```
// Kal.bcpl -- Alto Kaleidoscope by (in order of appearance):
//             Martin Newell
//             Joe Maleson
//             Dana Scott
//             David Boggs
// Copyright Xerox Corporation 1979
// Last modified June 11, 1979  1:41 PM by Boggs

get "AltoDefs.d"

// Kal uses three-parameter (a, b, c) sequence generators to
//   create and erase points with 8 or 12 way symmetry.  Two
//   other parameters, (period and persistence) are global
//   to all sequence generators.  On each cycle, each generator
//   does a ← G(a, b) and the 'a's are used to create 8 or 12 new points
//   and erase 8 or 12 old ones.  Every 'period' cycles each generator
//   does b ← G(b, c).  Generators come in pairs - one to create
//   points and the other to erase them.  The erasing generator
//   runs 'persistence' cycles behind the creating one.  Thus
//   'persistence' determines the life of each point, and hence
//   the number of points on the screen at any one time.
//
// Specification of parameters:
//   Type any char to stop the display - char is discarded.
//   Each parameter may be typed, terminated with non-digit,
//   or it may be defaulted to its existing value by typing <space>
//   or it may be reset to its initial value by typing <cr>.

external
[
// incoming procedures from KalA.br & KalMc.mu (microcode)
InitPoint; ShowPoint; ErasePoint

// incoming procedures from the OS and packages
Gets; Puts; Endofs; PutTemplate; Ws
Zero; MoveBlock; LoadRam; InitBcplRuntime; CallSwat

// incoming statics from the OS
keys; dsp; lvCursorLink; kalRamImage
]

manifest
[
screenWidthWords = 32  //known to microcode
screenHeightDots = 512

stackLimit = 335b
]

structure GP:               // Generator Parameters
[
Xa word
Xb word
Xc word
Ya word
Yb word
Yc word
period word
cnt word     // initialized to GP.period and decremented to 0
symmetry word
]
manifest lenGP = size GP/16
```

```
//-----------------------------------------------------------------
let Kal() be
//-----------------------------------------------------------------
[
if LoadRam(kalRamImage, false) ne 0 then CallSwat("LoadRam failed")
InitBcplRuntime()

@lvCursorLink = false    // Inhibit update from mouse
@cursorX = -1                        // Move cursor off the screen

let screenAreaWords = screenWidthWords*screenHeightDots
let screenBitMap = (@stackLimit+1) & -2
@stackLimit = @stackLimit + screenAreaWords + 1
Zero(screenBitMap, screenAreaWords)
InitPoint(screenBitMap)

let screenDCB = vec IDCB+1; screenDCB = (screenDCB+1) & -2
Zero(screenDCB, IDCB)
screenDCB>>DCB.width = screenWidthWords
screenDCB>>DCB.indentation = 3
screenDCB>>DCB.background = 1
screenDCB>>DCB.bitmap = screenBitMap
screenDCB>>DCB.height = screenHeightDots rshift 1

let marginDCB = vec IDCB+1; marginDCB = (marginDCB+1) & -2
Zero(marginDCB, IDCB)
marginDCB>>DCB.next = screenDCB
marginDCB>>DCB.height = (808 screenHeightDots)/4
marginDCB>>DCB.background = 1
let dspDCB = @displayListHead
@displayListHead = marginDCB

// initial, built-in parameters
let persistance = 10900
let init = table [ 3; 5; 10; 5; 3; 10; 496; 0; 8 ]

  [  //main program loop
  init>>GP.cnt = init>>GP.period
  let show = vec lenGP; MoveBlock(show, init, lenGP)
  let erase = vec lenGP; MoveBlock(erase, init, lenGP)

  for i = 1 to persistance do Generate(show, ShowPoint)
  while Endofs(keys) do
    [
    Generate(show, ShowPoint)
