Is Computer Programming a Form of Art?
Martin Ruckert

Abstract
Since the publication of “The Art of Computer programming” by Donald E. Knuth in 1968, the notion that programs can be considered works of art is familiar to computer scientist, but the general public has taken little notice of such works of art. For example, there are no art reviews where computer programs are presented and evaluated based on their artistic value. This paper, written for the occasion of Donald Knuth’s 80th birthday, attempts to fill this gap. It presents and evaluates three small programs. The selection reflects the personal preferences and taste of the author.

1 Introduction
Fifty years ago, in 1968, the first volume of ‘The Art of Computer programming’ [5] was published. Already in the first sentence of the preface, Donald E. Knuth states that “the process of preparing programs for a digital computer is especially attractive ... because it can be an aesthetic experience much like composing poetry or music.” And in his 1974 ACM Turing Award Lecture [6], after reviewing the various possible meaning of the word ‘art’, he reiterates: “When I speak about computer programming as an art, I am thinking primarily of it as an art form, in an aesthetic sense.” And he concludes: “We have seen that computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty.”

I do not remember, however, a recent auction for example at Christie’s, where a collector was buying a computer program, maybe even setting a new price record; and I do not recall having seen a review of a computer program for example in the Arts&Leisure section of the New York Times. One might object that it needs a professionally trained programmer to appreciate the beauty of a program, but then in many cases only the professionally trained painter is able to expect that there are far more professional programmers among the readers of the New York Times than professional painters1. As a matter of fact, the recognition of computer programming as an art form remains mostly within the practitioners of that art and there is little or no recognition from the wider fine arts community.

So in preparation for this lecture, I decided to select a few sample programs (of one of the best programmers), research the established criteria of the fine arts community, and write an art review that—at least in principle—could appear in the Arts&Leisure section of the New York Times. This turned out to be a quite difficult project, but never the less, I will present the results in what follows.

2 Composition of Time and Place
“Time & place must be taken into consideration in the discussion of any human affair” writes Eric Gill in ‘An Essay on Typography’[3]. Eric Gill, famous for the typefaces he designed, was not only an artist and a printer but also a fighter for social justice and a philosopher. In his essay, he analyzes the transition of typography and the printing bussines in England during the 1930’s from the “humanity of craftsmanship” to the “power of industrialism”, and I think, we have seen a very similar transition in computer programming over the last fifty years since the publication of ‘The Art of Computer Programming’. So it is instructive to review his thoughts—concerned with an established fine art: Typography—and see how they apply to computer programming.

Eric Gill writes: “Now the chief and, though we betray our personal predilection by saying so, the most monstrous characteristic of our time is that the methods of manufacture which we employ and of which we are proud are such as make it impossible for the ordinary workman to be an artist, that is to say a responsible workman, a man responsible not merely for doing what he is told but responsible also for the intellectual quality of what his deeds effect.”

1 Students enrolled in Germany during the Fall semester of 2015: computer science 195,279; fine arts 5,251.[1]
He exemplifies the new way of production by the use of “ornament & the ornamental” and insists that in industrial production “everything in the nature of ornament must be omitted ... and this is not because we hate ornament & the ornamental, but because we can no longer procure such things; we have not got a system of manufacture which naturally produces them, and, most important of all, if we insist on the ornament we are not making the best of our system of manufacture, we are not getting the things which that system makes best.” [3, pp. 12–13] He characterizes ornament as a result of “exuberance” (see Figure 1); arguing that exuberance is reserved for the human artist, for whom it is “natural and proper”, where as industrial production must “be entirely free from exuberance and fancy” because “you cannot be exuberant by proxy”.

Is there such a thing as exuberance in computer programming? Here is my first example: the \texttt{input\_ln} function of \TeX{}[8] shown in Figure 2. When I teach my students computer programming—or to be more precise: software engineering—I tell them that they must not write a comment unless it contributes some significant information improving the readability of the program. And I hope by now, you have spotted the line that caused me to pick the program shown in Figure 2: The comment that contributes nothing whatsoever to the understanding of the program and according to the rules of software engineering shouldn’t be there.

