Seminar Report

Piet
An Artistic Programming Language

Manfred Moosleitner
manfred.moosleitner@student.uibk.ac.at

31 July 2015

Supervisor: Michael Färber

Abstract

This seminar report is about the esoteric programming language Piet, which uses pictures as programs. It introduces the building blocks and mechanics behind Piet. The report also features a step-by-step walk through by the means of executing an example program and display of example programs. Additionally the report shows a tool set for creating and running Piet programs and lists current usage and possible additions.

Contents

1 Introduction

In the field of computer programming exist a multitude of paradigms and principles like *imperative*, *declarative* or *structured*, just to name a few, and a large number of programming languages using one or more paradigms were created over the years. One matter most programming languages have in common is the use of an alphabet with Roman characters and special characters like brackets. With this alphabet, programs are then written and well known constructs like *loops*, *if-then-else* and such are created.
Other concepts were also tried, the so called *visual programming languages*, which are nowadays used throughout in education, game industry, multimedia, and other fields. The editors for these languages often offer building blocks for programming concepts like branching or looping and programs are built by dragging-and-dropping these blocks on a canvas like workspace inside the editor and connecting them with arrows to build the program flow. Another unusual concept was conceived by David Morgan-Mar in the late 2000's, because he wanted programs to look like paintings. This led to the development of the programming Language *Piet*. Programs that are build in Piet are painted, thus pictures are used as program code.

Overview: This report starts with an introduction of the author and states the origins of Piet. The following three sections consist of the language specification of Piet with a traversal through a program, the presentation of a tool set to create and run Piet programs and the display of an assortment of example programs. The report succeeds with a listing of instances of usage of Piet and finishes with the conclusion.

2 Origins of Piet

The esoteric programming language Piet was created by David Morgan-Mar, an Australian Scientist in the field of Astrophysics. He taught lectures and classes in universities and schools in Australia and worked for different institutes and companies like the Australian Telescope National Facility and IBM Global Services Australia to name 2 and he writes books and publications in the field of GURPS (Generic Universal Role Playing System). Beside Piet, David Morgan-Mar also created some other esoteric programming languages like Chef (programs are in the form of recipes), Ook (a derivation of Brainfuck) or Zombie (a programming language for necromancers) [1].

The name giver for Piet is Piet Mondrian, a Dutch artist who was born in 1872 in the Netherlands. After his education and first works as artist, he went to Paris for a few years, came back to the Netherlands before going to Paris again for several years and moved to London on the verge of World War II. In 1940 he finally left Europe and moved to New York, where he died in 1944 [7].

Piet Mondrian is the founder of the „Neoplastizismus“ [2], where paintings are mostly constructed with horizontal and vertical lines building rectangles which can be filled with

colours.

This resembles the vision of David Morgan-Mar about Piet, as can be read on his web page:

> „Piet is a programming language in which programs look like abstract paintings. The language is named after Piet Mondrian, who pioneered the field of geometric abstract art.“[4]

The following will give an overview about the concepts behind Piet, how programs are constructed and how an interpreter traverses a program on execution.

3 Language Specification

It is worth mentioning that at the present time only interpreters exist for Piet. Piet is stack based and its stack works solely with integer values.

3.1 Alphabet

Instead of a using an alphabet consisting of Roman letters, Piet uses 6 basic „colourful colours“ and black and white. The colourful colours are red → yellow → green → cyan → blue → magenta → red, which form the hue-cycle as indicated by the arrows. These 6 basic colours are used in 3 different intensities and thus forming a lightness-cycle which goes like this: light → normal → dark → light. The colours and cycles are shown in Table 1.

light	red	yellow	green	cyan	blue	magenta
normal	red	yellow	green	cyan	blue	magenta
dark	red	yellow	green	cyan	blue	magenta
	white			black		

Table 1: Colour- and lightness-palette used in Piet. [4]

The colourful colours are used to encode commands while black and white are special colours. Before explaining the meaning of the colours, the equivalent of tokens, as can be found in other programming languages, needs to be introduced and how an interpreter moves through a program.

