Piet++

This is a work in progress, and is still in development.

Piet++ is a variant of Piet, enabling access to multiple stacks within the main stack. The goal of **Piet++** is to provide a means to limit side effects from manipulating the stack while staying true to the feel and philosophy behind Piet. Piet++ provides a rich set of operators to manipulate the new stack type available, as well as expanding the set of operators available and providing a way to read and write to the bitmap that represents the program.

Contents [hide]

Language Concepts

Piet++ aims to remain true to the concept of Piet, and maintains the core concepts of Piet. Piet code takes the form of graphics made up of the recognized colors. Codel refers to a block of color equivalent to a single pixel. Color blocks are regions of equal colored codels connected in four directions. Program execution begins in the top left of the program, and the program maintains a direction pointer, initialized pointing right and able to point left, right, up, and down, and a codel chooser, initialized left and able to point left and right. As the program is executed, the interpreter traverses the color blocks of the program, according to the same rules as Piet:

1. The interpreter finds the edge of the current color block which is furthest in the direction of the DP. (This edge may be disjoint if the block is of a complex shape.)
2. The interpreter finds the codel of the current color block on that edge which is furthest to the CC's direction of the DP's direction of travel. (Visualize this as standing on the program and walking in the direction of the DP; see table at right.)
3. The interpreter travels from that codel into the color block containing the codel immediately in the direction of the DP.

The Colors

Piet++ uses the 6 bit color space. Command execution is controlled by the delta between the red, blue, and green channels between codels. The most significant bit of the green channel is ignored unless the color is #333/#FFFFFF (white) or #000/#000000 (black), which have the same behaviors as in Piet.

#000/#000000	#001/#000055	#002/#0000AA	#003/#0000FF	#010/#005500	#011/#005555	#012/#0055AA	#013/#0055FF
#100/#550000	#101/#550055	#102/#5500AA	#103/#5500FF	#110/#555500	#111/#555555	#112/#5555AA	#113/#5555FF
#200/#AA0000	#201/#AA0055	#202/#AA00AA	#203/#AA00FF	#210/#AA5500	#211/#AA5555	#212/#AA55AA	#213/#AA55FF
#300/#FF0000	#301/#FF0055	#302/#FF00AA	#303/#FF00FF	#310/#FF5500	#311/#FF5555	#312/#FF55AA	#313/#FF55FF
#020/#00AA00	#021/#00AA55	#022/#00AAAA	#023/#00AAFF	#030/#00FF00	#031/#00FF55	#032/#00FFAA	#033/#00FFFF
#120/#55AA00	#121/#55AA55	#122/#55AAAA	#123/#55AAFF	#130/#55FF00	#131/#55FF55	#132/#55FFAA	#133/#55FFFF
#220/#AAAA00	#221/#AAAA55	#222/#AAAAAA	#223/#AAAAFF	#230/#AAFF00	#231/#AAFF55	#232/#AAFFAA	#233/#AAFFFF
#320/#FFAA00	#321/#FFAA55	#322/#FFAAAA	#323/#FFAAFF	#330/#FFFF00	#331/#FFFF55	#332/#FFFFAA	#333/#FFFFFF

The Stacks

Piet++ is initialized with the *Stack Pointer* pointing towards an empty stack. Stacks and integers may be pushed onto the stack, and through commands the stack pointer can point to other stacks within the program. Except for the top stack, all stacks are identical, having a number of elements and a parent stack. There will always be one and only one way to access a stack that has been created. Commands which refer to or require a stack above the current stack will fail on the top stack, and the command will be ignored and execution will resume at the next instruction.

Commands

Piet++ provides a rich set of 31 commands with which to manipulate data on the stacks and manipulate the stacks themselves. When a commands behavior is undefined, it is recommended to ignore it and continue with execution.

Commands are defined by the transition of color from one color block to the next as the interpreter travels through the program. Commands are picked based on the difference between the first and second color modulo four in the red and blue channels and modulo two in the green channel, unless the color is black or white, when the transition is ignored and special behavior is executed.

Delta Green 0	**Delta Red**			
Delta Blue	DR 0	DR 1	DR 2	DR 3
DB 0	Noop	Push-Int	Push-Stack	Pop
DB 1	Dup	Roll	Roll-Context	Push-up
DB 2	Push Down	Pull Up	Up	Down
DB 3	Add	Subtract	Multiply	Divide
Delta Green 1				
DB 0	Mod	Negate	Not	Greater
DB 1	Equal	Lesser	Size	In Integer
DB 2	In Character	Out Integer	Out Character	Depth
DB 3	Read	Write	Pointer	Toggle

Push-int: Pushes an int equal to the size of the color block just exited onto the stack.

Push-stack: Pushes an empty stack onto the current stack.

Pop: Pops the top element off the stack and discards it.

Dup: Duplicates the top element of the stack, performing a deep copy if the element is a stack. In Piet++, there is always only way to access any type of data, and multiple references to data are not allowed.

Roll: performs exactly the same as in Piet: Pops the top two values off the stack and "rolls" the remaining stack entries to a depth equal to the second value popped, by a number of rolls equal to the first value popped. A single roll to depth n is defined as burying the top value on the stack n deep and bringing all values above it up by 1 place. A negative number of rolls rolls in the opposite direction. A negative depth is an error and the command is ignored. If a roll is greater than an implementation-dependent maximum stack depth, or one of the operands is a stack, it is handled as an implementation-dependent error, though simply ignoring the command is recommended.