Reading it, however, I start smiling. Given the assignment of \texttt{first} to \texttt{last} there is no further need to grab the bible and look up the reference; seeing it immediately activates the mirror neurons in my brain enabling empathy and affective resonance with the author of that line, who suddenly is present as a complete human being. Modern neuro-science suggests[2] that we are able to understand and share the emotions of others by processing expressions of emotions with our very own emotion system.
3. The following eight-liner is conjectured to be the shortest MIX sorting routine, although it is not recommended for speed. We assume that the numbers appear in locations 1, . . . , N (that is, INPUT EQU 0); otherwise another line of code is necessary.

```
2H LDA 0,1 B
CMPA 1,1 B
JLE 1F B
MOVE 1,1 A
STA 0,1 A
START ENT1 N A+1
1H DEC1 1 B+1
J1P 2B B+1
```

Note: To estimate the running time of this program, note that $A$ is the number of inversions. The quantity $B$ is a reasonably simple function of the inversion table, and (assuming distinct inputs in random order) it has the generating function

$$z^{N-1}(1 + z)(1 + z^2 + z^{2+1}) \times (1 + z^3 + z^{3+2} + z^{3+2+1}) \ldots
(1 + z^{N-1} + z^{2N-3} + \cdots + z^{N(N-1)/2})/N!.$$ 

The mean value of $B$ is $N - 1 + \sum_{k=1}^{N} (k - 1)(2k - 1)/6 = (N - 1)(4N^2 + N + 36)/36$; hence the average running time of this program is roughly $7/9 N^3 u$.

Figure 5: Donald E. Knuth: *The shortest MIX sorting routine.*

reveals that its author not only knows his bible, he also knows how to typeset bible references. Taken all these details together made me choose this innocent line as my first example. It certainly exemplifies the exuberance—reserved for the true artist—in computer programming.

3 Extreme Programming

In the Historic Green Vault in Dresden, Germany, you can admire a tiny cherry stone (see Figure 3), presented in 1589 as a gift to Kurfürst Christian I of Saxonia, in which the artist carved 185 faces\(^2\) from clergy and nobility. The carving is set in a golden pendant and adorned with a pearl. These

\(^2\)Newer counts confirm 113 faces.
micro-carvings were quite popular in the 16th century providing an opportunity for the artist to demonstrate his skills under even the most restrictive conditions.

I consider the program shown in Figure 5 as a similar demonstration of skill and beauty. The solution to exercise 3 in section 5.2.1, Sorting by Insertion, of ‘The Art of Computer Programming’ is set in a short analysis of its running time and adorned with a “reasonably simple” generating function.

Looking for the best examples of fine art from 16th century Germany, however, the carved cherry stone would certainly not make it to the top 500 list. Similarly, the sorting program might make it into the Guinness book of world records but not into the top 500 list of fine art in computer programming. For a serious artists, achieving impressive results under severe restrictions may be more of a popular pastime than part of the true core of his art. So what then is at the core of art?

4 Fine Art

In his 1974 ACM Turing Award Lecture, cited already earlier, Donald E. Knuth writes: “The chief goal of my work as educator and author is to help people learn how to write beautiful programs.” Looking at established master-pieces of modern art, it seems unlikely, however, that beauty is the foremost criteria for works of art.

Let’s look at an example: the famous ‘Black Square’ by Kazimir Malevich shown in Figure 6. Painted in 1915, it became the turning point not only in the development of Russian avant-garde. It is definitely one of the important works of Modern Art. Malevich or Mondrian, pursuing art as such, reduced their paintings to pure color and the most simple of geometric forms bringing to mind the thoughts of Eric Gill about “ornament & the ornamental”, about “humanity of craftsmanship” versus the “power of industrialism”.

When the ‘Black Square’ was first exhibited, people found it a strange thing and people still find it a strange object today. Figure 7 shows it hanging up in the corner where in a traditional Russian home you would expect the icon of a saint, painted on a golden background. If we consider time and place—the first world war was raging and in Russia the bleak conditions would soon explode in a revolution—we might get a glimps of what moved Malevich to paint it and hang it like this. I have not seen the original painting myself, but I can imagine the ‘Black Square’ having a mystic quality.

Is the ‘Black Square’ beautiful? And if it is, in what sense is it beautiful? What exactly is it that distinguishes a true piece of Art from any other object that is unusually expressive (like a comic
I am not an expert in Fine Arts and so I resort to my favorite book on Art: Ernst H. Gombrich’s ‘The Story of Art’[4]. He writes: “What an artist worries about as he plans his pictures, makes his sketches, or wonders whether he has completed his canvas, is something much more difficult to put into words. Perhaps he would say he worries about whether he has got it ‘right’. Now it is only when we understand what he means by that modest little word ‘right’ that we begin to understand what artists are really after.” And he illustrated his explanation by Raphael’s studies for the ‘Virgin in the meadow’. And I can’t resist to include the sketches and the completed painting as Figures 8 and 9. We can see how Raphael struggles with how to place the little boy, Jesus, in relation to his mother, but when we look at the completed painting, no traces of Raphael’s worries remain visible. Everything looks completely natural.