3.2 Tokens

The smallest entity is called **codel** and this name is made up of *code* and *pixel*. A codel can consist of a single pixel or of several pixels in enlarged programs. The next bigger entity is called **colour block** which is a continuous block of codels of the same colour. The number of codels in a colour block is used as an integer value. These values can be stored on and retrieved from the above mentioned **stack**. An example how codels and colour blocks are used to build programs can be seen in Figure 1, which is a picture of an execution trace of a Piet program.

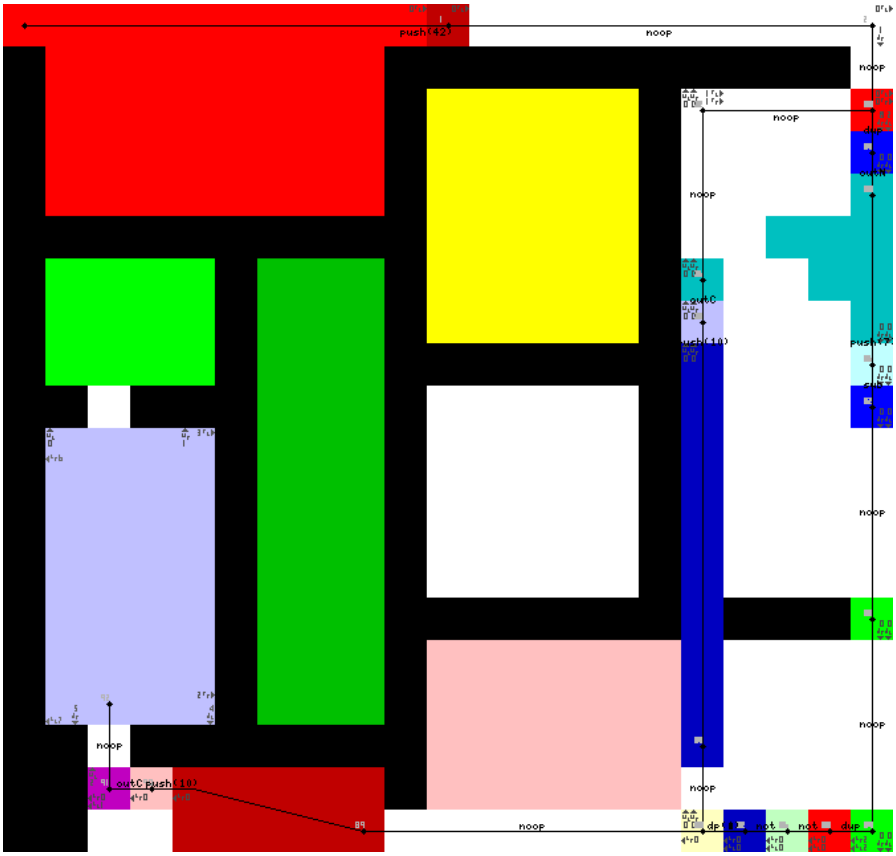

Figure 1: Graphical execution trace of an example Piet program.

3.3 Program Traversal

An interpreter starts its traversal of a Piet program always at the top left codel and this codel must not be of black colour. The interpreter moves from one colour block to another and uses two pointers to decide which colour block is next.
The first one is the **direction pointer (DP)**. The DP can point either right, down, left

or up and points to the right at the beginning. From the starting or entry point of a colour block, the interpreter searches for the border of the current colour block, which is furthest away of the starting position, in the DP's active direction.
The other one is the **codel chooser (CC)** which can point either to the left or to the right. At the start the CC points to the left. The interpreter uses the CC to choose a codel along the edge and so decides which one of the neighbouring colour blocks along the border is the next one to move into. The possible combinations of DP- and CC-directions and their meaning can be seen in Table 2.

DP	CC	codel
right	left	upper-most
	right	lower-most
down	left	right-most
	right	left-most
left	left	lower-most
	right	upper-most
up	left	left-most
	right	right-most

Table 2: Possible combinations for DP and CC and their meaning [4].