Roll-Context: A trinary operator, it requires the stack to have two integers on the top and a series of nested stacks whose top variable is a stack, like so:

```
 x y S ...
     ╙ S ...
       ╙ S ...
         ╙ S ...
           ╙ S ...
             ╙ S ...
               ╙ ...
```

The stacks are then rolled in a manner similar to the roll command: the top stacks data is placed at a depth equal to the second value and all other stack's data moved up one stack. This is repeated a number of times equal to the first number, with a negative number rolling a stacks data from the bottom stack to the top of the stack heirarchy.

Examples:

Stack before *rolc*:

```
1 2 S 1
    ╙ 2
```

After:

```
S 2
╙ 1
```

Before:

```
1 3 S 1234 5
    ╙ S 234 90 8
      ╙ 8900 342 6
```

After :

```
S 234 90 8
╙ S 8900 342 6
  ╙ 1234 5
```

Before:

```
1 3 S 12 34 545 6
    ╙ S 342 67 S 123 8
      ║         ╙ 2 3 4 5
      ╙ S
        ╙ 8 90 69
```

After :

```
S 342 67 S 123 8
║         ╙ 2 3 4 5
╙ S
  ╙ S 12 34 545 6
    ╙ 8 90 69
```

Before:

```
1 3 S 1324 345 354 765 657
    ╙ S 132 657 68
      ╙ 2 S 34 234 5
          ╙ 342 645 324
```

After:

```
// The stack remains unchanged, as the stack hierarchy did not have sufficient depth to accommodat
1 3 S 1324 345 354 765 657
    ╙ S 132 657 68
      ╙ 2 S 34 234 5
          ╙ 342 645 324
```

Push up: Pops the top element of the current stack and pushes it onto the the stack above it.

Push down: If a stack is the second item on the current stack, it pops the top item off the current stack and pushes it onto the stack now at the top of the stack.

Pull up: If there is a stack at the top of the current stack, it pops the top value off that stack and pushes it onto the current stack.

Up: Moves the Stack Pointer one level in the stack hierarchy up.

Down: If the top item in the current stack is a stack, moves the Stack Pointer into that stack.

Add: Pops the top two elements of the stack. If both are integers, they are added together and pushed onto the stack. If one is an integer and one is a stack, the integer is pushed onto the stack or pushed onto the bottom, depending on if the integer is above or below the stack. If both are stacks, the bottom one is appended to the top one.

Subtract: If the top two items on the stack are integers, it pops the top two items from the stack and pushes the result of subtracting the top one from the second onto the stack.

Multiply: If the top two items on the stack are integers, it pops the top two items from the stack and pushes the result of their multiplication to the stack.

Divide: If the top two items on the stack are integers, it pops the top two items from the stack and pushes the result of the integer division of the second item by the top item onto the stack.

Mod: If the top two items on the stack are integers, it pops the top two items from the stack and pushes the result of modding the second item by the top item onto the stack

Negate: If the top item on the stack is an integer, it pops an integer x from the stack, and pushes -x onto the stack

Not: If the top item on the stack is an integer, it pops an integer from the stack and pushes 0 if the number is not equal to 0 and 1 if it is equal to zero.

Greater: If the top two items on the stack are integers, it pops the top two items from the stack and pushes 1 on to the stack if the second value is greater than the top value, and pushes 0 if it is not greater.

Equal: If the top two items on the stack are integers, it pops the top two items from the stack and pushes 1 on to the stack if they are equal, and pushes 0 they are not equal.

Lesser: If the top two items on the stack are integers, it pops the top two items from the stack and pushes 1 on to the stack if the second value is lesser than the top value, and pushes 0 if it is not lesser.

Size: Pushes the size of the top item onto the stack. The size of a stack is the number of elements inside it, and the size of an integer is -1.

In: Reads a value from STDIN as either a number or character, depending on the particular incarnation of this command and pushes it on to the stack. If no input is waiting on STDIN, this is an error and the command is ignored. If an integer read does not receive an integer value, this is an error and the command is ignored.

Out: Pops the top value off the stack and prints it to STDOUT as either a number or character, depending on the particular incarnation of this command. If the top item on the stack is a stack, the stack is popped off the stack and Out is called on it's elements.

Depth: Returns how deep in the stack hierarchy the pointer is.

Read: If the top two objects on the stack are integers, the top value is popped off the stack and is added to the pointers y position, and the x value is added to the pointers x position. The RGB value of the codel at that location is pushed onto the stack. Whether the alpha channel is considered is implementation and format dependent. IF the pointer would go off the program, that is an error and the command is ignored. The top left codel is codel (0,0), and the bottom right codel has the values of the width and height of the program minus one.

Write: If the top three objects on the stack are integers, the top value is popped off the stack and is added to the pointers y position, and the x value is added to the pointers x position. The third value is popped of the stack, and the RGB value of the codel at the determined location is set to that value. Whether the alpha channel is considered is implementation and format dependent.

Pointer: Pops the top value off the stack and rotates the DP that many times clockwise, counterclockwise if negative.

Toggle: Pops the top value off the stack and toggles the CC that many times.

Categories: Languages | Non-textual | Stack-based | Turing complete | Two-dimensional languages | Self-modifying | Unimplemented