Gombrich’s words are the fitting introduction to the third and last program that I present here. This program also leads us back to Eric Gill and his thoughts about typography: the METAFONT program for the letter ‘S’. A comprehensive review of this program has been published by its author in ‘The Mathematical Intelligencer’[7, 9]), and I highly recommend to all the Art lovers to read this art-icle, because it not only explains a work of art but, as a piece of literature, it is a work of art itself.

Let me start by quoting two sentences from this article. Donald E. Knuth writes: “For three days and nights I had a terrible time trying to understand how a proper S could really be defined.” And “The purpose of this paper is to explain what I now consider to be the ‘right’ mathematics underlying printed S’s, and also give an example of the METAFONT language I have recently been developing.”

As most pictures that nowadays hang glazed and framed on the walls of museums were originally commissioned by a customer and made for a definite occasion and a definite purpose, this program was not created as an exercise in abstract art, but as a utility to be used. As an indication, I should mention that the program itself is found only in the appendix of Donald E. Knuth’s paper. Still, we can safely assume that every one of its features is a result of a decision by the artist: that he may have pondered over them and changed them many times, until every detail looked ‘right’, and nothing could be added or removed without disturbing the balance of the whole.
In his paper Donald E. Knuth, after explaining the historical background in type design, decides to study boundary curves defined by ellipses, the all-time favorite curve of classical mathematics.

Figure 10: Donald E. Knuth: Joining an ellipse and a straight line.

Starting with the problem of joining a straight line with an ellipse (see Fig. 10), he develops some beautiful mathematics and arrives at the solution:

\[
\begin{align*}
  x &= x_t + \frac{2\sigma(x_l - x_t)^2(y_t - y_m)}{\sigma^2(x_l - x_t)^2 + (y_t - y_m)^2}, \\
  y &= y_m + \frac{2\sigma^2(x_l - x_t)^2(y_t - y_m)}{\sigma^2(x_l - x_t)^2 + (y_t - y_m)^2}, \\
  y_t &= y_t - \frac{(y_t - y_m)^2 - \sigma^2(x_l - x_t)^2}{2(y_t - y_m)}. 
\end{align*}
\]

The \textsc{metafont} language is a perfect match for the problem at hand. Surprisingly, the four lines of the \textsc{metafont} program\cite{7} that embody these equations looks even simpler than the three equations above:

```latex
subroutine scomp(index i) % starting point
  (index p) % turning point (yp to be defined)
  (index j) % transition point (to be defined)
  (index k) % ending point
  (var s): % ending slope
  % This subroutine computes yp, xj, and yj so that
  % yp - yj = s*(xk - xj) and so that the following curve
  % is consistent with an ellipse:
  % i{x_p - x_i, 0} .. p{0, y_p - y_i} .. j{x_k - x_p, s*(x_k - x_p)}.
  %
  y_k - y_j = s*(x_k - x_j);
  new a, b: a = s*(x_p - x_i); b = y_k - y_i - s*(x_k - x_i);
  x_j - x_i = -2a*b*(x_p - x_i)/(a*a + b*b);
  y_p - y_i = .5(b*b - a*a)/b.

Figure 11: Donald E. Knuth: The letter S.
```

In the words of Eric Gill, we “got a system of manufacture which naturally produces them” and we are “getting the things which that system makes best.”

The code is nicely arranged in three parts of increasing size and decreasing density and complexity: from the abstract joining of an ellipse with a straight line, to the composition of an S-curve, to the application of finishing touches.

