Santa Teresa
Laboratory
San Jose, CA

Graphics Application Using Complex Numbers in APL2

by James A. Brown
Harlan Crowder

March 1985

TR 03.265

March 1985
TR 03.265

Graphics Applications using Complex Numbers in APL2

by
James A. Brown
Harlan Crowder

International Business Machines Corporation
General Products Division
Santa Teresa Laboratory
San Jose, California

ABSTRACT

This report explains and demonstrates the use of complex numbers in APL2 for two-dimensional graphics applications. We discuss the APL2 concepts of data and data types, complex numbers, arrays, and operations on arrays. We show how complex numbers and arrays of complex numbers can be used for two dimensional computer graphics. Finally, we demonstrate these graphical concepts and techniques, using examples from elementary fractal geometry.

The graphic images for this report were created using the Graphical Data Display Manager running under APL2. Text and graphics were integrated using the Document Composition Facility. The report was produced on the IBM 4250 printer

Introduction

Techniques for using graphics in computer applications is currently a popular topic among software designers and users. APL has traditionally been a good graphics programming tool because graphics data structures are easily created and manipulated using APL arrays and functions, and because APL has provided good interfaces to existing graphics services. In IBM's new APL2[1], the domain of numeric data has been expanded to include complex numbers. This development, in conjunction with the APL2 interface to the Graphical Data Display Manager (GDDM)[2], has implications for APL graphics applications. The purpose of this report is to explore and demonstrate how complex numbers can be used for two-dimensional graphics applications

First, we discuss the APL2 concepts of data and data types, complex numbers, arrays, and operations on arrays. Then we show how complex numbers and arrays of complex numbers can be used for two dimensional computer graphics. Finally, we demonstrate these graphical concepts and techniques, using examples from elementary fractal geometry.

APL2 data and operations

In this section, we describe how APL2 represents data, and, in particular, how complex numbers are used and exhibited. We look at some of the operations that manipulate complex numbers, and develop the concept of general operations on arrays.

APL2 data

APL2 has two kinds of data -- numbers and characters -- from which arrays are structured. The elements of character data are the APL2 character set, for example, 'A' and 'c'. Previous implementations of APL limited numeric data to boolean (0 and 1), integers (e.g., 7), and reals (e.g., 3.14159). APL2 has extended the domain of numeric data to include *complex numbers*. The complex domain is a superset of previous APL numeric data, there are complex number representations for boolean, integer and real numbers.

Real numbers can be thought of as being composed of two parts -- an integer part and a fractional part. These parts are connected by a decimal point (.). Complex numbers may also be thought of as being composed of two parts -- a real part and an imaginary part. These parts are connected by the letter 'J' with the real part on the left and the imaginary part on the right. For example, the complex number 3J4 has real part equal 3 and imaginary part equal 4. Real numbers can be interpreted as complex numbers with imaginary part equal zero. Thus, 5.3 is 5.3J0.

In APL2, a complex number can be specified as *xJy*, where *x* is the real part and *y* is the imaginary part. Complex numbers can also be specified using the magnitude-phase forms *mDp* (for phase in degrees) or *mRp* (for phase in radians). The relationships between these representations is illustrated in Figure 1.

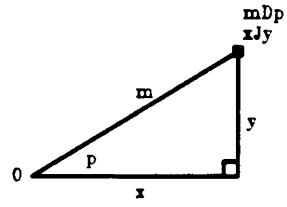

Figure 1: Complex number representations.

The same number can be represented as $x\mathcal{J}y$ or $m\mathcal{D}p$. In the $\mathcal{J}$ form, x and y are displacements along the real (horizontal) and imaginary (vertical) axes, respectively. In the $\mathcal{D}$ form, m is the distance of the complex point from the origin and p is the angular displacement, in degrees, from the horizontal.

