HHC 2004 Conference User RPL Programming Contest

A Post-conference Analysis

Jeremy Smith

Goal: Write a program that graphs a circle and fills it with random dots.

Programming contests are a blast, and calculator programming contests are especially interesting because the platform is discrete which makes the contest more self contained and therefore tighter.

The contest tests programming skill, knowledge of the machine, math and science, and working under duress. Also, the judge is tested to produce a simple but adequate challenge, a rapid and robust scoring process, an unambiguous set of rules, and a satisfied set of contestants. A classic optimization problem.

Most attendees get the contest instructions with their proceedings on Saturday morning, before their coffee, and hand in their efforts on Sunday afternoon. Talks go on all day both days, folks catch up during breaks, meals and Saturday evening are pretty social, all of which means that there is actually no time to even think about the contest, unless you do it during the boring talks. This year there were very few boring talks, which made it an excellent personal multitasking challenge, programming under battle conditions. At some previous conferences, contests were distributed on the Friday evening – much better. How about getting it emailed as an incentive for early registration?

On Sunday afternoon, the judge is deluged with results and has to fairly evaluate them and declare a winner, an intense ten minutes of which few relish a repeat experience.

Since many folks have planes to catch right after the closing remarks, there's little chance for a post-mortem which is a great shame because lots of effort goes into the contest and there's tons of good stuff to pick through.

This is the inspiration for this article, and I'm sharing it at large for your enjoyment. I scrutinize this years programs (and the contest and judging) and report many cracks, inconsistencies, and subtle errors. This is by no means a complaint or murmurings of dissatisfaction. The inevitable laxity allows for great creative freedom, from which we all (contestants, judges, and contest creators) benefit, learn (improve) and discover.

The contest and rules are in appendix A. Basically, write a program that graphs a circle and fills it with random dots. The winning program was the smallest (fewest bytes) that drew the dots whilst complying with the rules.

There were seven entries by four people. Incredibly, all seven methods had very different approaches. Cyrille was particularly creative and prolific, with his methods partially eclipsing two of the other entries. The programs are each given below as a code listing followed by an overview and then commentary and analysis. They are ordered largest to smallest (winning). Following the programs are some general comments, and then some appendices.

Programs

```
%%HP: T(3)A(D)F(.);     @ JS – 246 bytes – Jeremy |-) Smith
\<< { # 0d # 0d } PVIEW { # 64d # 32d } DUP # 31d 0 360 ARC 0 2345
  START RAND RAND { } + + .5 DUP { } + + - 64 DUP { } + + * DUP
    IF SQ OBJ\-> DROP + \v/ 31 <
    THEN OVER B\->R ADD R\->B PIXON
    ELSE DROP
    END
  NEXT DROP { } PVIEW
\>>
```

This program generates a random x, y point which it plots if $\sqrt{(x^2 + y^2)}$ is less than the radius – in other words, if the point falls inside the circle.

The code and method are very generic, and it's surprising that more entries didn't use this method. The CdB.HORIZ below was the only other based on the same algorithm. The circle was centered in the display, a byte-consuming non-requirement.

```
%%HP: T(3)A(D)F(.);     @ CdB.ROM – 185 bytes – Cyrille de Brebisson
\<< "\"FFFFF\"0 3{#0#0}7" OBJ\-> FREEZE PVIEW
  FOR R ".1(0 0)R 0 7" OBJ\-> ARC
  STEP DUP + DUP + RAND 100000. * R\->B # 2176d PEEK + H\->S PICT RCL \->H H\->S AND S\->H
H\-> PICT STO "(0 0)3 0 7" OBJ\-> ARC
\>>
```

CdB.ROM draws a filled circle in PICT, which is then converted to a string. An equivalent amount of code from a random location in ROM is AND'd with the string, and converted back into PICT, effectively randomizing the circle.

The result usually looks random, but ROM is anything but random and sometimes the result is quite (interestingly) patterned. But it looks random often enough. There is no 'fill' graphics command, and so this is accomplished by drawing concentric circles. However, the inevitable moiré pattern is apparent, diminishing a pure random distribution even if ROM was random, since a particular pattern of pixels will always be off.

PICT is 1098 bytes which converts to a hex string (\->H) and then to a string (H\->S) 1098 characters long. 2176 bytes of ROM, from a random location, are PEEK'd and appended to a string of 20 F's, for a total of 2196, which makes a 1098 character string. This is AND'd with the PICT string; the 20 F's are presumably sufficient to preserve the PICT preamble. The resultant string converts back to a hex string (S\->H), then an object (H\->), then put back into PICT.

