Image Based Rendering of Iterated Function Systems

J.J. van Wijk [a,*], D. Saupe [b]

[a] *Technische Universiteit Eindhoven, Dept. of Mathematics and Computer Science, P.O. Box 513, 5600 MB Eindhoven, The Netherlands*

[b] *Universität Konstanz, Dept. of Computer Science, Box D697, D-78457 Konstanz, Germany*

Abstract

A fast method to generate fractal imagery is presented. Iterated Function Systems (IFS) are based on repeatedly copying transformed images. We show that this can be directly translated into standard graphics operations: Each image is generated by texture mapping and blending copies of the previous image. Animations of dynamic IFS codes are generated at fifty frames per second on a notebook PC, using commodity graphics hardware. Various extensions and variations are presented.

Key words: Fractal imagery, iterated function systems, hardware acceleration

* Corresponding author.
Email addresses: vanwijk@win.tue.nl (J.J. van Wijk), dietmar.saupe@uni-konstanz.de (D. Saupe).

1 Introduction

Fractal imagery is one of the most fascinating topics in computer graphics. With only a few lines of code, highly intricate images with details on all scales can be generated. We consider one class of these images: The well-known deterministic fractals produced by two-dimensional Iterated Function Systems (IFS's). One example is the Black Spleenwort Fern image, devised by Michael Barnsley, which is defined by only a few numbers and a compact algorithm. In the next section we give a very short overview of IFS's and algorithms to generate imagery; a very readable introduction can be found in [1], in [2] an in-depth treatment can be found.

In section 3 we present a new technique to generate images of IFS's using graphics hardware acceleration. We show that the definition of the attractor can be translated almost directly into graphics operations. Several extensions and variations, i.e., the use of colour, dynamic IFS's, and the visualization of the space surrounding of the attractor, are presented. The performance of the method is discussed in section 4 and compared to results with a standard algorithm. Thanks to the acceleration by graphics hardware a frame rate of 50 frames per second can be achieved for animations of dynamic IFS's. Finally, conclusions are drawn.

2 Background

An Iterated Function System (IFS) code with probabilities consists of a set of transformations $\{w_1, w_2, \cdots, w_N\}$ and associated positive probabilities $\{p_1, p_2, \cdots, p_N\}$ with $\sum_{i=1}^{N} p_i = 1$. We consider affine and contractive two-dimensional trans-

formations $w_i : \mathbb{R}^2 \to \mathbb{R}^2$, defined by

$$w_i \begin{pmatrix} x \\ \\ y \end{pmatrix} = \begin{pmatrix} a_i & b_i \\ \\ c_i & d_i \end{pmatrix} \begin{pmatrix} x \\ \\ y \end{pmatrix} + \begin{pmatrix} e_i \\ \\ f_i \end{pmatrix}.$$

Given an IFS code, there exists a unique associated geometrical object, denoted by $\mathcal{A}$, a compact subset of $\mathbb{R}^2$, called the attractor of the IFS. The attractor is defined as the set of points for which

$$\mathcal{A} = \bigcup_{i=1}^{N} w_i(\mathcal{A}).$$

Informally, the attractor can be understood as a result of an iteration as follows. Suppose that initially a sufficiently large domain (e.g., a square Q with $w_i(Q) \subset Q$ for $i = 1, \ldots, N$) is covered with an (infinitely fine) powder, such that the amount of mass that covers each point is the same. In one iteration each transformation w_i moves a portion p_i of mass around as specified by the corresponding affine mapping. Now, points that remain covered with mass in the long run belong to the attractor. The set $\mathcal{A}$ is controlled by the affine maps w_i and does not depend on the choice of the probabilities. Furthermore, an IFS code has an associated measure denoted by μ, which is governed by the probabilities p_i. Intuitively, some areas will be covered with more mass than other areas. The measure of a subset B of $\mathcal{A}$ can be defined (informally) as the weight of the mass which lies upon B.