In APL2, complex numbers are always displayed using the $\mathcal{J}$ form notation. For example,

```
    4J3
4J3
    5D36.869897
4J3
    5R0.6435011
4J3
```

Operations on complex numbers

All the usual arithmetic operations in APL2 are defined on complex numbers. We will be concerned here primarily with addition and multiplication

Multiplication of complex numbers

Multiplication of complex numbers is best understood using the $\mathcal{D}$ form. If R and S are complex numbers, then their distance from the origin is $|R$ and $|S$, respectively. Their angular relationship to the real axis (*phase*) is $12\bigcirc R$ and $12\bigcirc S$ radians, respectively. The product $R\times S$ has a magnitude, which is the product of the magnitudes $(|R)\times(|S)$, and phase, the sum of the phases $+/12\bigcirc R\ S$. For example,

```
    5D30 × 2D90
¯5J8.66025

    FMTPD 5D30 × 2D90
10D120
```

Here *FMTPD* is a function from the APL2 distributed workspace 1 *MATHFNS* which displays complex numbers in the $\mathcal{D}$ form.

Addition of complex numbers

Addition of complex numbers is best understood when using the $\mathcal{J}$ form. If R and S are complex scalars, then the sum $R+S$ has real part equal to the sum of the real parts of R and S $(+/9\bigcirc R\ S)$ and imaginary part equal to the sum of the imaginary parts of R and S $(+/11\bigcirc R\ S)$. For example,

```
    4J3+3J5
7J8
```

Operations on arrays

An array is a collection of numbers and characters Most APL operations apply to a whole collection all at once This permits us to control the sequencing of operations by arranging the structure of the the data rather than the structure of the program. In this paper. we will only use the APL scalar functions addition. subtraction. and multiplication All the examples here work the same way for any of these functions.

If we write a list of numbers (a *vector*) on each side of a scalar function. the operation is applied independently between pairs of corresponding items. one from each side. For example.

```
     1 2 3 + 10 20 30
```
means
```
     (1+10) (2+20) (3+30)
```
and results in
```
11 22 33
```

If we write a vector on one side but a single number on the other side. the scalar is paired with each item of the vector. For example.

```
     1 + 10 20 30
```
means
```
     (1+10) (1+20) (1+30)
```
and results in
```
11 21 31
```

Any item of an array may itself be an array Such an array is called a *nested array* Here is an example with a vector that contains other vectors as items

```
     1 2 × (10 20 30) (15 16 17)
```
means
```
     (1×10 20 30) (2×15 16 17)
```
and results in
```
10 20 30  30 32 34
```

APL has a way to make any array into a scalar by using the function *enclose* (⊂). In the following example. the array ⊂10 20 30 is a scalar and so is paired with each item of the vector left argument

```
     1 2 3 × ⊂10 20 30
```
means
```
     (1×10 20 30)(2×10 20 30)(3×10 20 30)
```
and results in
```
10 20 30  20 40 60  30 60 90
```

There is another important way to apply functions. *Outer product* (∘.*F*) applies the function *F* to pairs of data one from each side in all combinations.

```
     10 20 ∘.- 1 2 3
 9  8  7
19 18 17
```

"All combinations" in a problem means "outer product" in APL. Outer product applies to both primitive and user-defined functions in APL2.

Complex numbers for two-dimensional graphics

In this section, we describe how points in the plane can be represented as complex scalars, how polygonal objects can be represented as simple complex vectors, and how collections of objects can be represented as nested complex arrays. We also show the geometrical effects of multiplication, addition, and subtraction of complex arrays

Graphical data representation using complex numbers

Real numbers are a special case of complex numbers: reals have imaginary part equal zero. For example, ¯2.5, 1, and 3.1415 are real numbers; they can be represented spatially as points on the real number line, as shown in Figure 2.

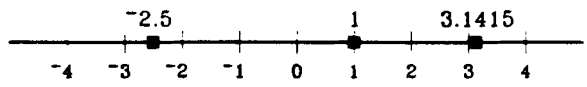

Figure 2: Points on the real number line.

Similarly, complex numbers can be represented as points in the *complex plane*. The real part of a complex number determines its location along the real or horizontal axis, and the imaginary part determines its location along the imaginary or vertical axis. For example, the numbers 3J2, 4, 0J¯2, ¯3J¯1, and ¯2J1 can be represented as illustrated in Figure 3.