Because the whole of PICT is AND'd, the required circle is drawn afterwards as it otherwise would have been randomized as well. However, this inevitability violates the rules.

```
%%HP: T(3)A(D)F(.);     @ CdB.HORIZ – 155 bytes – Cyrille de Brebisson
\<< "7(0 0)3 0 7 -3 3{#0#0}" OBJ\-> PVIEW
  FOR Y -3. 3.
    FOR X
      IF X SQ Y SQ + 9. < RAND 1. 2. / < AND
      THEN X Y R\->C PIXON
```

```
     END .1
    STEP .1
  STEP ARC FREEZE
\>>
```

CdB.HORIZ steps through every pixel inside a circle, row by row, with half a chance that any pixel will be turned on. The pixels are kept inside the circle by checking for each point (x, y) that $x^2 + y^2$ is less than the radius^2.

The circle is drawn last (thus disqualifying the program) but the ARC command could trivially have been moved to the beginning.

```
%%HP: T(3)A(D)F(.);     @ BVB – 145.5 bytes – Bill Butler
\<< ERASE 30 R\->B DUPDUP 2. \->LIST DUP 0. * PVIEW DUP ROT 9. 6. EXPM ARC 1. 29
  FOR i 9. 6. EXPM
    FOR j RAND 2. * IP { DUP i R\->B j DUP ARC } IFT 1. R\->D i /
    STEP
  NEXT 7. FREEZE DROP
\>>
```

Bill's program draws a series of concentric circles with radii from 1 to 30 pixels, but each circle is drawn a pixel at a time with half a chance that any pixel will be skipped.

The pixels are drawn in an arc from 9° to 402.4° (that's the 9. 6. EXPM, which presumably is a 2 byte shortcut for 0 360) but the loop steps by 57.3°/radius (1. R\->D i /) which is exactly one pixel length (measured along the circumference) for a circle of any radius.

This program was developed on a machine in degrees mode. In radians mode the pixels would not be adjacent but plotted relatively arbitrarily along each concentric circumference. But there were still sufficient to fill about half the circle, and they certainly appear random.

The end result could not be considered random for a number of reasons. There is an overlap segment of the circle from 369° to 402.4° and it appears as a slightly darker wedge, but easily missed. Since the circles were drawn with radii from 1 to 29, it meant that the middle pixel was always off. On a pixel screen, drawing concentric circles as a means of filling a circle doesn't work as a moiré pattern forms and some pixels are never accessed (watch CdB.ROM to see this).

```
%%HP: T(3)A(D)F(.);     @ RCH.1 – 102.5 bytes – Roger Hill
\<< ERASE "{#0#0}PVIEW(0,0)3 0 360ARC 0 1400" OBJ\->
  FOR n RAND \v/ 3. * 0. RAND \pi \->NUM * R\->C EXP SQ * PIXON
  NEXT 0. FREEZE
\>>
```

Roger's program uses a mathematical function which generates an x, y point inside a circle, and this is randomized and plotted.

RCH.1 uses the 'well-known identity' $e^{i\theta} = \cos\theta + i*\sin\theta$ (news to me). As Roger told me – an instructor once said, "this is well-known to those who know it well". (I've just learned that it's one of de Moivre's, and substituting π gives the celebrated $e^{\pi i} = -1$.)

The main program produces 3√(rand) and $e^{(0,\ \pi*\text{rand})}$ squared, and multiplies them. $e^{(0,\ \pi*\text{rand})}$ is evaluated as the complex number $e^{\pi*i*\text{rand}}$ (the real part is zero) which, from the identity, is cos(π*rand), i*sin(π*rand). Plotting this plots random points on a semicircle (π = 180°).

$e^{\pi i}$ squared is $e^{2\pi i}$, which plots points on a whole circle (squaring in this way saves 2.5 bytes over doubling π before taking the exponent).

