Producing p6 Images with OCaml

Produce a `P6 .ppm` image with the following method:

```
let dump_image img =
  let h = Array.length img in
  let w = Array.length img.(0) in
  let n = Array.length img.(0).(0) in
  Printf.printf "P6\n";
  Printf.printf "# %d\n" n;
  Printf.printf "%d %d\n" w h;
  Printf.printf "255\n";
  for y = 0 to pred h do
    let row = img.(y) in
    for x = 0 to pred w do
      let cl = row.(x) in
      print_char (char_of_int cl.(0));
      print_char (char_of_int cl.(1));
      print_char (char_of_int cl.(2));
    done;
  done;
;;
```

The second line is a comment indicating the color-depth.

The third line indicates the width and the height of the image.

This code assumes to recieve a regular matrix, and doesn't check if it doesn't.

The module will assume that the matrix image was produced properly by another method of the same module, so in this use case we don't really need to make these checks.

The new image can be created by this following function:

```
let new_img (_w, _h) (r, g, b) () =
  let img =
    Array.init _h (fun y ->
      Array.init _w (fun x ->
        Array.copy [| r; g; b |]
      )
    )
  in
  (img)
;;
```

If the image is initialised by this method, there is no risk to get an image that is not formed correctly.

A simple `fill_rect` method could be writen like this in a simple way:

```
let fill_rect img (_x, _y) (_w, _h) (r, g, b) =
  let h = Array.length img in
  let w = Array.length img.(0) in
  begin
    for y = _y to pred (_y + _h) do
      for x = _x to pred (_x + _w) do
        img.(y).(x).(0) <- r;
        img.(y).(x).(1) <- g;
        img.(y).(x).(2) <- b;
      done;
```

```
    done;
  end;
;;
```

But in this function there is no bound check if the upper-left corner of the rectangle is outside of the image. And also if the right-side, or the bottom will also be out-side of the image.

Here bellow we try to check if the rectangle is outside of the image, and if the rectangle is partially out-side.

```
let fill_rect img (_x, _y) (_w, _h) (r, g, b) =
  let h = Array.length img in
  let w = Array.length img.(0) in
  let n = Array.length img.(0).(0) in
  if _x + _w <= 0 then () else
  if _y + _h <= 0 then () else
  let _x, _w = if _x < 0 then 0, _w + (_x) else _x, _w in
  let _y, _h = if _y < 0 then 0, _h + (_y) else _y, _h in
  if _x >= _w then () else
  if _y >= _h then () else
  if n <> 3 then () else
  begin
    for y = _y to pred (_y + _h) do
      for x = _x to pred (_x + _w) do
        img.(y).(x).(0) <- r;
        img.(y).(x).(1) <- g;
        img.(y).(x).(2) <- b;
      done;
    done;
  end;
;;
```

In the check bellow, the first line verify if the rectangle is out-side of the image, on the left.
And the second line checks if the rectangle is above the area of the image.

```
  if _x + _w <= 0 then () else
  if _y + _h <= 0 then () else
```

This check adjusts the coordinates in case the rectangle is partially in-side and out-side of the area of the image:

```
  let _x, _w = if _x < 0 then 0, _w + (_x) else _x, _w in
  let _y, _h = if _y < 0 then 0, _h + (_y) else _y, _h in
```

This part `_h + (_y)` should be in fact mathematically `_h - (- _y)`, but here we avoid one extra-un-needed op.

This check verifies if the rectangle is on the right or under:

```
  if _x >= w then () else
  if _y >= h then () else
```

This check adjusts the width and the height of the rectangle if it overlaps on the right or at the bottom:

```
  let _w = if (_x + _w) > w then _w - ((_x + _w) - w) else _w in
  let _h = if (_y + _h) > h then _h - ((_y + _h) - h) else _h in
```

Here is below how to produce this image:

```
let () =
  let w, h = (120, 80) in
  let img = new_img (w, h) (0, 0, 0) () in

  fill_rect img (  0,  0) (80, 40) (255,   0,   0);
  fill_rect img (-60,  0) (80, 40) (255, 255,   0);
  fill_rect img ( 20, 40) (40, 20) ( 20,  80, 255);
  fill_rect img ( 40, 20) (60, 30) (  0, 255,   0);

  dump_image img;
;;
```

Initializing full white or black pictures could also be achieved with these `new_img_0` and `new_img_1` methods:

```
let new_img_0 (_w, _h) () =
  let img =
    Array.init _h (fun y ->
      Array.init _w (fun x ->
        Array.init 3 (fun c -> 0)
      )
    )
  in
  (img)
;;
```

```
let new_img_1 (_w, _h) () =
  let img =
    Array.init _h (fun y ->
      Array.init _w (fun x ->
        Array.init 3 (fun c -> 255)
      )
    )
  in
  (img)
;;
```

(filling with `0` and `1`.)

Now drawing a circle:

```
let draw_circ img (_x, _y) (_r) (r, g, b) =
  let _w = Array.length img.(0) in
  let _h = Array.length img in
  for i = (_x - _r) to (_x + _r) do
    for j = (_y - _r) to (_y + _r) do
      if i < 0 then () else
      if j < 0 then () else
      if i >= _w then () else
      if j >= _h then () else
      let dx = i - _x in
      let dy = j - _y in
      let sq_dist = (dx * dx) + (dy * dy) in
      let sq_r = (_r * _r) in
      if (sq_dist < sq_r) then
```

```
      begin
        let x, y = i, j in
        img.(y).(x).(0) <- r;
        img.(y).(x).(1) <- g;
        img.(y).(x).(2) <- b;
      end
    done
  done
```

In order to garanty that the image matrix is properly formed, we can define a `type`, so that the user will have to create the image with the functions provided by the module.

```
type t = int array array array
```

And we hide its implementation in the interface:

```
type t
```

Here are the files of this p6 module:

- p6.ml
- p6.mli

```
$ ocamlc -c p6.mli
$ ocamlc -c p6.ml
```