Various algorithms have been developed to produce images of $\mathcal{A}$. The *deterministic algorithm* follows almost directly from the definition of $\mathcal{A}$. Choose a

compact set $A_0 \subset \mathbb{R}^2$. Next, generate a sequence $A_k, k = 1, 2, \cdots$ according to

$$A_{k+1} = \bigcup_{i=1}^{N} w_i(A_k).$$

This sequence will converge to $\mathcal{A}$. The algorithm can be implemented in a straightforward way to generate a sequence of binary images F_k that approach an image F of $\mathcal{A}$. For simplicity, we assume that the domain of the image is the unit square ($\square$), we adopt the convention that a value of 0 denotes black (background) and a value of 1 denotes white ($\mathcal{A}$), and we assume that the image is discretized in square pixels. Obviously, images are bounded, outside a fixed rectangle the image is assumed to be 0. In each iteration of Barnsley's implementation of the deterministic algorithm the new image is set to zero first, next all pixels are scanned. For pixels (u, v) that are white in the old image, the coordinates $w_i(u, v)$ are calculated and the corresponding pixels are set to white.

The most well-known algorithm is the *random iteration algorithm*. It relies on the generation of a sequence of points $x_n, n = 0, 1, \cdots$. Starting from an arbitrary point x_0, each next point is picked from the set $\{w_1(x_{n-1}), w_2(x_{n-1}), \cdots, w_N(x_{n-1})\}$, where the propability of the event $x_n = w_i(x_{n-1})$ is p_i. The first, say ten, points are discarded, plotting the remaining points gives an image that approaches $\mathcal{A}$ again. A variation of this algorithm is to count how many points fall inside each pixel. In the limit this count, suitably weighted, gives an image of the measure μ of each pixel.

Another algorithm is the *escape time algorithm*. For each point of the image pre-images are calculated recursively. If all pre-images remain within a bounded area, the initial point is in $\mathcal{A}$; if not, the maximal number of itera-

tions until all preimages of the point leave an a priori defined bounding box of the attractor provides a measure for the distance to $\mathcal{A}$ (larger counts correspond to smaller distances). Mapped to a colour scale an intriguing image of the attractor and its environment results.

The deterministic algorithm is a brute force algorithm, the random iteration algorithm requires sometimes millions of points to be evaluated before pixels are covered with enough points to obtain a detailed and stable image of μ, the escape time algorithm also involves a lot of computation. Several other algorithms have been developed [3,5,4,9,10] that are more efficient. For example, it is possible to limit the number of calculated points that fall into any given pixel.

We present an alternative approach to improve the efficiency: We exploit graphics hardware, as can be found in todays commodity consumer PCs.

3 Method

How can we use graphics hardware to render images of IFS codes? Instead of point based calculations as used, e.g., in the random iteration algorithm, we propose an image based approach that lends itself for an implementation with graphics primitives supported by graphics hardware. Consider the following sequence of images:

$$F_0 = 1;$$
$$F_k = \min(1, \sum_{i=1}^{N} s_i w_i(F_{k-1})), k = 1, 2, \ldots.$$

Here $w(F)$ denotes the image F, warped by transformation w. The minimum is taken pointwise, and 1 is here the image with uniform maximal intensity 1.0. Furthermore, s_i denotes a weight factor for the transformed image $w_i(F_{k-1})$. As described later on, different values for these weights can be used to select either rendering of the attractor or of the measure. Each image F_k is generated via operations on complete images, each next image is defined as the sum of intensity scaled and geometrically transformed copies of the previous image. The intensity of the image is clamped to 1, to account for the limited dynamic range of images. An example is shown in figure 1. The transformations are defined as the mapping from a base frame (yellow) to sub frames (white). The figure shows the result after 1 to 5 steps and the final result. For the intermediate steps here a low value for the weight factors s_i was used to show the separate images of the original square more clearly. The process converges quickly. Image F_k contains N^k copies of the original image. The largest copy has size $(\max_{i=1,\ldots,N} D_i)^k$, where D_i is the area of a unit square transformed by w_i, i.e., $D_i = |a_i d_i - b_i c_i|$. Hence, if we render an $M \times M$ size image, after $-\log M / \log(\max_{i=1,\ldots,N} D_i)$ iterations the copies are smaller than a pixel in area. Likewise the diameters of the copies shrink and converge to zero.