Multiplying by 3√rand randomly spreads the random points on the circle from the circumference (now at 3) back to the center. Multiplying by just 3*rand (evenly from 0 to 3) would cause pixels to cluster nearer the center, because the same number of points nearer the center have less space than nearer the circumference. 3√rand (still from 0 to 3) decreases the density at points nearer the center proportionally to the area, and so they're evenly distributed.

```
%%HP: T(3)A(D)F(.);    @ RCH – 97.5 bytes – Roger Hill
\<< ERASE "{#0#0}PVIEW(0,0)3\pi 364ARC 0\pi" OBJ\-> ALOG
  FOR n RAND \v/ 3. * 0. RAND \pi * R\->C EXP SQ * PIXON
  NEXT 0. FREEZE
\>>
```

This was Roger's original program which, upon handing in, was discovered to require having flag –3, numeric results, set, which disqualified it. A couple of hasty mods gave the acceptable RCH.1 (above).

```
%%HP: T(3)A(D)F(.);    @ CdB.TLINe – 91.5 bytes – Cyrille de Brebisson
\<< "7(0 0)3 0 7 0 200{#0#0}" OBJ\-> PVIEW
  START "`e^(RAND*i*7)*3`" DUP OBJ\-> \->NUM SWAP OBJ\-> \->NUM TLINE
  NEXT ARC FREEZE
\>>
```

CdB.TLINe uses essentially the same mathematical function (as in RCH.1 above), twice, to generate a pair of random points on a circle, and then toggles all the pixels on a line between them. Initially, you just see random lines appearing, but as the density gets up, more lines are crossing old lines, and more pixels are being randomly toggled, achieving half-randomness ten times quicker than plotting point by point.

The main algorithm is contained in the string "`e^(RAND*i*7)*3`". When the expression is liberated by OBJ\-> the backticks cause their contents to be immediately evaluated. By including i (√-1) in the parenthesized exponent, a real part of zero is assumed, resulting in a random point on the circumference of a circle. The exponent of rand*i*7 is essentially rand*i*2*π, and taking the exponent gives cos(rand*2*π), sin(rand*2*π). Multiplying by 3 expands the circle to radius 3.

The expression is evaluated twice, generating two random points, and the line is drawn between them.

Does toggling lines of pixels create a real random distribution? Certainly, after sufficient lines have been laid down, the distribution will tend towards half on/off, and randomness seems concomitant. Furthermore, even though random points will occur on the circumference on the overlap region twice as often (2π to 7 radians, an arc of 41°), since pixels are toggled, proximity to randomness is probably maintained.

Technically, the program should have been disqualified both since it needed flag –103, complex mode, set (default is clear), and also as the enclosing circle was drawn after the fact, details missed or overlooked by the judge. It's a trivial change to draw ARC first, adding only a couple of bytes (still smallest program). However, had it been drawn first, it would have been randomized since TLINE toggles pixels (even a circle of radius 3.1 loses the occasional pixel).

Commentary

... plot random pixels ...

On a computer, especially when using tools like RNGs, it is easy to generate random looking data, and hard to check for and eliminate artifacts of the computer. Strictly adhering to ‘plot random pixels’ would eliminate some of the programs.

Presumably, this is why random was qualified by ‘must LOOK random’. There are plenty of applications for which something that might be less than random still satisfies the requirements.

... inside that circle ...

The rules specify that the program must graph a large circle, then plot random pixels inside that circle. This is actually not possible. This is because the circle is one pixel thick, instead of infinitely thin, and about half the points inside a circle will actually hit the circle. This is noticeable for programs that toggle pixels (CdB.TLINe) because parts of the circle are (or would be) nibbled away, which is why the circle is drawn afterwards. Would the judge have accepted a nibbled circle?

Also, points can fall outside the circle. This is apparent if you plot points on a circumference, point by point, one pixel width along the circumference (2*π*radius steps – about 200) on top of an arc of the same radius.

If you plot points up to and including the circle of one pixel radius less than the circle, then some points inside the circle would never be accessed. Draw two concentric circles to see.

And finally, Bill Butler has identified 16 different circles of radius 30, depending on the angle/arguments to ARC (and they’ll contain different pixel counts, which is relevant to the discussion in appendix B).

... smallest program ...

Cyrille’s CdB.TLINe program won at 91.5 bytes. I’ve heard of two post-conference programs at 77 & 80 bytes.

... other monkey business ...

A number of programs put code inside a string, and then `OBJ\->` (or evaluated) it. This is a byte saving technique used solely because it increases the chance of winning. Many folks despise that because it wouldn’t be used outside the competition to otherwise achieve the same goals, and many new to these contests learn about this trick after the fact. In a famous past case, an entire program was stringed, and won, but otherwise was not particularly any different from the other programs. A bit like running a program out of a zip or sit file and saying it’s smaller than equivalent but uncompressed programs.