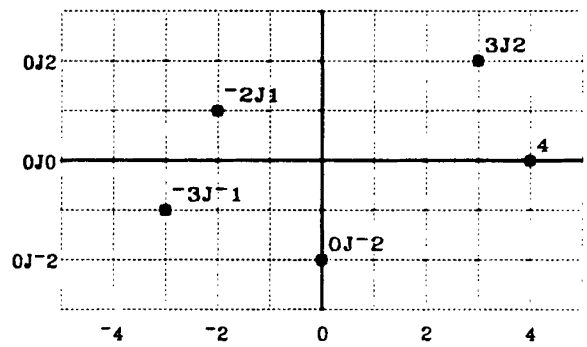

Figure 3: Points in the complex plane.

A common operation in computer graphics is to represent two-dimensional polygonal objects as ordered lists of points in the plane. For example, the equilateral triangle *TRI* is defined as

```
TRI ←→ 0J2 ¯1.732J¯1 1.732J¯1 0J2
```

Since the first and last points are the same, connecting the points results in a closed figure in the complex plane as shown in Figure 4

Similarly, the list *ARROW* is defined as a vector of length 7:

```
ARROW ←→ 0J.25 .5J.25 .5 1J.5 .5J1
         .5J.75 0J.75 0J.25
```

The result of drawing *ARROW* in the complex plane is shown in Figure 5.

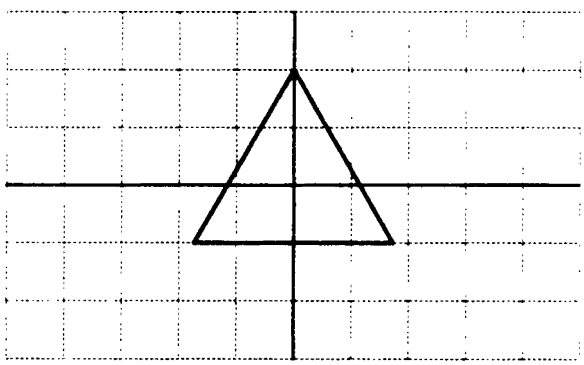

Figure 4: Points of an equilateral triangle in the complex plane.

This idea can be extended by allowing collections of polygons to be represented by nested lists of lists. For example, the depth-2 list *NEST* is defined as

```
T ← ¯4J2 ¯4J¯1 ¯1J¯1 ¯4J2
S ← 1J1 4J1 4J¯2 1J¯2 1J1
NEST ← T S
```

Drawing and shading the two items of *NEST* give the illustration in Figure 6.

We can define a recursive APL2 function *DRAW* that interprets simple arrays as collections of points describing polygonal objects, and nested arrays as collections of such objects. *DRAW* has the following definition:

```
[0]  DRAW A
[1]  →(1<≡A)/L1
[2]  MOVE ¯1↑A←,A
[3]  JOIN 1↓A
[4]  →0
[5] L1: DRAW¨A
```

Line [1] tests the structure of the argument *A*, if *A* is nested, then a transfer of control is made to the recursive call at line [5]. Line [2] causes a move (without drawing) of the graphics pen to the coordinates given by the first scalar in the simple vector *A*. Line [3] joins points given by the remaining elements of *A* with a series of straight lines, starting at the current point. The number of line segments is ¯1+⍴,*A*. Line [4] exits the current level of call to *DRAW*. Line [5] performs the recursive call; if *A* is nested, then *DRAW* is applied to each item of *A*.

The definitions of the generic subfunctions *MOVE* and *JOIN* are appropriate to the graphics management subsystem used by *DRAW*.

Note that this representation and use of complex arrays for two-dimensional polygonal objects lends itself to edge-representation as well as to point-representation. Edge-representation simply requires an extra level of nesting. For example, in point-representation, the unit square is

```
0 1 1J1 0J1 0.
```

The corresponding object in edge-representation is

```
(0 1)(1 1J1)(1J1 0J1)(0J1 0).
```

Applying *DRAW* to these two objects gives the same graphical result.

