COMPARING RENDERING METHODS FOR JULIA SETS

V. Drakopoulos

Department of Informatics and Telecommunications, Theoretical Informatics
University of Athens, Panepistimioupolis
157 84, Athens
Greece
vasilios@di.uoa.gr http://cgi.di.uoa.gr/~vasilios

ABSTRACT

Sequential rendering methods for the graphical representation of Julia sets are compared. Two groups of methods are presented. In the first, the attractor of the Julia set is rendered and, in the second, the complement of the attractor is rendered. Examples of images obtained using these methods are also given.

Keywords: Attractor, Julia set, rendering methods

1 INTRODUCTION

One fascinating aspect of fractals is the beauty of their graphical representation. This paper is devoted to a discussion of various fractal aspects involved in the polynomial $p_c\colon \mathbb{C} \to \mathbb{C}$ with

$$p_c(z) = z^2 + c, \quad c \in \mathbb{C}. \tag{1}$$

The dynamics of p_c is an enormously rich fountain of fractal structures. Although the fractal sets generated from the above-mentioned transformation have been discussed extensively in the literature, as far as we know, no previously published work exists that comprises the best known sequential visualisation methods and whose scope is the comparison of their performances. In order to present these methods, we must first introduce some useful terminology.

A *periodic orbit* or *cycle* is a set of $k \geq 2$ distinct points $\{a_1, \ldots, a_k\}$ such that

$$p_c(a_1) = a_2, \ldots, p_c(a_{k-1}) = a_k, p_c(a_k) = a_1;$$

so, in fact, for each $j = 1, 2, \ldots, k$, $z = a_j$ is a solution of $p_c^k(z) = z$, where $p_c^k(z) = p_c(p_c^{k-1}(z))$. Hence, a point a is *periodic*, if $p_c^k(a) = a$ for some $k > 0$; it is *repelling, indifferent* or *attracting* depending on whether $|(p_c^k)'(a)|$ is greater than, equal to or less than one, respectively. If $|(p_c^k)'(a)| = 0$, a is termed *superattracting.* If $k = 1$, z is called a *fixed point* of p_c. Naturally, attracting means that points z_0 near a will generate orbits

$$z_0 \mapsto z_1 \mapsto z_2 \mapsto z_3 \ldots$$

$z_{k+1} = p_c(z_k)$, $k = 0, 1, \ldots$, which approach a. By collecting all such points one obtains the *basin of attraction* of an attracting fixed point a

$$A_c(a) = \{z \in \mathbb{C} : \lim_{k \to \infty} p_c^k(z) = a\}. \tag{2}$$

It is obvious that ∞ is an attracting fixed point of p_c. The boundary of $A_c(\infty)$ is denoted by $\partial A_c(\infty)$ and is called the *Julia set* of p_c. We also use the symbol $J_c = \partial A_c(\infty)$. Other than $A_c(\infty)$ and J_c, also to be considered is a third object

$$\begin{aligned} K_c &= \mathbb{C} \setminus A_c(\infty) \\ &= \{z \in \mathbb{C} : p_c^k(z) \text{ stays bounded for all } k\} \end{aligned}$$

sometimes called the *filled-in Julia set.* Obviously, we have that

$$\partial K_c = J_c = \partial A_c(\infty),$$

i.e., J_c separates competition between orbits being attracted to ∞ and orbits remaining bounded as $k \to \infty$.

The rest of this paper is organised as follows. Firstly, after describing briefly the most widely used sequential methods for constructing Julia sets, we present efficient sequential algorithms for rendering purposes. As examples we give sequential algorithms in the form of ready-to-use code to attack the problem of determining the Julia set by inverse iterating and by examining the nearest neighbour pixels. Next, we compare all the implemented sequential methods with each other in order to find the best balance between speedup and accuracy. Finally, some conclusions are drawn along with a discussion of implementational issues.