The \textsc{sdraw} routine uses the \texttt{scomp} routine, shown before, to join the four ellipses defining the outer and inner boundaries of the rounded parts of the
S to the straight straight lines defining the middle stroke:

\begin{verbatim}
subroutine sdraw(index i)  % starting point
  (index p)  % upper turning point (y_p to be defined)
  (index k)  % middle point
  (index q)  % lower turning point (y_q to be defined)
  (index j)  % ending point
  (index a)  % effective pen width at turning points
  (index b)  % effective pen height at middle point
  (var s)  % slope at middle point

open:  top_y = top_y_k;  bot_y = bot_y_k;
x_5 = x_6 = x_k;
rt_x_p = rt_x_1;  lft_x_p = lft_x_2;
rt_x_q = rt_x_9;  lft_x_q = lft_x_10;
y_2 = y_p;  y_3 = y_q;
call scomp(i, 1, 3, 5, s);  % compute y_1 and point 3

end:
call scomp(i, 2, 4, 6, s);  % compute y_2 and point 4

call scomp(i, 9, 7, 5, s);  % compute y_3 and point 7

call scomp(i, 10, 8, 6, s);  % compute y_4 and point 8


c draw:  uwdraw i(x_1, x_0) . . 1(0, y_1 - y_1) . .
  3{x_q - x_p, s(x_q - x_p)} . . 7{x_q - x_p, s(x_q - x_p)} . .
  9{0, y_i - y_i} . . 3{j - x_q, y_q};
i\{x_2 - x_0} . . 2{0, y_2 - y_1} . .
  4{x_q - x_p, s(x_q - x_p)} . . 8{x_q - x_p, s(x_q - x_p)} . .
  10{0, y_i - y_i} . . 3{j - x_3, x_3, 0},  % the s-curve

What remains to be done before a final call to the sdraw routine is defining the key points and attaching the serifs:

“The letter S”;

hpen:  top_y = round(h + o);  bot_y = -o;
x_3 = 5u;  y_3 = 52h;
lft_x_2 = round u;  rt_x_4 = round 9u;
x_3 = 4.5u;  x_3 = 5.5u;

lft_x_6 = round u;  rt_x_7 = round 8.5u;
y_6 = good 3/4 h - 1;  y_7 = good 3/4 h + 1;

bot_y_8 = 0;  y_8 = y_6;  x_8 = x_6;  rt_x_9 = rt_x_9;
top_y_10 = h;  y_11 = y_7;  x_10 = x_7;  lft_x_11 = lft_x_11;
w_6 ddraw 6 . . 8.9 . . 3;  % lower serif
ddraw 7 . . 10, 11 . . 10;  % upper serif

pen#:  w_4 draw 6(0, -1) . . 5(1, 0);  % erase excess

pen#:  w_4 draw 7(0, 1) . . 1(-1, 0);  % ditto

top_y_10 = 0. w_0 draw 6(.0, 1) . . 5(1, 0);  % lower left stroke
draw 7(0, 1) . . 1(-1, 0);  % upper right stroke
call “a sdraw(1, 2, 3, 4, 5, 8, 9, -h/(50u)).  % middle stroke

The program is an elegant formulation of a non-trivial problem. Some people might disagree on whether this program really captures the essence of the letter S or may not share the authors interest in the subject, but without doubt, the program achieves what it is supposed to achieve, with clarity and elegance, with a perfect balance between readability and brevity, simplicity and generality. In my judgment this is a master-piece.

5 Conclusion

Comming to the conclusion, you might expect an answer to the question which is the title of this article. But I have to caution you.

In the first sentence of the Introduction of ‘The Story of Art’[4], Ernst Gombrich writes: “There really is no such thing as Art. There are only artists.” And he finished his book by repeating this statement and adding: “There are only artists—men and women, that is, who are favoured with the wonderful gift of balancing shapes and colours till they are ‘right’, and rarer still, who possess that integrity of character which never rests content with half-solutions but is ready to forgo all easy effects, all superficial success for the toil and agony of sincere work. Artists, we trust, will always be born. But whether there will also be art depends to no small extent on ourselves, their public. By our indifference or our interest, by our prejudice or our understanding we may yet decide the issue.”

I should add that teaching my students ‘how to program’, I often must realize how few of them are born artists in the art of computer programming. All my students, however, benefit tremendously by exposing them to examples of true art in computer programming: example programs that are so well written that the computer program seems to be the most natural expression of its content. The influence these programs have on the ability of every one of my students to write decent computer programs can not be underestimated.

If you know the works of Malewich or Mondrian, your will see their influence in many well designed contemporary objects of daily use. If you know the computer programs of Donald E. Knuth, you will see their influence in the source code of many well designed programs that run every day on your computer.

List of Figures

References


[7] 0.10 Exhibition, 1915, Petrograd, Photo: Wikimedia Commons.