The edges of a program and black colour blocks cannot be penetrated and so limit possible movement options. When the interpreter's next move would be into a black colour block or over the border, the CC changes direction to choose another codel and tries to move again. If further movement is not possible, the DP's direction is turned clockwise for one step and the interpreter tries again if movement is possible and repeats to alternately change the direction of CC and/or DP till either it can move to another colour block or triggers program termination after switching the direction of DP and CC 8 times in total and still being inside the same colour block. A block that is build solely to terminate the program is called „trap“.

If the interpreter moves into a white colour block, it follows the current direction of the DP in a straight line until a colour block is reached and entered or a black colour block or edge is found. In the latter case the CC is toggled and the DP is turned one step clockwise right away. It then either moves on in a straight line when the white colour block continues, enters the colour block if one is there or reruns the above procedure if an obstacle is found.

3.4 Commands

Two things happen when the interpreter moves from one colour block to the next one. First, it triggers the execution of commands and these commands are encoded in the amount of steps taken in hue and lightness of the previous colour block to the current one, e.g. moving from red to green with the same lightness means „2 steps in hue“ or

when going from dark green to normal green means „2 steps in lightness“. Note that moving from dark to light means *one step darker*. With this method we can encode 17 different commands, as can be seen in Table 3 and some exemplary comand-description can found in Figure 2.
Second, it counts the number of codels of the previous colour block and pushes this value on the stack in case of a `push`-command. Moving into or exiting out of a white colour block does not trigger any commands and the amount of codels in a white colour block does not mean anything.
For all commands applies that if values from the stack are used by a command, they are poped off the stack, thus removed from the stack and are no longer available for further use. If values are needed for later use, they must be duplicated beforehand.

	Change in Lightness		
Change in Hue	None	1 darker	2 darker
none		push	pop
1 step	add	subtract	multiply
2 steps	divide	mod	not
3 steps	greater	pointer	switch
4 steps	duplicate	roll	in(number)
5 steps	in(char)	out(number)	out(char)

Table 3: Available commands in Piet [4].

This concludes the introduction to the mechanics behind Piet. What follows is a step-by-step walk through of the example program from Figure 1.

3.5 Program Execution

The program shown in Figure 1 displays examples of basic flow control including decremental of a value, looping until a certain value reaches zero and program termination. The highly enlarged picture is an execution trace of the program and was created by the interpreter npiet, see section Tools for more details. Program execution follows along the thin black line and the executed commands are printed between the colour blocks.

As stated earlier, the interpreter starts at the top left colour block with colour normal-red and it consists of 42 codels. The interpreter moves into next colour block consisting of 1 single dark-red codel and so the `push` command is triggered. Execution follows the white colour block until the upper-right corner of the program, where the CC is toggled, the DP is turned clockwise to point downwards and the interpreter continues in this direction. Here is the start of the loop.
The next sequence of colour blocks on the right border of the program first duplicate the top value at the stack (normal-red → normal-blue) and then print this value to standard output as number (normal-blue → dark-cyan). The following colour block is

- **push:** pushes number of codels of the colour block just left onto the top of the stack
- **add:** adds the 2 top values and pushes the result on top of the stack
- **subtract:** subtracts 1^{st} top value from the 2^{nd} top value and pushes the result onto the stack
- **pointer:** uses the top most value and rotates the DP counter-clock-wise for as many steps as the used value
- **switch:** uses the top most value and changes the CC regarding to the value used
- **roll:** uses the 2 top most values of the stack and changes position of the remaining values according to the 2 values, i.e. top most values gives the position that is targeted and the 2nd value indicates the depth. This means when the first value is 1 and the second value is 5, the top value of the remaining stack is moved to the fifth position on the stack while the position of the values at position 2 to 5 are moved one position up.