    Generate(erase, ErasePoint)
    ]
  Gets(keys)  //flush interrupt character

  // get new params from user
  screenDCB>>DCB.next = dspDCB
  init>>GP.Xa = GetNumber("Xa", init>>GP.Xa, show>>GP.Xa)
  init>>GP.Xb = GetNumber("Xb", init>>GP.Xb, show>>GP.Xb)
  init>>GP.Xc = GetNumber("Xc", init>>GP.Xc, show>>GP.Xc)
  init>>GP.Ya = GetNumber("Ya", init>>GP.Ya, show>>GP.Ya)
  init>>GP.Yb = GetNumber("Yb", init>>GP.Yb, show>>GP.Yb)
  init>>GP.Yc = GetNumber("Yc", init>>GP.Yc, show>>GP.Yc)
  init>>GP.period = GetNumber("Period", init>>GP.period)
  persistance = GetNumber("Persistance", persistance)
  let symmetry = init>>GP.symmetry ne 12? 8, 12
  init>>GP.symmetry = GetNumber("Symmetry (8 or 12)",
   (init>>GP.symmetry ne 12? 8, 12))
  Zero(screenBitMap, screenAreaWords)
  screenDCB>>DCB.next = 0
  ] repeat
]
```

```
//-------------------------------------------------------------------
and Generate(gp, Proc) be
//-------------------------------------------------------------------
[
gp>>GP.Xa = (gp>>GP.Xa + gp>>GP.Xb) xor gp>>GP.Xb
gp>>GP.Ya = (gp>>GP.Ya + gp>>GP.Yb) xor gp>>GP.Yb
gp>>GP.cnt = gp>>GP.cnt -1; if gp>>GP.cnt eq 0 do
   [
   gp>>GP.Xb = (gp>>GP.Xb + gp>>GP.Xc) xor gp>>GP.Xc
   gp>>GP.Yb = (gp>>GP.Yb + gp>>GP.Yc) xor gp>>GP.Yc
   gp>>GP.cnt = gp>>GP.period
   ]

test gp>>GP.symmetry ne 12
   ifso
      [
      let x0, y0 = gp>>GP.Xa rshift 8, gp>>GP.Ya rshift 8
      unless x0 gr y0 do
         [
         let x1, y1 = 511-x0, 511-y0
         Proc(x0, y0); Proc(x0, y1)
         Proc(y0, x0); Proc(y0, x1)
         Proc(x1, y0); Proc(x1, y1)
         Proc(y1, x1); Proc(y1, x0)
         ]
      ]
   ifnot
      [
      let x0, y0 = gp>>GP.Xa rshift 9, gp>>GP.Ya rshift 9
      unless x0 gr y0 do
         [
         let x1, y1 = x0 lshift 1, y0 lshift 1
         for yi = 256 to 257 do  //double each scan line to avoid flicker
            [
            Proc(256+x0+y0, yi-x1+y1); Proc(256+x0+y0, yi+x1-y1)
            Proc(256-x0-y0, yi-x1+y1); Proc(256-x0-y0, yi+x1-y1)
            Proc(256+x0-y1, yi-x1); Proc(256+x0-y1, yi+x1)
            Proc(256-x0+y1, yi-x1); Proc(256-x0+y1, yi+x1)
            Proc(256+x1-y0, yi-y1); Proc(256+x1-y0, yi+y1)
            Proc(256-x1+y0, yi-y1); Proc(256-x1+y0, yi+y1)
            ]
         ]
      ]
]
```

```
//-------------------------------------------------------------------
and GetNumber(name, initial, current; numargs na) = valof
//-------------------------------------------------------------------
[
test na eq 3
   ifso
      [
      Ws("*N*N<CR> => initial value, <Space> => current value, or type a number.")
      PutTemplate(dsp, "*N$S (initial $UD, current $UD): ",
       name, initial, current)
      ]
   ifnot
      [
      current = initial
      Ws("*N*N<CR> or <Space> => current value, or type a number:")
      PutTemplate(dsp, "*N$S (current $UD): ", name, current)
      ]
let first, number = true, 0
   [
   let char = Gets(keys)
   switchon char into
      [
      case $*N: test first
         ifso [ PutTemplate(dsp, "$UD", initial); resultis initial ]
         ifnot resultis number
      case $*S: test first
         ifso [ PutTemplate(dsp, "$UD", current); resultis current ]
         ifnot resultis number
      case $0 to $9:
         [
         first = false
         number = number*10 + (char-$0)
         Puts(dsp, char)
         endcase
         ]
      ]
   ] repeat
]
```