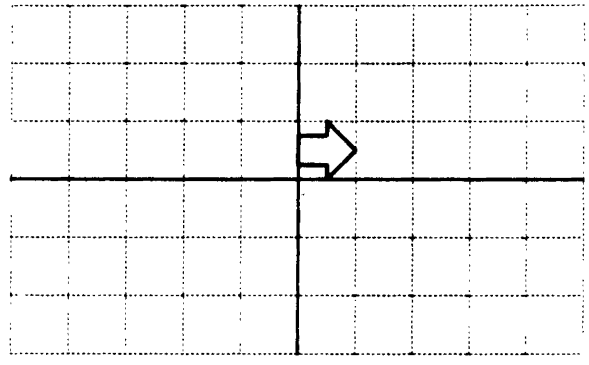

Figure 5: Points of an arrow in the complex plane.

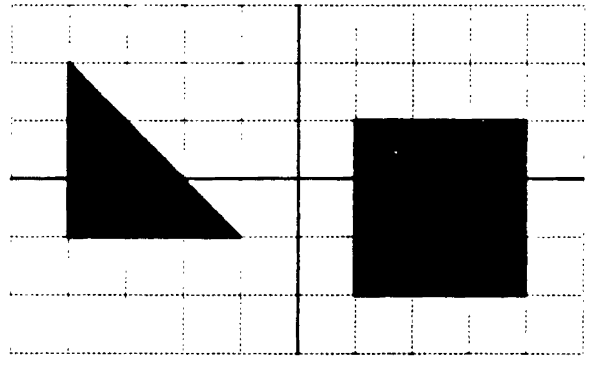

Figure 6: A nested array as a collection of objects.

Complex multiplication

We previously gave a mathematical description of complex multiplication, here is the geometric interpretation If Q is a point in the complex plane, then the product of Q and a complex number in the degree-form mDp gives a result that has distance from the origin of Q altered by a factor m, and has been rotated through p degrees with respect to the origin

For example, if A is $2J1$, then $1D90\times A$ has the same magnitude as A but has been rotated 90 degrees anticlockwise with respect to the origin. The product $2D180\times A$ is twice as far from the origin as A and has been rotated 180 degrees with respect to 0. The product $.5D^{-}120\times A$ is half the distance from the origin as A and has been rotated 120 degrees clockwise around the origin. These examples are illustrated in Figure 7.

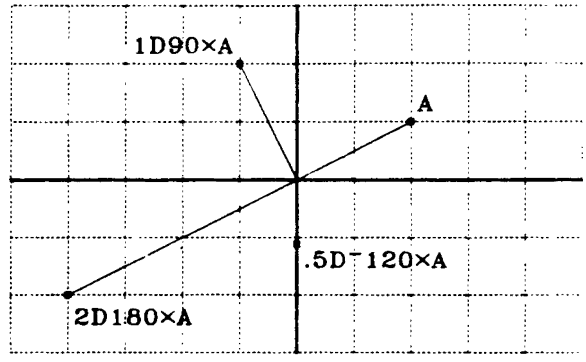

Figure 7: Geometric effects of complex multiplication.

What has been described here for multiplication of scalars carries over to other instances of APL scalar multiplication. For example, $1D90\times ARROW$ is drawn and shaded in Figure 8.

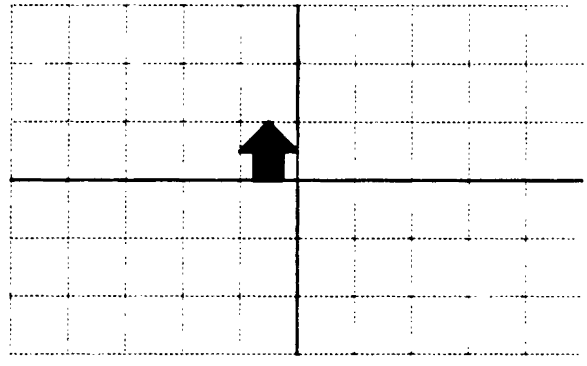

Figure 8: Rotation of points by complex multiplication.

Similarly, nested collections of objects can be created and transformed using scalar multiplication. For example, the result of

```
1 1D90 1D180 1D270 × ⊂ARROW
```

is drawn and shaded in Figure 9. (Recall that the enclose function makes *ARROW* a scalar, which is then paired with each of the 4 numbers on the left.)

All points that define *TRI* have the same magnitude, as do all points that result from

```
1 1D30 1D60 1D90 × ⊂TRI
```

The collection of objects resulting from this expression are drawn in Figure 10.