2 VISUALISATION METHODS REVISITED

We consider methods representing Julia sets as they result from iterating the complex quadratic polynomial (1). The methods for rendering Julia sets are diagrammatically represented in Fig. 1. For clarity, these methods are subdivided into two

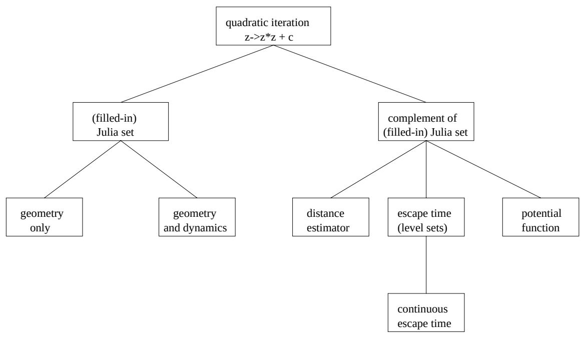

Figure 1: An overview of the methods for rendering Julia sets

groups: those for the (filled-in) Julia sets and those for the corresponding complements. In a particular picture both aspects are usually combined. There are three approaches to the last group, namely methods representing *Euclidean distance* from the filled-in Julia set; *repelling* methods, computing the escape time of a point from K_c and methods using (electrostatic) *potential functions* of the K_c. The last two methods are equivalent, the *escape time* is proportional to the logarithm of the *potential function*. For a more detailed study of Julia sets and the sequential algorithms for rendering them an interesting reference is [Peitgen88].

2.1 Inverse Iteration Method

In general, it is not obvious at all how to obtain a reasonable picture of J_c, though there is an immediate algorithm (*Inverse Iteration Method - IIM*) obtained from the following characterisation due to Julia and Fatou: For any c, the equation $p_c(z) = z$ has two finite solutions $u_0 \neq \infty \neq v_0$ - the fixed points. If $c \neq 1/4$, then at least one of them is a repelling fixed point, say u_0. Then one has

$$J_c = \overline{\{z \in \mathbb{C} : p_c^k(z) = u_0 \text{ for some } k \in \mathbb{Z}\}}.$$

Note that, in general, $p_c^k(z) = u_0$ has 2^k solutions, i.e. the total number of iterated preimages of u_0 obtained by recursively solving the equation $z^2 + c = u_0$ is

$$n(k) = 2^{k+1} - 1, \quad k = 0, 1, \ldots.$$

The recursion is nicely represented in a binary tree as in Fig. 2(a). For the whole tree one needs

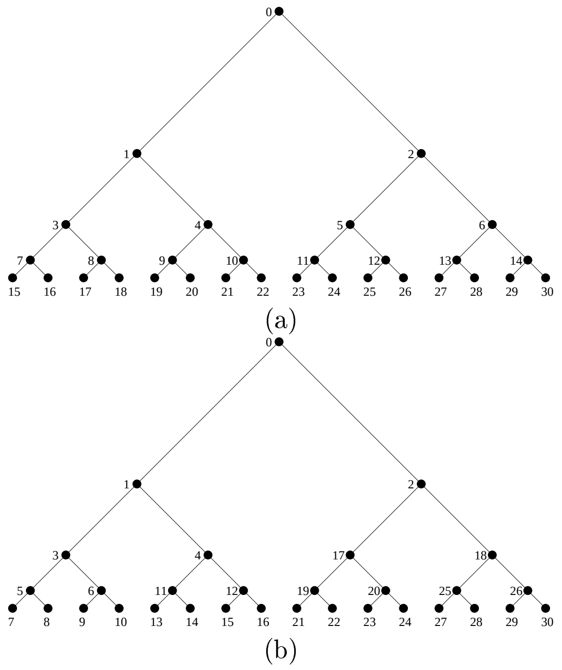

Figure 2: Binary tree structures

all 2^k preimages of the level k in order to compute level $k + 1$. If one, however, anticipates that N iterations suffice, then there is an obvious way to label the tree as in Fig. 2(b) (depth-first search), which requires only $2(N - 1)$ (as compared to 2^{N-1}) units of storage.

Another approach is obtained by choosing one of the two roots at random at each stage of the recursion for preimages. This amounts to a random walk on the tree in Fig. 2(a). Usually the method will work for almost all initial $u_0 \in \mathbb{C}$. The first few preimages will have to be excluded from the plot. Iterated preimages will approximate J_c. Formally, this is a typical example for an *iterated function system* (IFS) with maps

$$w_1(u) = +\sqrt{u-c} \text{ and } w_2(u) = -\sqrt{u-c},$$

where any set A of points so far computed yields a larger set $w_1(A) \cup w_2(A)$. Barnsley in [Barnsley93] and Hepting et al. in [Hepting91] explore this viewpoint in detail.