However, the technique can act as a subroutine, which can be repeatedly called by duplicating the string as needed. Cyrille’s CdB.TLINe program generates the two random points for TLINE in this way. This kind of use validates the technique, and therefore it cannot be eliminated. Stringed code can sometimes run faster than the naked code, worth checking where speed is also at stake.

Putting numbers in strings is the most byte-efficient, putting code in is less so – there is a minimum amount of code that can be compressed in this way. Cyrille's programs almost entirely just strings numbers.

... assume machine default flag settings ...

The calc has hundreds of modes, and it is a rude surprise for many to reset all their personalizations. The contest should include instructions on both how to revert to default and how to restore personalizations.

The most apparent issue in this contest was deg/rad mode (deg is the default on the HP 48, rad on the HP 49). All programs ran correctly in rad mode but most of them would not have in deg mode.

RCH was disqualified because of a flag setting, but there are other crucial settings. For instance, if PDIM was set with a different aspect ratio half the programs would have plotted random ovals instead of circles.

Conclusion

When the PPC Journal was being published, one of the most popular columns was Tips & Routines, a collection of very short but elegant solutions to common problems. The editor Richard Nelson said, though, that it was also the hardest column to get material for.

These contests are like that – they're simple but tough for the contestants, but also a challenge for the contest creator. They're often part of a larger programming project and you just know that the particular segment could be approached in many diverse ways. A contest demonstrates that very nicely by the optimization process.

The judging process would, ideally, be a list of checks that each program would need to pass, and many of the checks would be done on the computer. The software industry writes a 'software test harness' to put any piece of software through a battery of tests and simulations before it's released or deployed. But test harnesses can be big projects even for small programs. This is an area that's always contentious, as reflected in rule 1.

Appendix A – the rules

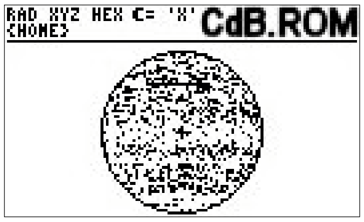
User RPL Programming Contest

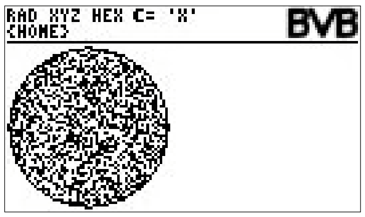
Goal: Write a program that graphs a circle and fills it with random dots.

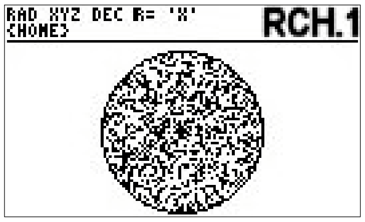
Specifics: The program must graph a large circle (at least 60 pixels high for all HP48 & HP49 models), then plot random pixels inside that circle until approximately half the pixels are dark, then exit without user intervention, leaving the graph displayed. The dots must be random, that is, (a) they must be different each time the program is run, and (b) they must LOOK random (patternless) to the judge. The entire graphing process must be visible as it runs. No dots allowed outside the circle. The distribution of the pixels inside the circle must not be dense near the center, nor near the circle, nor anywhere else, but not patterned either. A violation of any of the above disqualifies the program.

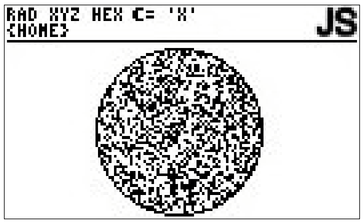
Winning: The winner will be the smallest program (in bytes).

Examples: [In the actual rules, Joe Horn had eight example pictures, six ‘bad’ – blatant patterning, two ‘good’. The following are actual outputs from CdB.ROM, which shows banding (bad), and BVB, RCH.1, and JS, all presumably good.]









