Beauty and Computer Programming

Victor T. Norman
Calvin College
August, 2011

Abstract

This article asks two big questions: First, could one consider the act of computer programming to be a form of artistry, and thus regard the programmer as an artist? Second, can a computer program truly be beautiful, in a similar way that a painting, musical work, or sculpture is beautiful?

We argue that the creation of a computer program and the creation of a piece of fine art share so many characteristics that one can rightfully consider programming to be a form of artistry. We then argue that, just as there is ugly code, there is also beautiful code. We identify some qualities of code that one should pursue in order to create beautiful code.

Introduction

All computer science educators have seen "ugly code" submitted by students for programming assignments. We know what ugly code looks like: it is poorly indented, poorly documented, not robust, uses inappropriate data structures, uses poor identifier names, etc.

But, what difference does it make that the code is ugly? If the code works correctly, who should care that it is "ugly"?

While aesthetics has been widely discussed and investigated in the area of user interfaces (Isaacs and Walendowski 2001) little has been written about the definition of or importance of beauty in computer code. Computer science educators have talked about "rediscovering the passion, beauty, joy, and awe" in programming (McGettrick et al. 2008), but little has been written about what beauty really means in computer code.

This paper investigates the question of whether computer code can actually be beautiful, similar to the beauty found in a painting or a sculpture. Is the purpose of writing computer code simply to produce something that works? Or should we be concerned about the code's internal beauty? Are the "innards" of a program not important? We first investigate a similar question: if a programmer can create "beautiful code", then is the programmer correctly considered an *artist* and could the program itself be considered *art*?

The Artistry of Programming

In 1968 Donald Knuth published the first of a seven volume series of books on computer science, and surprisingly titled it *The Art of Computer Programming* (Knuth 1968). Knuth writes in the preface,

> The process of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music.

However, Knuth does not go on to expound on this idea -- that is the end of the discussion in the book. The rest of the book is a masterful and comprehensive catalogue of algorithms.

(Similarly, in the book *Beautiful Code* (Oram and Wilson 2007), chapter two contains a section called "But Is It Art?" that does not address that question at all.)

Knuth's statement above seems to imply that the process of creating a computer program is an art (perhaps better referred to as *artistry*), but he does not explicitly state that the product – the program itself – is a work of art. However, six years later, Knuth gave a speech called *Computer Programming as an Art*, in which he explains his choice of the word *art*.

> When I speak about computer programming as an art, I am thinking primarily of it as an art *form*, in an aesthetic sense. The chief goal of my work as educator and author is to help people learn how to write *beautiful programs.* [...] My feeling is that when we prepare a program, it can be like composing poetry or music; as Andrei Ershov has said, programming can give us both intellectual and emotional satisfaction, because it is a real achievement to master complexity and to establish a system of consistent rules. (Knuth 1974)

Knuth makes two important statements: first, he is stating that there is not only artistry in creating a computer program, but also that the product is art. One can be an artist because one has many choices to make while programming (at least when working on a non-trivial program) and thus one can employ creativity and imagination. Creating a program is not just assembling parts in the correct order, with no room for creativity.

Second, Knuth is tying the artistry of computer programming to beauty. He states that his chief goal is to teach students to use their artistry to create beauty in their computer code.

How similar is the creation of fine art (paintings, sculptures, music, etc.) to the creation of a program? The next section explores this question.

Similarities between the Creation of Fine Art and a Computer Program

Art exists in order to communicate -- ideas, emotions, memories, etc. Similarly, programming *is* communication. A programmer is, of course, communicating with the computer, instructing the computer what operations to execute to solve a problem. However, a programmer is also

communicating with the people (including the programmer himself) who may need to execute, alter, and maintain the program in the future. Donald Knuth talks about this idea in his book titled *Literate Programming*:

> Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do. (Knuth 1992)