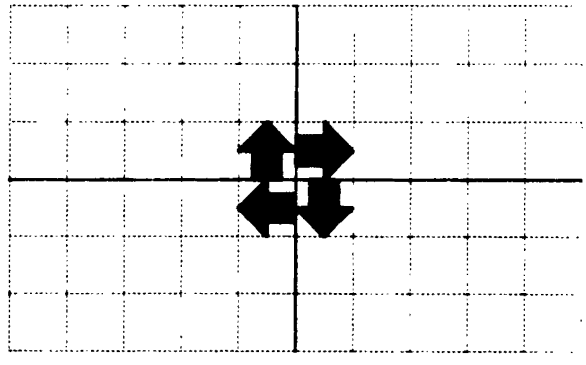

Figure 9: Replication and rotation of arrow points.

Complex addition

You have seen that addition on complex numbers requires only adding the real and imaginary parts. In graphics terms, complex addition gives translation of points in the complex plane.

For example, if A is $2J1$, then $A+A$ is $4J2$, $A-5$ is $^{-}3J1$, $-A$ is $^{-}2J^{-}1$, $A-1J2$ is $1J^{-}1$, and $A+0J^{-}3$ is $2J^{-}2$. These examples are shown in Figure 11.

Scalar addition of simple arrays representing polygonal objects corresponds to translation of these objects in the plane. For example, the *ARROW* array can be translated by the expression

```
ARROW+¯1J¯1
```

This expression is drawn and shaded in Figure 12.

In a similar way, objects can be translated and replicated by addition of nested arrays. For example, the expression

```
0 ¯1 ¯1J¯1 0J¯1 + ⊂ ARROW
```

is drawn and shaded in Figure 13.

The scalar function *CJ* is useful for creating and manipulating complex numbers in APL2. *CJ* has the following definition and use:

```
[0]  Z←R CJ I
[1]  Z←R+0J1×I

      3 CJ 5
3J5

      5 CJ 1 2 3
5J1 5J2 5J3

      2 4 6 ∘.CJ 1 3 5 7
2J1 2J3 2J5 2J7
4J1 4J3 4J5 4J7
6J1 6J3 6J5 6J7
```

The result of *CJ* is an array of complex numbers, the real parts of which are composed of the left argument and the imaginary parts of which are composed of the right argument, following the rules for scalar functions.

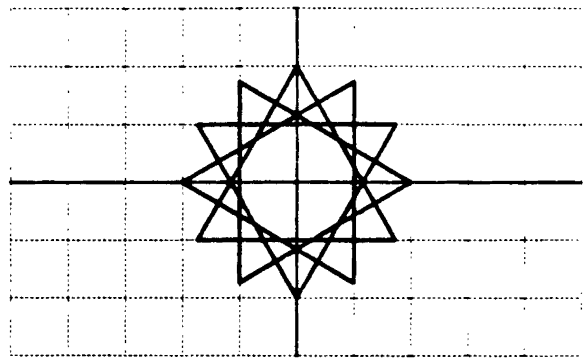

Figure 10: Replication and rotation of triangle points.

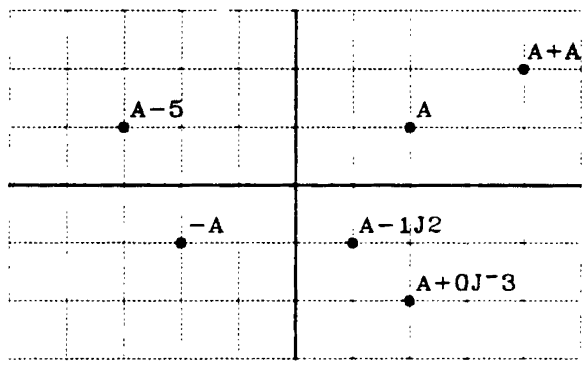

Figure 11: Geometric effects of complex addition.