The IIM is rather fast in providing a first impression of the shape of the Julia set, although for some parameter choices it takes a very long time to obtain all the details (Fig. 3(a)). This is why

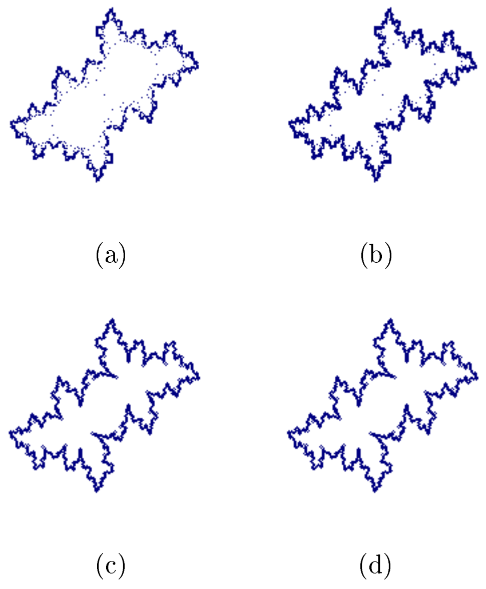
(a) (b) (c) (d)

Figure 3: A Julia set obtained by (a) the IIM, (b) the MIIM, (c) the BSM and (d) the MBSM

variations of IIM or totally different methods are necessary. Note that this method belongs to the second method set of the first group (see Fig. 1). In the case of the IIM the algorithm is:

```
void Julia::IIMethod(CDC *pDC)
{
 int x, y;
 int times = 0;
 int maxdepth = 15;

 Stack<IIMRec> CStack;
 IIMRec Data, root;
 Data.label = 0;
 if (C.Re()==0 && C.Im()==0)
   Data.Z = Complex(1,0);
 CStack.push(Data);
 while (!CStack.isEmpty()){
   CStack.pop(Data);
   x = round((Data.Z.Re()-Xmin)/dx);
   y = round((Data.Z.Im()-Ymin)/dy);
   pDC->SetPixel(x,y,RGB(0,0,100));
   if (Data.label<maxdepth){
     root.Z = sqrtC(Data.Z-C);
     root.label = Data.label + 1;
     CStack.push(root);
     root.Z = (-1)*root.Z;
     CStack.push(root);
   }
 }
}
```

2.2 Modified Inverse Iteration Method

A detailed mathematical motivation is given in [Peitgen86], pp. 37–38. The idea of the algorithm is to make up for the nonuniform distribution of the complete tree of iterated preimages by selecting an appropriate subtree, which advances to a much larger level of iteration k and forces preimages to hit sparse areas more often.

Put J_c on a square lattice with small mesh size β. Then for any box B of that mesh, stop using points from B for the preimage recursion, provided a certain number N_{max} of such points in B have been used. Optimal choices of B and N_{max} depend very much on J_c and other computergraphical parameters, such as the pixel resolution of the given system.

Another variant attempts to estimate the contractiveness of w_1 and w_2 (see IIM). Given any point $u_{m_k} \neq u_0$ on the k-th level of the binary tree in Fig. 2(a) there is a unique path on the tree from u_{m_k} to u_0 which is determined by the forward iteration of u_{m_k} (k times): $p_c^k(u_{m_k}) = u_0$. Now, the idea is to stop using u_{m_k} in the preimage recursion (i.e. to cut off the subtree starting at u_{m_k}), provided that the derivative

$$|(p_c^k)'(u_{m_k})| = \left|\prod_{i=1}^{k} p_c'(u_{m_i})\right|$$

exceeds some bound $D \in [0,1)$, which is the parameter of the algorithm. Here we have written

$$u_{m_i} = p_c^{k-i}(u_{m_k}), \quad i = 0, \ldots, k.$$

Of course, the above derivatives can be cheaply accumulated in the course of the recursion:

