/~eperfect/Papers/Perfect%20et%20al.%20(2006).pdf
13. Mann, W. R. *Mean Value Methods in Iteration.*
    Proc. Amer. Math. Soc., 4 pp. 506–510 (1953).
14. Abbas, M., Khan, S. H., and Rhoades, B. E. *Simpler is also better in approximating fixed points.*
    Applied Mathematics and Computation 205, 428–431 2008).

http://qspace.qu.edu.qa/bitstream/handle/10576/10521/Simpler%20is20also20better%20in%20approximating%20fixed%20points%20.pdf?sequence=1

15. Pallat, G., Lovasz, L., and Vicsek, T. *Multifractal Network Generator.* http://arxiv4.library.cornell.edu/PS_cache/arxiv/pdf/1004/1004.5225v1.pdf
16. Resig, J. *Sub-pixel Problems in CSS.* http://ejohn.org/blog/sub-pixel-problems-in-css/
17. WebKit. http://en.wikipedia.org/wiki/WebKit
18. Opera Browser http://en.wikipedia.org/wiki/Opera_browser
19. Internet Explorer. http://en.wikipedia.org/wiki/Internet_Explorer
20. Mozilla Firefox. http://en.wikipedia.org/wiki/Mozilla_Firefox
21. Google Chrome http://en.wikipedia.org/wiki/Google_Chrome
22. Comparison of Layout Engines. http://en.wikipedia.org/wiki/Comparison_of_layout_engines
23. Wikipedia.org, *Usage Share of Web Browsers.* http://en.wikipedia.org/wiki/Usage_share_of_web_browsers#Summary_table
24. Jiang, M., Ko, J. Y., Liu, W. K., and Chiang, J. C. *A Computer Simulation Study of Diffusion-Limited Aggregation on Sierpinski Lacunar Lattice.* Chinese Journal of Physics Vol. 32, No. 5-1 (1994). http://psroc.phys.ntu.edu.tw/cjp/download.php?d=1&pid=800
25. Felber, P. *Fractal Antennas. A literature study as a project for ECE 576* (2000). http://www.ece.iit.edu/~pfelber/fractalantennas.pdf
26. Fractal Antenna System, Inc. http://www.fractenna.com/
27. Aliverti, B. (2001). *A JavaScript Mandelbrot Viewer.* http://theacf.com/apps/mandelbrent.html