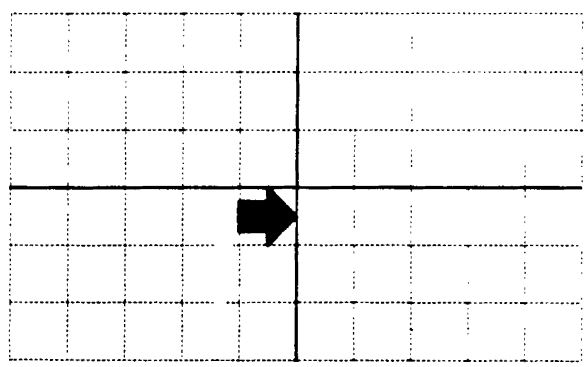

Figure 12: Translation of points by complex addition.

The function *CJ* can be used in the context of complex addition to fill the complex plane with replications of the *ARROW* object. The expression

```
((¯1+⍳10)∘.CJ ¯1+⍳6) + ⊂ ARROW - 5J3
```

is drawn and shaded in Figure 14. We subtract 5J3 to move to the bottom left corner. We then add to the scalar arrow at that spot, complex integers corresponding to the other grid points on our complex plane. (The grid lines are elided in this picture.)

Complex multiplication and addition can be combined to simultaneously translate, rotate, and replicate objects in the plane. For example, the following sequence involves all three operations:

```
A ← 1 1D30 1D60
B ← 1 1D90 1D180 1D270
B × ⊂A × ⊂ ARROW + 1.66J¯.5
```

The result of the final expression is drawn and shaded in Figure 15.

ARROW is first translated, replicated, and rotated in the first quadrant, and then this intermediate result is replicated and rotated into the other three quadrants.

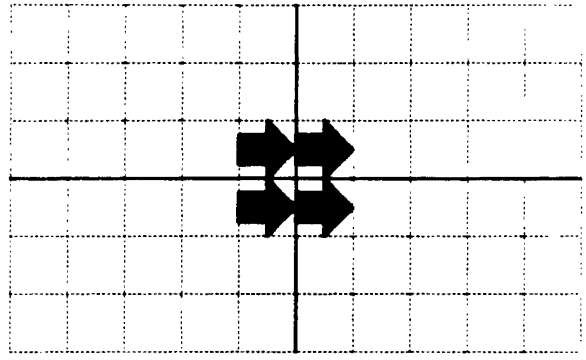

Figure 13: Replication and translation of arrow points.

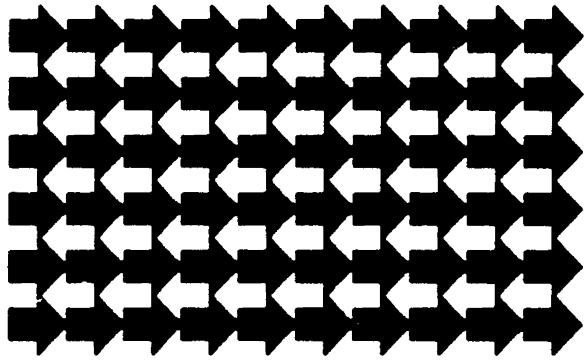

Figure 14: Multiple replication of arrow points.

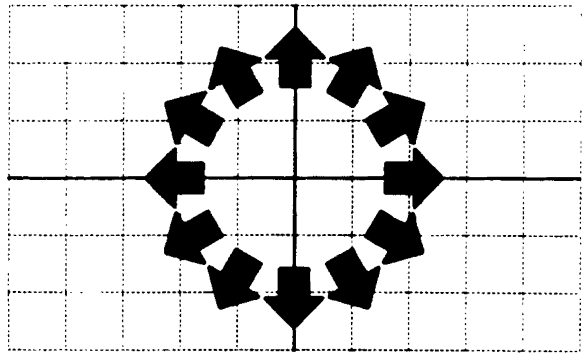

Figure 15: Translation, rotation, and translation of arrow points.