$$NewDerivative = 2 \cdot OldDerivative \cdot |u_{m_i}|.$$

In the case of the MIIM the algorithm is:

```
void Julia::MIIMethod(CDC *pDC)
```

```
{
 int x, y;
 long iter = 0;

 Stack<MIIMRec> CStack;
 MIIMRec Data, root;
 Data.label = 0;
 Data.Deriv = 1;
 if (C.Re()==0 && C.Im()==0)
   Data.Z = Complex(1,0);
 CStack.push(Data);
 while (!CStack.isEmpty()
        && ++iter<SENTINEL){
   CStack.pop(Data);
   x = round((Data.Z.Re() - Xmin)/dx);
   y = round((Data.Z.Im() - Ymin)/dy);
   pDC->SetPixel(x,y,RGB(0,0,100));
   if (Data.label<MAXDEPTH
       && Data.Deriv<Dbound){
     root.Z = sqrtC(Data.Z-C);
     root.label = Data.label + 1;
     root.Deriv = 2*Data.Deriv
                  *root.Z.abs();
     CStack.push(root);
     root.Z = (-1)*root.Z;
     CStack.push(root);
   }
 }
}
```

2.3 Boundary Scanning Method

The *Boundary Scanning Method*, or *BSM* for short, is even more elementary than IIM. It uses the definition of K_c, Eq. (3), and $A_c(\infty)$, Eq. (2), in a straightforward manner.

Similar to MIIM, this method is based on a lattice - let's assume a square lattice of mesh size β, which could be just the pixel lattice. Choose N_{max} - a large integer - and R - a large number. Now let q be a typical pixel in the lattice with vertices $v_i, i = 1, 2, 3, 4$. The algorithm consists in a labeling procedure for the v_i's:

v_i is labelled 0,	provided $v_i \in A_c(\infty)$;
v_i is labelled 1,	provided $v_i \in K_c$.

Then q is called *completely labelled*, provided the vertices of q have labels which are not all the same. A good approximation of J_c is obtained by coloring all completely labelled pixels in the lattice (Fig. 3(c)). Thus it remains to decide whether $v_i \in A_c(\infty)$. The answer is yes, provided that $|p_c^k(v_i)| > R$ for some $k \le N_{max}$. Otherwise, it is assumed that $v_i \in K_c$. Note that BSM belongs to the first method set of the first group (see Fig. 1). In the case of the BSM the algorithm is:

```
int Julia::SetLevel
           (double x, double y)
{
 double SQRx, SQRy, temp;
 int iter = 0;

 SQRx = x*x;
 SQRy = y*y;
 for(;(iter<Nmax)
     && (SQRx+SQRy<Rmax);iter++){
   temp = SQRx-SQRy+C.Re();
   y = 2*x*y+C.Im();
   x = temp;
   SQRx = x*x;
   SQRy = y*y;
 }
 return iter;
}

int Julia::CompletelyLabelled
           (double x, double y)
{
 int labelledpixels=0;
 if (SetLevel(x,y+dy)==Nmax)
   labelledpixels++;
 if (SetLevel(x,y-dy)==Nmax)
   labelledpixels++;
 if (SetLevel(x+dx,y)==Nmax)
   labelledpixels++;
 if (SetLevel(x-dx,y)==Nmax)
   labelledpixels++;
 return (labelledpixels<4
         && labelledpixels>0);
}

Rec Julia::PointInSet(int i, int j)
{
 int OnBoundary=0;
 double x, y;
 Rec R;
 Complex Z0 = RFP(C);
 R.i = round((Z0.Re()-Xmin)/dx);
 R.j = round((Z0.Im()-Ymin)/dy);
 if (!CompletelyLabelled(R.i, R.j)){
   i = 2*MAXROW/3;
   y = Ymin;
   x = Xmin + i*dx;
   for (j=1; j<MAXCOL
        && !OnBoundary; j++){
     y += dy;
     OnBoundary =
       CompletelyLabelled(x,y);
   }
   R.i = i; R.j = j-1;
 }
 return R;
}
```

```
void Julia::BSMethod(CDC *pDC)
{
 double x, y;

 x = Xmin;
 for(int i=1;i<=MAXCOL;i++){
   x+=dx; y=Ymin;
   for(int j=1;j<=MAXROW;j++){
     y+=dy;
     if (CompletelyLabelled(x,y))
       pDC->
       SetPixel(i,j,RGB(0,0,100));
   }
 }
}
```

