Flood fill

Every decent paint program offers a *flood fill* function. You select a color, then click on a location in your drawing, and the program automatically fills an entire region around the selected location. 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. A typical way to specify the color of a pixel is to use 8 bits each for the intensity of red, green, and blue light (so 24 bits in total per pixel). We use the `Image` class from the Python imaging library (module `PIL`), which represents colors as triples (r, g, b), where each component is an 8-bit integer in the range $0, \ldots, 255$.

To compute the region containing the selected location, 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, since this could cause the flood fill to "leak" through diagonal lines.

The first algorithm - using recursion

Flood filling is nearly the same as uncovering a region in MineSweeper: remember, when we uncovered a cell whose neighborhood contains no bomb, we could simply immediately uncover all neighboring cells.

In flood fill, we first have to remember the original color of the selected pixel. All pixels with the same color that can be reached through neighborhood relationships will be changed to the new color, all pixels with a different color act as a boundary.

This is the recursive flood fill function (from example code `flood-rec.py`):

```
def flood_rec(im, p, oldCol, newCol):
  im.putpixel(p, newCol)
  for q in neighbors(im, p):
    if im.getpixel(q) == oldCol:
      flood_rec(im, q, oldCol, newCol)
```

It's important that we change the color of pixel `p` *before* the recursive calls on the neighbor pixels. Can you see why?

In principle, this function works well, but it has a serious drawback: The depth of the recursion may be quite large (the best bound we have is the number of pixels in the region). Since Python's runtime stack is quite limited (by default, it can hold only 1000 stack frames), this function may easily cause a runtime stack overflow.

We can increase the size of the runtime stack, but that is not a good solution: the problem is that stack frames are quite large, and when you have a drawing with millions of pixels, we are simply using up far too much memory.

The second algorithm - using a stack

We discussed earlier how the runtime stack makes recursion possible, and concluded that every recursive algorithm can be converted to a non-recursive algorithm that uses a stack. Let's apply this idea to our problem.

Here is the flood fill function using a stack (from example code `flood-stack.py`):

```
def flood(im, p, color):
  oldCol = im.getpixel(p)
  if oldCol == color:
    return
  A = Stack()
```

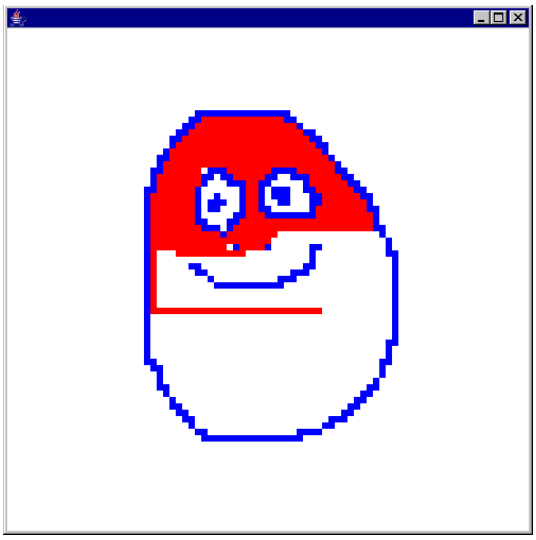

Figure 1: The first algorithm - using recursion

```
im.putpixel(p, color)
A.push(p)
while not A.is_empty():
  q = A.pop()
  for r in neighbors(im, q):
    if im.getpixel(r) == oldCol:
      im.putpixel(r, color)
      A.push(r)
```

What this algorithm does is really not different from the recursive algorithm—but instead of using the runtime stack, we use a stack implemented in Python. The advantage is that our own stack can be made as large as we want (if there is enough memory)—but, more importantly, pixel objects are *much* smaller than stack frames.

This function works well, and will not run out of memory even for large images. However, it may need quite a large stack. We can see this if we animate the algorithm. The program `animated-stack.py` does this: It colors the pixel green when it is pushed onto the stack, and colors it red when it is popped from the stack. There's also a small time delay after popping each pixel.

When you run the animation, you'll find that at some point many pixels will be colored green—and that means that at this time the stack is quite large. The green pixels cover a quite significant portion of the area to be colored.