Various settings for the weight factors s_i can be used. If all $s_i = 1$, the result will be an image of the attractor, assuming that all w_i are non-singular, i.e., all $D_i > 0$. If we use $s_i = p_i/D_i$ (and omit the clamping to the maximal intensity 1.0) an image of the measure μ will result. The division by D_i denotes the contraction of mass as a result of the transformation. It is necessary to include this here because warping in the graphics sense does not preserve measure in the mathematical sense. A standard setting for the probabilities, such that mass is transported "uniformly" over the transformed images is to

set $p_i = D_i / \sum D_i$.

This algorithm can be translated almost directly into graphics operations: A sequence of images is generated by blending transformed copies of the previous image. The current image F is stored in the frame buffer. The algorithm now proceeds as follows:

Fill F with white;
repeat
    Copy F to texture memory;
    Clear F with black;
    for $i := 1, 2, \cdots, N$ **do**
        Calculate a quadrilateral $R = w_i(\square)$;
        Render R, texture mapped with the previous image
            and scaled with s_i, and accumulate into F.

This algorithm can be implemented using standard OpenGL1.1 calls [6]. The scale factors s_i can be larger than 1. In OpenGL the scale factors for intensities are restricted to values between 0 and 1, but this can easily be handled by rendering R multiple times.

Does this algorithm lead to a stable image? One condition for this is that in the long run *mass*, i.e., the sum of all pixel values, has to be conserved. Unfortunately, mass can leak away as a result of several effects. Firstly, intensities are clamped to 1, excess intensities are lost. However, for images of μ this has a positive effect. Mass leaks away until the maximum values are 1, thereby optimally using the dynamic range of the display. Mathematically, the resulting image displays an intensity scaled version of the measure.

Secondly, if $\mathcal{A}$ is disconnected, i.e., a cloud of points, the image will be black in the end. A bright point is mapped to darker points, because of the weigths s_i or because of the linear interpolation used during texture mapping. As a result, the image slowly turns black. However, this can be easily compensated for by multiplying the s_i's with an extra factor f, typically 1.01 . Thirdly, the transformations w_i can be singular, i.e., $|a_i d_i - b_i c_i| = 0$. As a result, an image is mapped to an infinitely thin line or even a point, which cannot be handled properly by the graphics hardware. One example where this happens is in Barnsley's IFS code for the Black Spleenwort fern [2]: The stems are the result of such a singular transformation. One remedy is to perturb such transformations slightly, such that they are no longer singular.

Fourthly, and most problematic: mass may be transported outside the viewport. Parts of R can be located outside the image, and mass transported outside will not be reused in the next iteration. One remedy is simply to stick to IFS codes such that the attractor $\mathcal{A}$ fits completely within the screen. A second option is to render only subsets of $\mathcal{A}$ that do fit inside the image. This can be done by using, e.g., a set of conjugate transformations $w_i' = w \circ w_i \circ w^{-1}$ where w denotes a suitably chosen composition of affine IFS maps w_i such that $w(\mathcal{A})$ fits within the screen.

Given these limitations, are there positive aspects as well? Fortunately there are. In the following sections we present a number of variations and extensions.

3.1 Colour

We discussed so far gray scale imagery. However, all graphics procedures operate on (red, green, blue) tuples. We can exploit this, like others have done before also, by using different values for s_i per colour component, or, in other words, to show different probability distributions in a single image. Specifically, we implemented this by assigning a user definable colour (R_i, G_i, B_i) to each transformation. The weights s_{Ri} for red are now defined by

$$s_{Ri} = f\frac{R_i}{\sum_{i=1}^{N} R_i}/|a_i d_i - b_i c_i|,$$

for the other components similarly. An example is shown in figure 2. An IFS code with five transformations is used. On the left a gray scale image is shown, using white for all transformations. In the center image we switched three transformations to red, green, and blue, as shown by the colour of the local origin of the frame. Parts of the attractor disappear (combinations of coloured transformations), and the remaining effect of each coloured transformation is clearly visible. The use of desaturated colours, shown on the right, gives a more subtle effect.