Examples from elementary fractal geometry

Fractal geometry is a relatively new mathematical discipline concerned with characterizing the irregularity and fragmentation we encounter when attempting to give a geometrical description to natural objects. In the Introduction to *The Fractal Geometry of Nature* [3], Mandelbrot gives the following motivation for inventing a "geometry of nature":

> Why is geometry often described as "cold" and "dry?" One reason lies in its inability to describe the shape of a cloud, a mountain, a coastline, or a tree. Clouds are not spheres, mountains are not cones, coastlines are not circles, and bark is not smooth, nor does lightning travel in a straight line.
>
> Many patterns of Nature are so irregular and fragmented that, compared with ... standard geometry, Nature exhibits not simply a higher degree but an altogether different level of complexity.
>
> The existence of these patterns challenges us to study those forms that (standard geometry) leaves aside as being "formless," to investigate the morphology of the "amorphous" ...
>
> Responding to this challenge, I conceived and developed a new geometry of nature and implemented its use in a number of diverse fields. It describes many of the irregular and fragmented patterns around us, and leads to full-fledged theories, by identifying a family of shapes I call fractals.

We will not pursue here the theory and application of fractals and fractal geometry; a growing body of literature addresses those topics. Rather, we want to show some simple fractal constructions in order to demonstrate the recursive computational facilities of APL2.

Initiators and Generators

Fractal constructions are carried out in successive stages, with refinements being applied at each stage of the process. One begins with two shapes, an *initiator* and a *generator*. At each stage, the generator shape replaces each instance of the initiator shape; the resulting object is then operated on in a similar manner at the next stage of construction.

A simple example of this technique is the construction of a triadic Koch island; see [3], Chapter 6. This construction begins with an equilateral triangle with unit length sides as the initiator. The generator is given by the following shape:

Each line segment of this generator is length 1/3. A single application of the generator to the initiator gives a star hexagon, or Star of David. Subsequent applications give the sequence of objects in Figure 16.

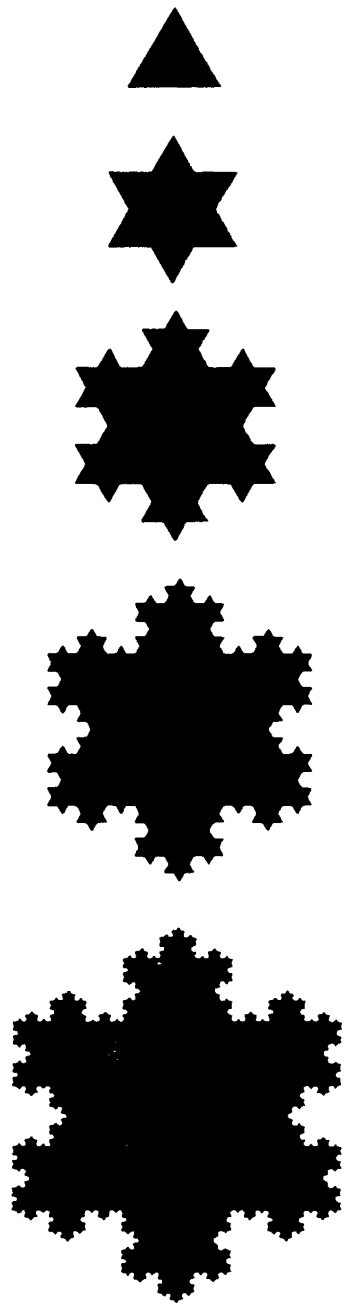

Figure 16: Triadic Koch island sequence.

An APL2 Implementation

The objects in Figure 16 can each be represented as a vector of points in the complex plane. The objects were all computed using the APL expression

```
Z ← N ( G1 FRAC ) TRI
```

TRI is a complex vector of length 4 that give the points of an equilateral triangle. *G1* is a function with the following definition.

```
[0]   Z←A G1 B
[1]   ⍝
[2]   ⍝        S
[3]   ⍝       / \
[4]   ⍝ A--R    T--B   <==    A----------B
[5]   ⍝
[6]    Z←(B-A)÷3
[7]    Z←(A+0,Z,(Z×(3*.5)×1D3C),(Z×2)),B
```

The arguments *A* and *B* of *G1* are complex simple scalars representing two points in the complex plane. The result *Z* of *G1* is a complex simple vector of length 5 representing the points of the generator shape. For example,

```
      1 G1 4
1 2 2.5J0.866 3 4
```