Second, the process of creating a piece of fine art or a computer program forces the creator to understand what he or she is trying to communicate more fully. In both cases, the process of creating -- the *reifying* of the artwork or computer program -- often causes the artist/programmer to see the problem or goal differently, i.e., to understand differently or more fully what needs to be communicated. Programmers often create an initial prototype of a program, and show it to a customer, or work with it themselves. This is useful because often a programmer does not fully understand the problem to be solved until implementing the code to solve it. Only then does the programmer understand, and only then can the programmer implement the code correctly and efficiently, and perhaps, beautifully. Note that a more experienced artisan (be he a visual artist or a computer programmer) may be able to skip implementing a prototype, or making initial sketches, and may move straight from understanding the problem (or what needs to be communicated) to implementing a solution. (Luttikhuizen 2010)

Third, the classical artist and the computer programmer both have various tools they can use to communicate their ideas, and the person's experience and abilities with his tools have a major impact on the person's ability to efficiently and effectively communicate his ideas. The painter has the canvas, color, paint brush, knives, etc., whereas the programmer has the choice of language, editor, debugger, static code analyzer, documentation generator, reference materials, etc. A novice artisan wields his tools clumsily, creating a product slowly, inefficiently, and perhaps ineffectively. A mature artisan wields his tools with dexterity, using the best tools at the best times to produce the best result quickly and effectively. It is for this reason that an instructor needs to teach her students not only the syntax of a programming language, or what a good painting looks like, but also how to use the available tools most effectively while programming or painting. (Wolterstorff 1980)

Fourth, experienced students of an art form are best able to appreciate the art. This holds true for both traditional art and for computer programs, although, perhaps to a different degree. It takes a trained eye to fully appreciate what is being communicated in the media. For computer programs, a person who has never seen code before will perhaps be unable to see any beauty at all in a program. However, for a person who has studied thousands of programs and hundreds of thousands of lines of code, that person may find beauty in the organization, order, naming, documentation, and efficiency in code.

Finally, both artists and computer programmers report getting “in the zone”, to put it colloquially:

> Many artists have spoken of seeing things differently while drawing and have often mentioned that drawing puts them into a somewhat altered state of awareness. In that different subjective state, artists speak of feeling transported, 'at one with the work', able to grasp relationships that they ordinarily cannot grasp. Awareness of the passage of time fades away, and words recede from consciousness. Artists say that they feel alert and aware, yet are relaxed and free from anxiety, experiencing a pleasurable, almost mystical activation of the mind. (Edwards 1989)

Beautiful Programs

So, there exist many similarities between the creation of fine art and the creation of a computer program. But the question still remains: What is a beautiful program? What does it really look like? Knuth offered these criteria, which have certainly stood the test of time (Knuth 1974):

- Utility: the "usefulness" of the program, to yourself, and especially to others.
- Correctness: a beautiful program must produce correct results, not in just the usual cases, but in the unusual or corner cases. One could easily argue that this criterion trumps all the others, for no one wants to use a program that is untrustworthy.
- Robustness: a beautiful program should be easy to modify and maintain, and extend to solve similar problems, take different input, or produce its output differently.
- Readability/understandability: a beautiful program must be readable to you and others, and not only the comments in the code, but also the code itself. Readability is enhanced by proper and consistent naming conventions, so that identifiers in the program (variable names, function names, class names, etc.) describe what they do or are used for. The judicious use of comments also makes the program more readable and thus more maintainable. Adam Kolawa in *Beautiful Code* states: "The problem with the new code being written today – especially in the C++ language – is that developers use so much inheritance and overloading that it's almost impossible to tell what the code is really doing, why it's doing it, and whether it's correct." (Oram and Wilson 2007)
- Clarity of interface: if the program interacts directly with a human user, the program's user interface must be beautiful -- not only in its use of colors, shapes, and fonts -- but in its ease of use. Similarly, for a program that interacts only with other programs, the inputs and outputs of the program should be well defined, easy to understand, and integrate with.
- Efficiency: a beautiful program should use memory and computer processing power efficiently, using the most appropriate data structures and algorithms to solve the problem, while still remaining robust.