3.2 Dynamic IFS's

So far we assumed that the transformations w_i were given and static, but these transformations can be dynamic as well. One application is educational. We have implemented an IFS modelling system, where the user can define transformations by dragging, rotating, and scaling frames. This can be used to illustrate for instance the collage theorem. Another application is recreational.

Standard algorithms for rendering $\mathcal{A}$ generate each image from scratch. The method presented here however exploits frame to frame coherence. The attractor $\mathcal{A}$ is continuously dependent on the transformations w_i, i.e., small changes of w_i lead to small changes of $\mathcal{A}$ [2]. If a good guess is available for $\mathcal{A}$, the iteration will converge quickly. If we now use a time varying set of w_i's while generating images, the result is a smoothly varying sequence of images.

Automatically changing transformations lead to fascinating animations of fractal imagery. It is hard to capture this in static images, an attempt is shown in figure 3. Animations can be found on the web [8]. The animation was generated simply by rotating each sub frame slowly with a different speed. The snap shots shown are taken with an interval of 25 images. Hence, at fifty frames per second the image on the left smoothly transforms into the image on the right in one second.

3.3 Condensation

In order to visualize the IFS-attractor $\mathcal{A}$ in an even stronger fashion, one may colour its surroundings in a meaningful way. The area outside $\mathcal{A}$ is not uniform: Some points will be closer to $\mathcal{A}$ than others. Usually, the escape time algorithm is used to visualize this. Similar images can easily be generated with our method by using:

$$F_k := \min(1, G + \sum_{i=1}^{N} s_i w_i(F_{k-1})),$$

where G is a fixed image. A closely related concept (except for the clamping) are IFS's with condensation [2]. In graphics terms, each time after clearing the screen we render some image on the screen. As a result, the screen will be

covered by the sum of a hierarchy of transformed copies of G:

$$F = G + \sum_{i=1}^{N} s_i w_i(G) + \sum_{i=1}^{N} s_i w_i \left(\sum_{j=1}^{N} s_j w_j(G) \right) + \cdots .$$

Different images G give different effects, especially, the continuity of G determines the continuity of the resulting image F. Figure 4 shows examples. On the left the attractor of a two transformation IFS code is shown. For the image in the center we used for G a large circle, black on the boundary and interpolating linearly to dark orange in the center. The resulting image is smooth, only discontinuities in the first derivative show up. For the image on the right we used a circle again, but now with the colours reversed. As a result, the discontinuities in intensity clearly stand out.

4 Results

We have implemented the preceding methods in an interactive system. The application was implemented in Delphi 5, using ObjectPascal. It consists of about 2,000 lines of code, most of which concerned user interfacing. Example animations and a copy of this program with example files can be downloaded [8]. In table 1 we enumerate the transformations used for the various examples shown, as well as the scaling factors used, split up in the three color components r_i, g_i, and b_i.

All images presented were made on a Dell Inspiron 8100 notebook PC, running Windows 2000, with a Pentium III 866 MHz processor, 256 MB memory, and a nVidia GeForce2Go graphics card with 32 MB memory. The typical framerate that we achieve is 50 frames per second for 512×512 images for up to twelve

transformations. This high performance comes from three factors. Firstly, all steps are expressed in graphics operations which can be performed quickly by nowadays hardware. Secondly, frame to frame coherence is exploited. Finally, only a few commands per image have to be passed from the CPU to the graphics hardware, hence the bandwidth between CPU and graphics hardware is not a bottleneck.