2.4 Modified Boundary Scanning Method

It is obvious that scanning all pixels of a lattice will be very time consuming, in particular for pixels inside K_c. If J_c is connected, a much more economical algorithm is obtained in the following way. Assume that q_0 is a pixel in the lattice which is completely labelled. Pixel q_0 is used as a seed for a neighbourhood search process: Move all (immediately) neighbouring pixels of q_0 onto a stack. Then test each pixel in the stack in three steps:

1. compute labels of vertices of a pixel from the stack;

2. index whether pixel is completely labelled as in BSM;

3. if last pixel is completely labelled, push all those (immediate) neighbours that have not been tested before onto the stack (Fig. 3(d)).

In the case of the MBSM the algorithm is:

```
void Julia::MBSMethod(CDC *pDC)
{
 Image JSet(MAXCOL, MAXROW);
 Stack<Rec> stack;
 double X, Y;
 Rec Z = PointInSet();
 Rec adjZ;

 stack.push(Z);
 while (!stack.isEmpty()) {
   stack.pop(Z);
   X = Xmin + Z.i*dx;
   Y = Ymin + Z.j*dy;
   if (CompletelyLabelled(X,Y)){
     pDC->
     SetPixel(Z.i,Z.j,RGB(0,0,100));
     JSet(Z.i, Z.j)=1;
     adjZ.i = Z.i; adjZ.j = Z.j+1;
     X = Xmin + adjZ.i*dx;
     Y = Ymin + adjZ.j*dy;
     if (JSet(adjZ.i,adjZ.j) == 0)
       stack.push(adjZ);
     adjZ.i = Z.i; adjZ.j = Z.j-1;
     X = Xmin + adjZ.i*dx;
     Y = Ymin + adjZ.j*dy;
     if (JSet(adjZ.i,adjZ.j) == 0)
       stack.push(adjZ);
     adjZ.i=Z.i+1; adjZ.j=Z.j;
     X = Xmin + adjZ.i*dx;
     Y = Ymin + adjZ.j*dy;
     if (JSet(adjZ.i,adjZ.j) == 0)
       stack.push(adjZ);
     adjZ.i = Z.i-1; adjZ.j = Z.j;
     X = Xmin + adjZ.i*dx;
     Y = Ymin + adjZ.j*dy;
     if (JSet(adjZ.i,adjZ.j) == 0)
       stack.push(adjZ);
     adjZ.i = Z.i+1; adjZ.j = Z.j+1;
     X = Xmin + adjZ.i*dx;
     Y = Ymin + adjZ.j*dy;
     if (JSet(adjZ.i,adjZ.j) == 0)
       stack.push(adjZ);
     adjZ.i = Z.i+1; adjZ.j = Z.j-1;
     X = Xmin + adjZ.i*dx;
     Y = Ymin + adjZ.j*dy;
     if (JSet(adjZ.i,adjZ.j) == 0)
       stack.push(adjZ);
     adjZ.i = Z.i-1; adjZ.j = Z.j-1;
     X = Xmin + adjZ.i*dx;
     Y = Ymin + adjZ.j*dy;
     if (JSet(adjZ.i,adjZ.j) == 0)
       stack.push(adjZ);
     adjZ.i=Z.i-1; adjZ.j=Z.j+1;
     X = Xmin + adjZ.i*dx;
     Y = Ymin + adjZ.j*dy;
     if (JSet(adjZ.i,adjZ.j) == 0)
       stack.push(adjZ);
   }
 }
}
```

2.5 Level Set Method

The *Level Set Method*, or *LSM* for short, also called the *Escape Time Method*, is just a very powerful variant of BSM that causes J_c to stand out against a spectrum of colour bands approaching from without or within (see [Hoggar92]). We fix a square lattice of pixels, choose a large integer N_{max} (iteration resolution) and an arbitrary set T (target set) containing ∞, so that $K_c \subset \mathbb{C} \setminus T$. For example, $T = \{z \in \mathbb{C} : |z| \geq 1/\varepsilon\}$, ε small, is a disk around ∞. Now we assign for each pixel