Figure 2: Description of some commands [4].

dark-cyan and has 7 codels and this value is pushed onto the stack when moving into the light-cyan colour block (dark-cyan → light-cyan). As the next colour block is normal-blue, `subtract` is triggered. This uses the top value 7 and subtracts it from the 2nd top value 42 and pushes the value 35 as result onto the stack. Below are 2 white colour blocks with a single normal-green codel in between. Following the specifications this does not trigger any commands.
As program flow reaches the right-bottom corner at the normal-green codel, the DP is turned to point to the left. The change from normal-green to normal-red triggers duplication of the value 35 on the stack. The next block is light-green and executes a `not` and the value 35 is replaced by the value 0. The following colour block is dark-blue and this triggers another `not` and so replaces the 0 by a 1. The DP's active direction is left and as the colour of the next codel is light-yellow, the `pointer` command turns the DP one step clockwise and so the DP is pointing up. On the way up, the value 10 is pushed onto the stack and print out as a character, which is a carriage-return (10 dark-blue → light-blue → dark-cyan).
The interpreter now reaches the black colour block, toggles the CC and turns the DP to the right and arrives at the start of the loop at the single normal-red codel. Here the top value is duplicated again, printed as number and so on. This loop continues until the subtraction at the right side pushes the value 0 onto the stack. When the interpreter reaches the `dup-not-not` sequence at the right-bottom of the program, the `pointer` command is executed with the value 0 and that means program flow will continue straight instead of going up. There the value 10 is pushed onto the stack, printed to standard

output as character and the light-blue colour block is entered. Due to the layout of this block, DP and CC are toggled and turned 8 times in total without leaving the current colour block and so the program is terminated.

4 Tools

As Piet programs are pictures, programming in Piet is not as simple as using your most favourite text editor and just hacking away. Although it is be possible to use any graphic editing program which allows manipulation of single pixels, it would still be a hassle to manually type in the the corresponding RGB values for the different colours. Additionally it would be quite mundane to keep track of the colour changes and to manually calculate the next hue and lightness to execute the correct command.
Erik Schoenfelder managed to lower that burden with his all-in-one package npiet [3]. It comes with a command line interpreter called *npiet* along with the graphical editor *npietedit*. The main features of the editor are selection of current colour by clicking in a colour palette-like area and the more important function to select a specific command and the editor will automatically choose the right colour for that command. Figure 3 shows the GUI of the editor with the colour-palette and command-area and the example program from above.

Although the editor makes creating Piet programs a lot easier than using a standard graphic editing program, it sadly does not offer a colour-picker functionality which lets you pick the colour from the program area itself and so a person creating programs with this editor needs to keep track of the colours manually.

5 Examples and Possibilities

The example page on the creators page [6] shows a broad assortment of programs that were done in Piet. There one can find programs like the obligatory *Hello World!* along with other programs that print some text to standard output and one program is even a text adventure. Other example programs calculate the first 100 Fibonacci Numbers, test if an input number is prime or not, solve the Towers of Hanoi problem or compute factorials. Other programs can approximate the number π as integer value (the bigger the program - the more accurate the result), calculate the day of the week according to a date read from standard input or interpret Brainfuck programs. The example programs can be seen in Figure 4.

One question that often arises when discussing programming languages is: „What is the computational class of the language?“. The already mentioned Brainfuck interpreter, takes a program written in Brainfuck, followed by its input, as input and interprets and deals with the result of the Brainfuck program. This Brainfuck interpreter is important, because this means that Piet is turing complete as an interpreter exists in Piet, which is able to interpret a program written in the turing complete programming language