Others have created their own criteria. In the book *Beautiful Code* (Oram and Wilson 2007), various authors submit these criteria:

- Elegant design: this includes not only the proper use of language elements (lists, arrays, hash tables, etc., which may be built into the language), but also the best use of design patterns for decoupling dependencies between components, for example.

- Conciseness/simplicity: code should omit needless words, be brief, and be as simple as possible. Yukihiro Matsumoto says, "We often feel beauty in simple code. If a program is hard to understand, it cannot be considered beautiful."
- Standards-based/familiarity: use of existing standards should be followed, to improve familiarity and consistency. "Never be too innovative."
- Modularity/loose-coupling: modules in the code should be self-contained, and as loosely coupled as possible.
- Scalability: when appropriate, code should efficiently scale to handle large data sets or heavy use.
- Balance: an appropriate balance of all these criteria.

These criteria for a beautiful program still allow for a great deal of freedom: no single naming scheme is defined, no single indentation scheme is required, no commenting rules are prescribed, etc. The important point is that the programmer must be thorough, intentional, and consistent, keeping in mind the future readers and users of the code.

Beautiful Code is Hospitable Code

Thoroughness and consistency in code give the reader of the code a sense of being welcomed, a sense of warmth, and a confidence that this code has been created with care. In short, the code is *hospitable.* When a guest arrives at a hospitable home, the guest sees that it is clean, properly lit, comfortable, and warm. Similarly, hospitable code welcomes the reader to come in and be comfortable, to enjoy the cleanliness of the code, to feel at home, and to see that the space has been carefully prepared for guests.

An Open Question

We have argued that the creation of a computer program can be an artistic activity. We have also argued that beauty can be found in a computer program (a beauty perhaps only appreciable by skilled artisans of programming). We have, however, not addressed an important fundamental question:

Can a computer program itself be considered a piece of art?

This is an open question to be explored elsewhere.

Conclusion

Computer programming shares many characteristics with that of the creation of traditional fine art, even though it is usually more closely associated with a science. One can create beautiful computer programs, just as one can create beautiful works of art. The similarities between the creation of traditional art (paintings, sculptures, music) and creation of computer code are

numerous. Beautiful code is consistent and thorough in its use of naming, comments, data structures, definitions of interfaces, and design. It is concise, efficient, scalable, and understandable, always keeping in mind the most importance consumer: programmers coming later who need to maintain the code.

References

[1] Edwards, Betty. 1989. *Drawing on the Right Side of the Brain*. Revised. Tarcher.

[2] Henry Luttikhuizen. 2010. Interview with Henry Luttikhuizen, Professor of Art at Calvin College. January 21.

[3] Isaacs, Ellen, and Alan Walendowski. 2001. *Designing from Both Sides of the Screen: How Designers and Engineers Can Collaborate to Build Cooperative Technology*. Sams.

[4] Knuth, Donald E. 1968. *Art of Computer Programming, Volume 1: Fundamental Algorithms*. 3rd ed. Vol. 1. Addison-Wesley Professional.

[5] ———. 1974. "Computer Programming as an Art." *Communications of the ACM*: 667-673.

[6] ———. 1992. *Literate Programming*. 1st ed. Center for the Study of Language and Inf.

[7] McGettrick, Andrew, Eric Roberts, Daniel D. Garcia, and Chris Stevenson. 2008. Rediscovering the passion, beauty, joy and awe: making computing fun again. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, 217–218. SIGCSE '08. New York, NY, USA: ACM. doi:10.1145/1352135.1352213. http://doi.acm.org/10.1145/1352135.1352213.

[8] Oram, Andy, and Greg Wilson. 2007. *Beautiful Code: Leading Programmers Explain How They Think (Theory in Practice)*. 1st ed. O'Reilly Media.

[9] Wolterstorff, Nicholas. 1980. *Art in Action: Toward a Christian Aesthetic*. Wm. B. Eerdmans Publishing Company.