Implementing flood fill

Paint programs often offer a *flood fill* function. When you click a point, the entire region around the point is painted with your selected color. How does this work?

Bitmaps

A bitmap is a two-dimensional array, where each entry corresponds to the color of one *pixel*. A pixel ("picture element") is a little square on the screen. There are different ways to specify color (a typical way is to use 8 bits each for red, green, and blue intensity information, so 24 bits in total). We will assume that the color is an `Int`.

To compute the region containing the clicked pixel, we need to define which pixels are "neighbors". We use the convention that pixels have four neighbors: up, down, left, and right. We do not consider the diagonally adjacent pixels as neighbors.

The first algorithm - using recursion

The first thing to perform flood filling is remembering the old color of the pixel to be painted. Pixels with the old color would be filled, and other pixels take a role of a boundary. Then we color the pixel, and we see all neighbors of the pixel. If there is a neighbor pixel with the old color, it should be painted, so we make the recursive call on the pixel.

We can prove that the algorithm fills the whole region to be painted by mathematical induction. If there are only one pixel, it would be wholly painted. If the region to be painted contains more than a pixel, the region would be still connected or separated into at most four pieces. Since they contains pixels of which the number is less then the number of pixels in the original region, and we will make the call recursively on the new regions, we know that they will be all filled by mathematical induction.

Is this good algorithm? The depth of the recursion may be large (the best bound we have is the number of pixels in the region). This algorithm can easily cause a stack overflow error.

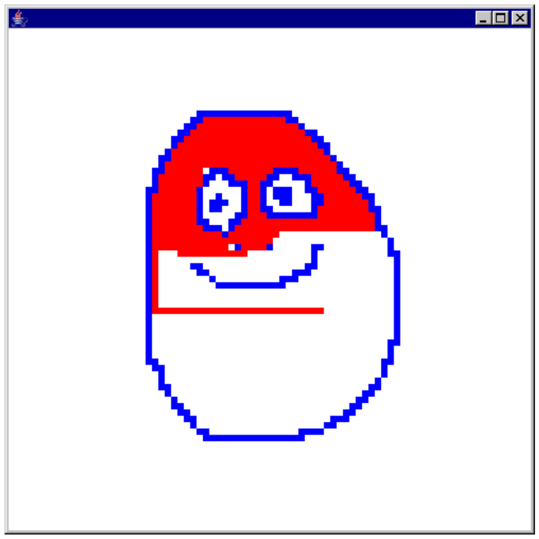

Figure 1: The first algorithm - using recursion

The second algorithm - using a stack

We discussed before that every recursive algorithm can be converted to an iterative algorithm using a stack. Our second algorithm does this.

This algorithm is not really different from the recursive algorithm—instead of the runtime stack, we use a stack implemented in Scala. The advantage is that our own stack can be made as large as we want (if there is enough memory), and pixel objects are much smaller than stack frames. This algorithm will not cause out of memory errors easily.

There is no real difference in runtime: In both algorithms the computing time is proportional to the number of pixels we have to paint and we cannot do better.

In the animation, green pixels denote the pixels in the stack, and red pixels denote pixels that have already been popped from the stack.

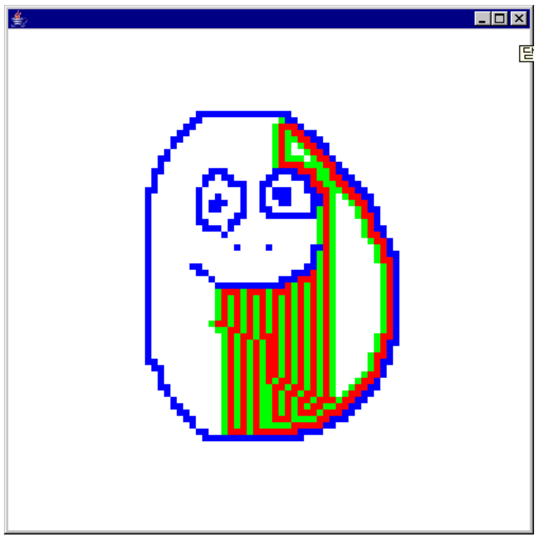
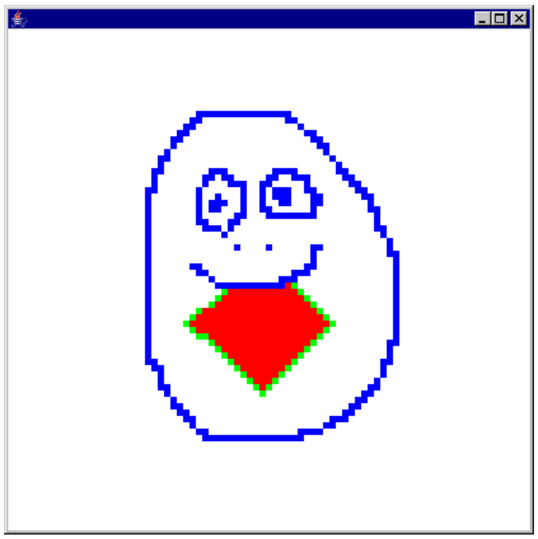

Figure 2: Using a stack, and using a queue

The third algorithm - using a queue

If we think about it a little, we realize that the algorithm in the previous section works correctly no matter in which order we retrieve the stored pixels from the data structure. And in fact, there is a better order, in which the flood filling progresses with equal speed in all directions, by using a *queue*. As a benefit, the storage is reduced drastically.

You will later (for instance in CS300) learn that the search algorithm using a stack is called "depth-first search", while the one using a queue is called "breadth-first search".

Queues

A queue supports three main operations:

- Putting an element at the end (rear) of the queue. We say we *enqueue* the element.
- Getting the element at the front of the queue.
- Removing the element at the front of the list. We say we *dequeue* the element.

The first element inserted into a queue is the first element that will come out of the queue, and so we say that a queue is a FIFO (first-in-first-out).

In the computing environment many things are queued. Like e-mails, GUIs, printer spool, and so on.