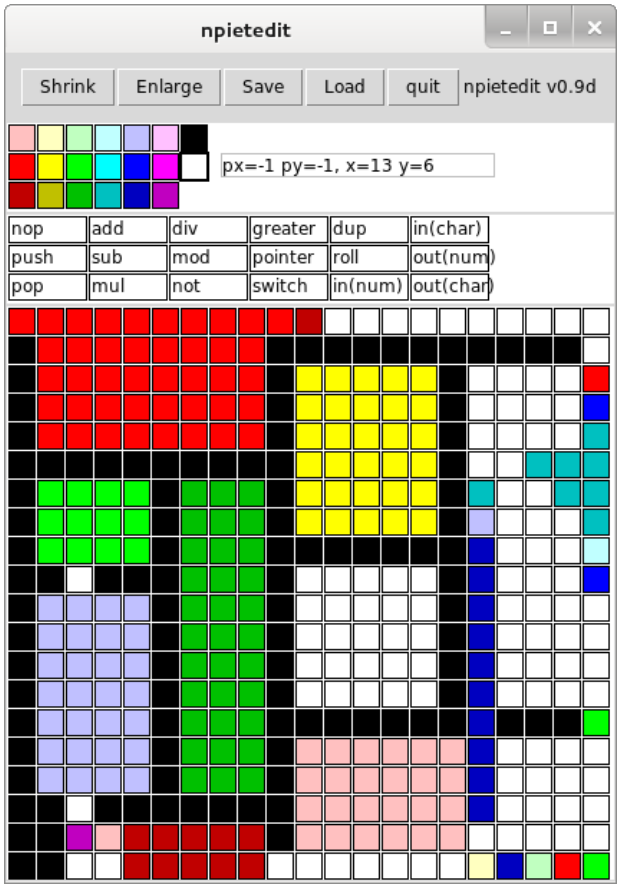

Figure 3: The graphical editor npietedit, showing the example program from above.

Brainfuck. Another hint is found on the web page *esolang.org*. The author of the article about Piet writes: „If the stack is allowed to hold any arbitrarily long number then it's very likely Turing-complete. There's no formal proof though." [5] and gives a sketch for a proof by using the argument that the roll-command can be used to simulate random-access on the stack and thus using it as a register bank.

6 Usage and Future

At the present time the community around Piet is not as big as more popular languages like C or Java as is indicated by the post count of about 60 of the only dedicated Piet forum I found at `http://piet.forumotion.com/`.
Piet was featured in the MIT Mystery Hunt in 2002 [1] and the language was mentioned in the „50 in 50" talk about history of computer languages from Guy L. Steele, known for his work with Lisp, Scheme and Emacs, and Richard P. Gabriel, known for his work

[1] `http://www.mit.edu/~puzzle/`

with Lisp and Lisp benchmarks [4].
As the concept behind Piet is unusual, it is also a subject in academic education and research, e.g. in specialization seminars.

Right now Piet only uses 6 basic colours and 3 intensities, thus making it possible to expand the language by adding more colours and possible steps in lightness. One idea would be to incorporate commands for drawing and thus making it possible to create Piet programs that draw other pictures.

7 Conclusion

Piet is a stack based, esoteric programming language which uses pictures as programs. Piet uses 20 colours as alphabet and blocks of colours as syntax elements. The available 17 commands are encoded in the difference between colours when moving from one block to another and uses one pointer for keeping track of the current moving direction and another pointer for choosing the next block. Piet features patterns for basic program flow like branching and looping and programs can read/print numbers/characters from and to standard output. A few Piet-interpreters and -editors exist and people use Piet to see what kind of programs they can build with it and study its unusual concept. The current language specification leaves room for expansion and new commands can be adding additional colours to interpreters.

Figure 4: 1^{st} row: Hello World (left) and 99-Bottles-of-Beer (right).
2^{nd} row: 1^{st} 100 Fibonacci numbers (left) and a primer tester (left).
3^{rd} row: Towers of Hanoi (left) and Day-of-the-Week calculator (right).
4^{th} row: approximation of π (left) and an interpreter for Brainfuck (left).

References

[1] David Morgan-Mar. http://www.dangermouse.net/me.html. Accessed: 2015-07-19.

[2] Neo-Plastizismus. https://de.wikipedia.org/w/index.php?title=Neoplastizismus&oldid=136855822. Accessed: 2015-07-19.

[3] npiet - an interpreter and editor for the piet programming language. http://www.bertnase.de/npiet/. Accessed: 2015-05-29.

[4] Piet. http://www.dangermouse.net/esoteric/piet/. Accessed: 2015-05-29.

[5] Piet - Esolang. https://esolangs.org/wiki/Piet. Accessed 2015-05-31.

[6] Piet Examples. http://www.dangermouse.net/esoteric/piet/samples.html. Accessed: 2015-05-29.

[7] Piet Mondrian. https://en.wikipedia.org/w/index.php?title=Piet_Mondrian&oldid=671941119. Accessed: 2015-07-19.