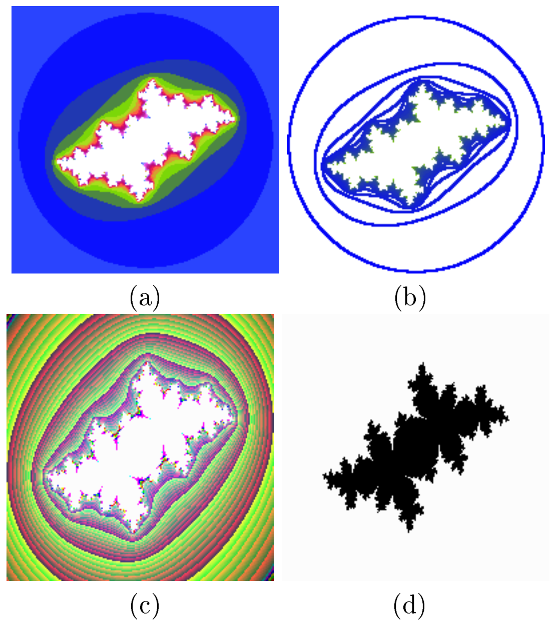

Figure 4: A Julia set obtained by (a) the LSM, (b) the LSM but showing the border of the encirclements, (c) the CPM and (d) the DEM

q from the lattice an integer label $l_c(q;T)$ in the following way:

$$l_c(q;T) = \begin{cases} k, & \text{provided } p_c^i(q) \notin T \text{ and} \\ & p_c^k(q) \in T \text{ for } 0 \leq i < k \\ & \text{and } k \leq N_{max} \\ 0, & \text{otherwise.} \end{cases}$$

The interpretation of a nonzero $l_c(q;T)$ is obvious: q escapes to ∞ and $l_c(q;T)$ is the "escape time", measured in the number of iterations, needed to hit the target set T around ∞. The collection of points of a fixed label, say k, constitutes a *level set* (Fig. 4(a),(b)).

2.6 Continuous Potential Method

The *Continuous Potential Method*, or *CPM* for short, allows to represent the potential of K_c as a smooth parameterised surface

$$pot_c: \mathbb{C} \setminus K_c \to \mathbb{C} \times \mathbb{R},$$

which is approximately given by

$$pot_c(z_0) = \left(z_0, \frac{\log |z_n|}{2^n} \right),$$

where $z_k = z_{k-1}^2 + c$, $k = 1, 2, \ldots, n$, $n = l_c(z_0;T)$ and $T = \{z \in \mathbb{C} : |z| \geq 1/\varepsilon\}$ for small ε (Fig. 4(c)). Note that the level curves of pot_c are circle-like far outside.

2.7 Distance Estimator Method

The *Distance Estimator Method*, or *DEM* for short, usually applies for z near K_c (connected); see Fig. 4(d). Let c be fixed. Choose N_{max} and $R = 1/\varepsilon$, where $T = \{z \in \mathbb{C} : |z| \geq 1/\varepsilon\}$ for small ε is the target set around ∞. For each z_0 we will determine a label $l(z_0)$ from $\{0, \pm 1, 2\}$ (0 for $z_0 \in K_c$, $\{+1, -1\}$ for z_0 close to K_c, 2 for z_0 not close to K_c): Compute

$$z_{k+1} = z_k^2 + c, \quad k = 0, 1, 2, \ldots$$

until either $|z_{k+1}| \geq R$ or $k = N_{max}$. In the second case we set $l(z_0) = 0$. In the other case we have $|z_n| \geq R$ with $n = k + 1 = l_c(z_0;T)$ and z_0 is still candidate for a point close to K_c. Thus we try to estimate its distance having saved the orbit $\{z_0, z_1, \ldots, z_n\}$:

$$z'_{k+1} = 2 z_k z'_k\ ,\ z'_0 = 1,\ k = 0, 1, \ldots, n-1. \quad (3)$$

If in the course of the iteration of Eq. (3) we get an overflow, i.e. if

$$|z'_{k+1}| \geq OVERFLOW$$

for some k, then z_0 should be very close to K_c, thus we label z_0 by -1. If no overflow occured, then we estimate the distance of z_0 from K_c by

$$d(z_0, K_c) = 2 \frac{|z_n|}{|z'_n|} \log |z_n|$$

and set

$$l(c) = \begin{cases} 1, & \text{if } d(z_0, K_c) < DELTA \\ 2, & \text{otherwise.} \end{cases}$$