For comparison purposes we have made a straightforward implementation of the random iteration algorithm to produce images of the measure. Note that for images of the attractor far more efficient algorithms have been developed (see section 2). Some results are shown in table 2. Obviously, timings depend on implementation details, but nevertheless, they point out some tendencies. Firstly, not only calculation of the points, but also filling the array with 0's and normalizing the result take time (first row, 50.0 fps) as well as sending the image to the graphics board (second row, 18,9 fps). The image quality depends on the number of points used. We used the example of figure 2 here, and found that here about 5,000,000 points were needed to obtain an optically stable image. The frame rate then drops to below one frame per second. Our method required about 20-30 iterations for a stable image, starting from scratch, so in this case the order of performance is comparable. However, for dynamic IFS's each new frame is a good approximation, hence a much higher performance is achieved here.

5 Conclusion

We have shown how the rendering of fractal imagery can be accelerated using commodity graphics hardware. For dynamic IFS codes a frame rate of fifty

512×512 frames per second can be achieved on a notebook PC. Furthermore, the algorithm can easily be implemented using standard OpenGL1.1 calls, and a variety of effects, such as the use of colour to visualize different subsets of $\mathcal{A}$ and the rendering of the neighborhood of $\mathcal{A}$ can be realized. A limitation is that zooming is limited, the complete attractor has to fit in the image.

An alternative route for the future is to exploit the capabilities of programmable graphics hardware. Olano and Lastra [7] have shown how the Mandelbrot set can be computed using pixel shaders and multiple passes. IFS attractors could also be calculated similarly. This would remove the limitation on zooming, at the expense of a more involved and less portable implementation.

Many fractal images have been generated and shown since the 1980'ies, and it seems that many people have seen already enough of them. However, animations of high resolution dynamic fractal imagery, displayed in real time under user control, are novel and highly fascinating to watch. Furthermore, the technique is very suitable for educational purposes. Watching the effect of changes to transformation in real-time enables students to understand the underlying concepts more easily.

Furthermore, we see this work as a first step towards other applications of commodity graphics hardware to accelerate fractal based operations, such as image compression and texture synthesis.

References

[1] M. Barnsley. Fractal modelling of real world images. In H.-O. Peitgen and D. Saupe, editors, *The Science of Fractal Images*, pages 219–242. Springer-

Verlag, 1988.

[2] M. Barnsley. *Fractals Everywhere*. Academic Press, 1988.

[3] S. Dubuc and A. Elqortobi. Approximations of fractal sets. *Journal of Computational and Applied Mathematics*, 29:79–89, 1990.

[4] D. Hepting and J.C. Hart. The escape buffer: Efficient computation of escape time for linear fractals. In *Proceedings Graphics Interface'95*, pages 204–214, May 1991.

[5] D. Hepting, P. Prusinkiewicz, and D. Saupe. Rendering methods for iterated function systems. In H.-O. Peitgen, J. M. Henriques, and L. F. Peneda, editors, *Fractals in the Fundamental and Applied Sciences*. North-Holland, Amsterdam, 1991.

[6] M.Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL(R) Programming Guide, Version 1.2 (3rd Edition)*. Addison-Wesley, 1999.

[7] M. Olano and A. Lastra. A shading language on graphics hardware: The pixelflow shading system. In *Proceedings of SIGGRAPH'98*, pages 159–168, 1998.

[8] J.J. van Wijk. `http://www.win.tue.nl/~vanwijk/ibifs`, 2003.

[9] N. Wadströmer. *Coding of Fractal Binary Images with Contractive Set Mappings Composed of Affine Transformations*. PhD thesis, Linköping University, 2001. Linköping Studies in Science and Technology, Dissertation No. 700.

[10] N. Wadströmer. An automatization of barnsley's algorithm for the inverse problem of iterated function systems. *IEEE Transactions on Image Processing*, 12(11):1388–1397, 2003.