The operator *FRAC* applies the generator function (*G1*) to the initiator array (*TRI*); the number of operation stages is given by the left array argument *N*. The *FRAC* operator has the following definition.

```
[0]   Z←N(GEN FRAC)INIT
[1]   Z←INIT
[2]   →(N=0)/0
[3]   Z←2 GEN/Z
[4]   Z←∊(-(⍴Z)≠1⍴⍴Z)↓¨Z
[5]   Z←(N-1)(GEN FRAC)Z
```

At each stage of (possibly recursive) invocations of *FRAC*, line [1] assigns the current initiator to the result array *Z*. Line [2] causes an exit if no more stages of the construction are to be performed. Line [3] performs a pairwise *GEN*-reduction of elements of the current initiator. For example:

```
      2 G1/1 4 7
 1 2 2.5J0.866 3 4  4 5 5.5J0.866 6 7
      ⍴ 2 G1/1 4 7
2
      ⍴¨ 2 G1/1 4 7
 5  5
```

Line [4] drops the last element of all but the last subarray created in the reduction step, and then ENLISTs the result. This step essentially excludes line segments of length zero from the new graphical object. Finally, line [5] recursively invokes *FRAC* to perform the next step of the construction.

The *FRAC* operator is a paradigm for a family of related fractal constructions; we can use other initiator arrays and generator functions to obtain different graphical sequences. For example, the expression

```
N (G2 FRAC) SQR
```

computes the sequence of quadric Koch island in Figure 17; see [3], Chapter 6. *SQR* is a complex vector of length 5 representing points of the unit square. The function *G2* produces the following generator shape for a unit line segment:

G2 has the following definition:

```
[0]   Z←X G2 Y;ADG;S;BCEF
[1]
[2]       B--C
[3]       |  |
[4]   X--A  D  G--Y   <==  X----------Y
[5]          |  |
[6]          E--F
[7]
[8]   ADG←X+1 2 3×S←(Y-X)÷4
[9]   BCEF←ADG[1 2 2 3]+1 1 ¯1 ¯1×S×1D90
[10]  Z←X,(ADG,BCEF)[1 4 5 2 6 7 3],Y
```

Conclusion

We have introduced here the use of complex numbers in APL2 for simplifying and understanding some techniques and operations in two-dimensional computer graphics. In particular, we have demonstrated the use of complex simple scalars for representing points in the plane, complex simple vectors for representing polygonal objects, and nested arrays for representing collections of polygonal objects. We have shown how graphical objects represented by complex arrays can be scaled, translated, and rotated using scalar arithmetic in APL2. These techniques may be particularly useful in APL2 graphics applications that are primarily concerned with representing and manipulating two-dimensional graphics objects.

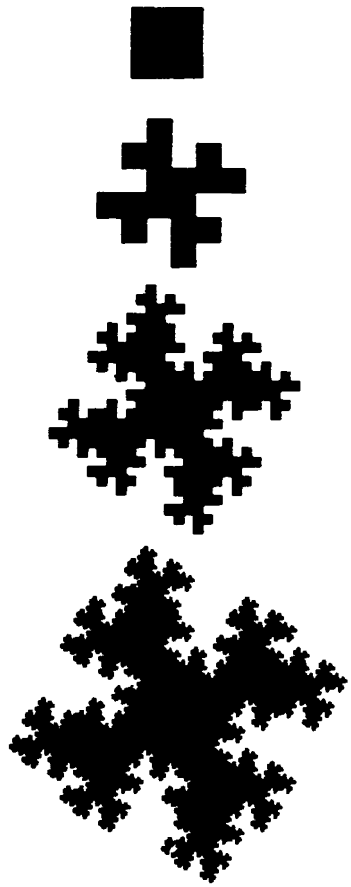

Figure 17: Quadric Koch island sequence

Acknowledgment

The authors wish to thank Edward Eusebi, Alan Graham, and Ray Trimble for their helpful suggestions and comments on various topics of this report. Special thanks also to Kacy Keene for her help on integrating text and graphic images.

References

[1] *APL2 Programming: Language Reference*, IBM Corporation, Form number SH20-9227 (1984)

[2] *Graphical Data Display Manager Base Programming Reference*, IBM Corporation, Form number SC33-0101 (1984)

[3] Mandelbrot, B.B., *The Fractal Geometry of Nature*, W.H. Freeman and Company (1982)