The third algorithm - using a queue

Can we improve this? We'd like to spread the color through the region in a "nicer" way.

Thinking about it a little, we realize that for the correctness of the algorithm, it is actually not important that we use the stack data type. The algorithm works correctly no matter in which order we retrieve the stored pixels from the collection of remembered pixels. As long as every remembered pixel will eventually be considered, the algorithm will flood the entire reachable region.

The last-in-first-out behavior of the stack caused the flood filling to first proceed far into one direction, before covering pixels close to the starting location. Let's try the opposite: we use a data type that realizes *first-in-first-out* behavior. This data type is called a *queue*, and you can test the animation with the program `animated-queue.py`.

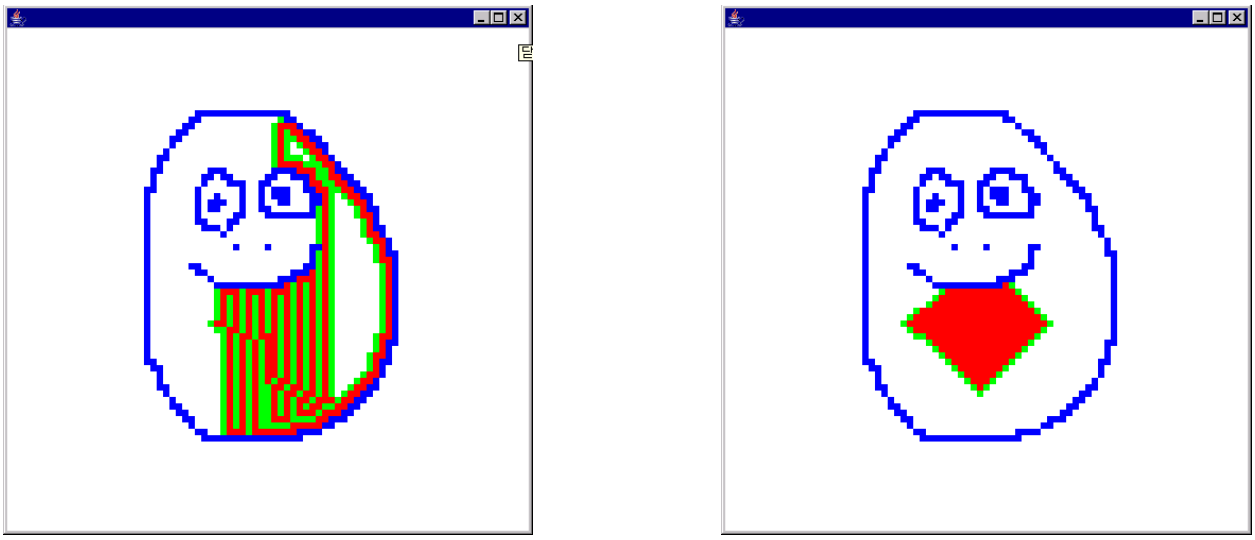

Figure 2: Using a stack, and using a queue

You will notice that using a queue the flood filling progresses with equal speed in all directions. You'll also notice that there are far fewer green pixels, so the amount of memory needed to remember pixel locations is smaller.

Here is the flood fill function using a queue (from example code `flood-queue.py`):

```
def flood(im, p, color):
  oldCol = im.getpixel(p)
  if oldCol == color:
    return
  A = Queue()
  im.putpixel(p, color)
  A.enqueue(p)
  while not A.is_empty():
    q = A.dequeue()
    for r in neighbors(im, q):
      if im.getpixel(r) == oldCol:
        im.putpixel(r, color)
        A.enqueue(r)
```

It's really completely identical to the stack version, except that the methods have different names: `push` becomes `enqueue`, `pop` becomes `dequeue`.

We will cover queues in more detail in the next lecture.

The flood filling problem can be considered as a graph exploration: the pixels are the nodes of the graph with the correct color, pixels are connected by an edge if they are neighbors. You will later (for instance in CS300) learn that the exploration algorithm using a stack is called "depth-first search," while the one using a queue is called "breadth-first search."