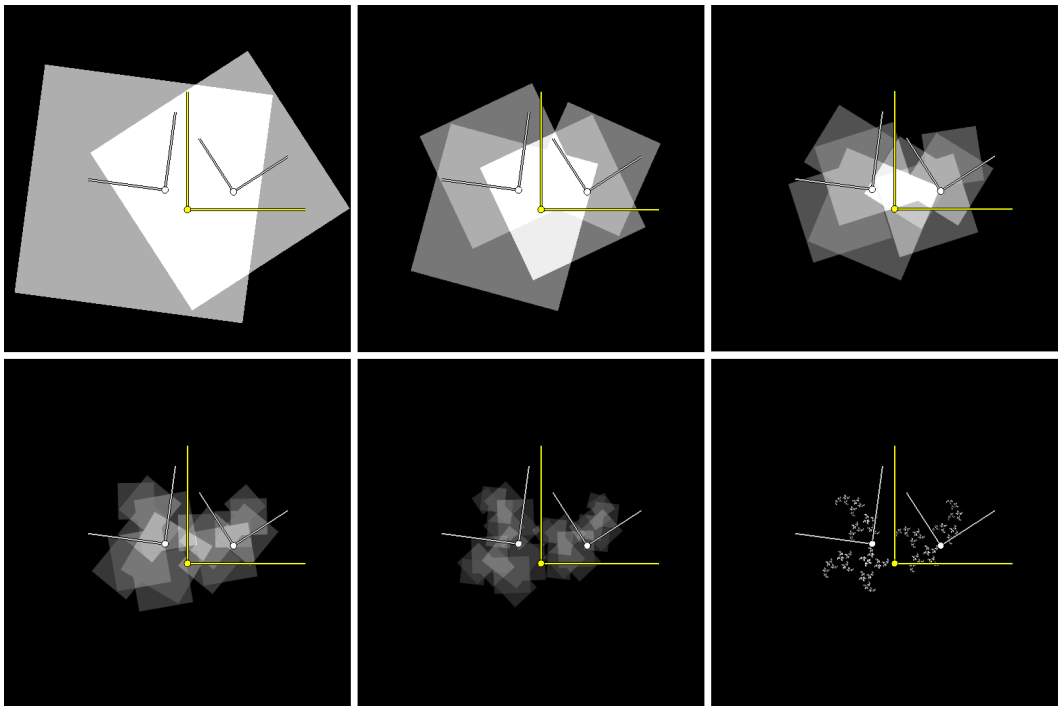

Fig. 1. Iterated mapping of an image

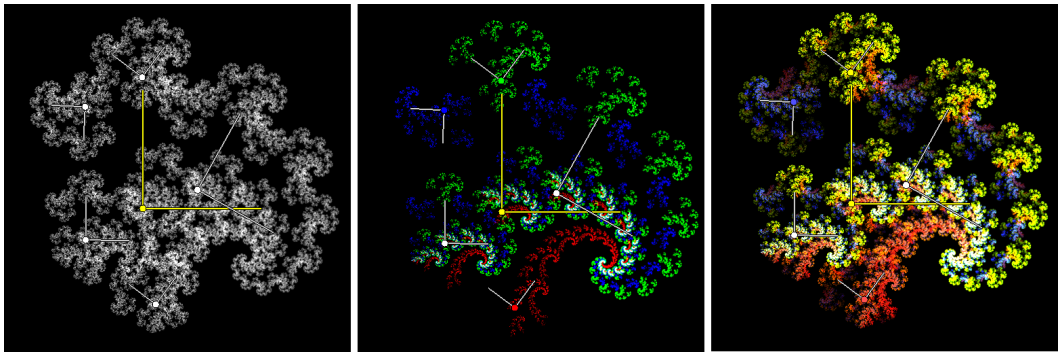

Fig. 2. Coloured transformations

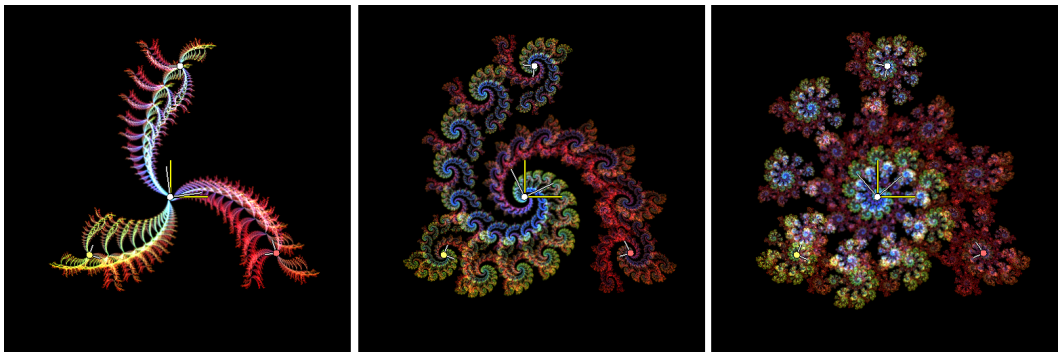

Fig. 3. Dynamic transformations

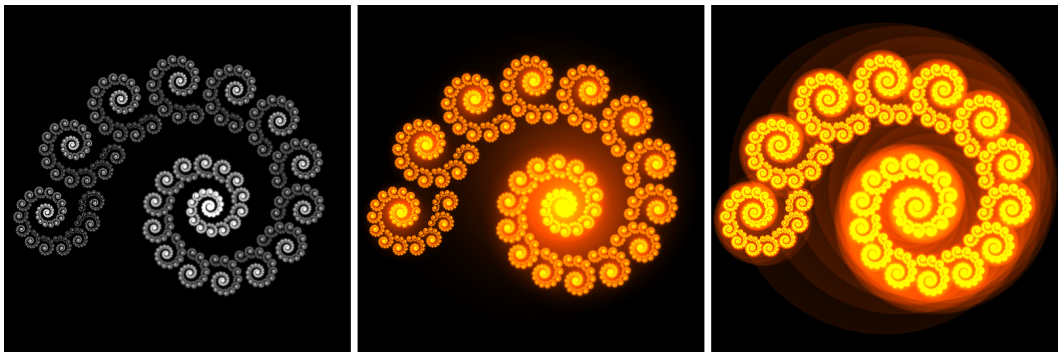

Fig. 4. Insertion of a background image

Example	i	a_i	b_i	c_i	d_i	e_i	f_i	r_i	g_i	b_i
Fig. 2c	1	0.6416	0.3591	-0.3591	0.6416	0.1480	0.3403	1.1901	1.2867	1.3398
	2	0.1906	-0.2554	0.2554	0.1906	0.4162	0.6122	1.1901	1.2867	0.0000
	3	0.1681	-0.2279	0.2279	0.1681	0.4531	-0.0205	1.1901	0.4087	0.4256
	4	-0.2848	-0.0141	0.0141	-0.2848	0.3362	0.8164	0.3780	0.4087	1.3398
	5	0.3672	0.0051	-0.0051	0.3672	0.0776	0.1726	1.1901	1.2867	1.3398
Fig. 3b	1	0.7155	-0.4589	0.4589	0.7155	0.3412	-0.0939	0.9988	1.0624	1.1281
	2	0.2362	-0.1849	0.1849	0.2362	0.2160	0.0852	0.9988	1.0624	0.4910
	3	0.2819	0.1849	0.1025	-0.3205	0.5670	0.3792	0.9988	0.4625	0.4910
	4	0.1080	0.2799	-0.2799	0.1080	0.3303	0.9098	0.9988	1.0624	1.1281
Fig. 4 a	1	-0.2960	0.0469	-0.0469	-0.2960	0.3791	0.5687	1.0602	1.0602	1.0602
	2	0.8302	0.4091	-0.4091	0.8302	-0.0674	0.3319	1.0602	1.0602	1.0602

Table 1

Transformations and scaling for examples

fps	description
50.0	0 points, initialization, no rendering
19.8	0 points, initialization and rendering
11.9	100,000 points
3.0	1,000,000 points
0.7	5,000,000 points

Table 2

Frames per second, 512×512 images, random iteration