3 COMPARATIVE RESULTS

We compared the above mentioned algorithms by evaluating two of their basic characteristics: the *speed* with which they compute the corresponding Julia set and the *efficiency* with which they display it to the computer screen. The complex number used in all cases was the "difficult" value $c = -0.48176 - 0.53165\imath$.

Table 1 presents the sequential runtime measured for each of the seven methods (IIM, MIIM, BSM, MBSM, LSM, CPM, DEM) for the computation of the Julia set. A first observation from these results concerns the increase of the runtimes, while increasing the resolution (Res row) of the images. A second observation concerns the low runtime obtained for the BSM and MBSM methods; the latter is obviously an improvement of the former. The IIM and the MIIM are the fastest methods, but, of course, it depends upon the number of

Method\Res	R_1	R_2	R_3
IIM	2	6	10
MIIM	1	6	12
BSM	3	18	44
MBSM	3	10	18
LSM	1	7	16
CPM	1	7	15
DEM	4	20	52

Table 1: Total runtime of some methods used for constructing the Julia set; $R_1 = 320 \times 200$, $R_2 = 640 \times 480$, $R_3 = 1024 \times 768$

Algorithm	*Efficiency*
IIM	⋆⋆
MIIM	⋆⋆⋆⋆⋆⋆
BSM	⋆⋆⋆⋆
MBSM	⋆⋆⋆⋆⋆
LSM	⋆⋆⋆
CPM	⋆⋆⋆
DEM	⋆⋆⋆⋆⋆⋆⋆

Table 2: Efficiency of some methods used for constructing the Julia set

points that lie on the attractor. If we want a very accurate picture of the attractor, we are obliged to give a large number of points and the MIIM can become extremely slow. The DEM is the slowest method with a slight difference from the BSM. Of course the result is worth such a delay!

Table 2 presents the efficiency measured for each of the seven methods (IIM, MIIM, BSM, MBSM, LSM, CPM, DEM) for the computation of the Julia set. When we speak about efficiency we mean the quality of the resultant picture, i.e. how accurate the graphical representation of the fractal set is. The more efficient method is the DEM; for that, it is the slowest. Nevertheless, in some cases the MIIM is better than the DEM. Sufficiently satisfactory results are obtained also with the LSM or the CPM.

4 CONCLUSIONS

The current implementation of the algorithms mentioned before is written in Microsoft Visual C++ 6.0. Time results are given in CPU seconds on a Pentium MMX PC with a 200 MHz CPU clock running Windows 98.

As can be easily extracted from the comparison analysis of the preceding section, the MIIM is the best method (over all measures) for rendering Julia sets. It is well known that DEM is one of the more accurate methods to obtain the best quality pictures of these fractal sets. The second best method is the MBSM and then following, in order, the CPM, IIM, LSM and BSM. If one wants to render only the Julia set J_c (and not the filled-in K_c), the MBSM must be chosen. Hence, depending on the sought-after fractal set, a compromise between runtime and accuracy must be made.

REFERENCES

[Barnsley93] Barnsley, M. F.: *Fractals everywhere, 2nd ed., Academic Press Professional*, 1993.

[Hepting91] Hepting, D., Prusinkiewicz, P. and Saupe, D.: *Rendering methods for iterated function systems*, in Peitgen, H.–O., Henriques, J. M. and Penedo, L. F. (eds), *Fractals in the fundamental and applied sciences, North-Holland*, pp. 183–224, 1991.

[Hoggar92] Hoggar, S. G.: *Mathematics for computer graphics, Cambridge Univ. Press*, 1992.

[Peitgen86] Peitgen, H.–O. and Richter, P. H.: *The beauty of fractals, Springer-Verlag*, 1986.

[Peitgen88] Peitgen, H.–O. and Saupe, D. (eds): *The science of fractal images, Springer-Verlag*, 1988.