Rules:

1. The decision of the judge is final.
2. The purpose of the contest is to have fun.
3. At least two contestants must participate.
4. Plain-vanilla user RPL only; no SYSEVALs or other monkey business.
5. Contest program submittal must be completed by afternoon break on Sunday September 26, 2004.
6. Include your initials in the name of your program. Transfer your program to the judge’s machine. Make sure your checksum is the same as the judge’s. [Is the onus on the judge to have all the necessary cables, etc. – ‘for all HP48 & HP49 models’ – Ed.]
7. This is a contest between *individuals, not teams*; one submittal per person, one person per submittal.
8. By submitting a program, you allow it to be shared with the community.
9. You must be present to win.
10. If a point is unclear, ***ask*** immediately. No excuses for ignorance. Clarifications will be officially announced during conference hours.
11. This contest entry instruction sheet and entry form will be available at registration Saturday Morning.
12. Assume machine default flag settings (except HP49; assume RPL mode). Altered flag settings must be returned to default status upon program completion. Stack contents before and after the program must be the same.
14. *Work alone*. Do not consult the Internet. The AUR, Owner’s Manual, or other books are permissible, of course.
15. Happy Programming!

Appendix B – randomness

... until approximately half the pixels are dark.

What a concept. I began thinking about this and, although quite peripheral to the contest, it was most interesting.

Most of the programs had circles with a radius of 30 pixels, which would be about 2922 pixels ($61^2 * \pi/4$). Assuming you’re randomly turning on these pixels, and that pixels could be turned on multiple times, how many times must you randomly turn on a pixel for half of them to be on? Surely more than half 2922, but how much more?

I found the following relationship by running billions of simulations. If you have just two pixels then, obviously, one shot does it, but if you have four pixels then you need an average of 2⅓ shots on average to turn half the pixels on, if you have six pixels – 3.7 shots, and so on.

When I ran the simulations, I found the number of shots, compared to the total number of pixels, was indeed more than a half, and this ratio got larger the more pixels there were (see table).

pixels	**series**	**ratio**	**average number of shots needed**
n=2	1 -1/2	0.5	1/2 of two pixels = 1 shot
n=4	1 -1/2 +1/3 -1/4	0.5833	0.5833 * 4 pixels = 2.33 shots
n=6	1 -1/2 +1/3 -1/4 +1/5 -1/6	0.6166	0.6166 * 6 pixels = 3.7 shots
n=8	1 -1/2 +1/3 -1/4 +1/5 -1/6 +1/7 -1/8	0.6345	0.6345 * 8 pixels = 5.076 shots
n=10	1 -1/2 +1/3 -1/4 +1/5 -1/6 +1/7 -1/8 +1/9 –1/10	0.6456	0.6456 * 10 pixels = 6.456 shots
n=2922		0.692976	2025 shots
n=3721		0.693013	2579 shots
n=∞	ln(2)	0.693147	

But the ratio converged to 0.693147 as n got huge, which turns out to be ln(2). There's a general series for ln(1+x)

$$\ln(1+x) = \int 1/(1+x)\,dx = x - x^2/2 + x^3/3 - x^4/4 + x^5/5 - x^6/6 \ldots$$

And if we take the particular case of ln(2), where x=1, we obtain the fractional series

$$\ln(2) = 1 - 1/2 + 1/3 - 1/4 + 1/5 - 1/6 + 1/7 - 1/8 + 1/9 - 1/10 \ldots$$

Summing n terms of this series fortuitously gives us the ratio for n pixels – the first few are shown in the table above. Why this is so I have not been able to figure out, but by means of this handy series we can calculate the ratio for any number of pixels. (Yes, I only looked at even n, since an odd number of pixels is hard to divide evenly. But for odd n the derived ratio would provide an accurate assessment.)

So, for the 2922 pixels in our 30 pixel radius circle we need to shoot 2922 * 0.692976 = 2025 random pixels to cover about half of them.

Roger's program shot 1386 in RCH, and 1400 in RCH.1. Way low (not a famous movie actress).

But if we're shooting a square and just saving those inside the circle, then we need to shoot 3721 (61^2) * 0.693 = 2579. (In my program I randomly guessed 2345. Close enough for government ballparks.)

The series converges rapidly enough, however, that for all practical purposes you can just use ln(2) as the ratio. And indeed 2922 * ln(2) = 2025, and 3721 * ln(2) = 2579, the same result as above. Which means that you could just have the loop count up to ln(2) * πr^2.

What if you want to make ⅓ or 5/8 of the pixels dark? A few more billion simulations showed that using a ratio of ln(1/(1-x)), where x is ⅓ or 5/8, will do the trick nicely.

Just a note to mention that I started running the simulations on the calculator, but it really is too slow, and so I switched to a desktop computer. But I shouldn't have had to. Many handheld computers and PDAs would have had sufficient horsepower.