Functional Raster Image Synthesis*

Topics

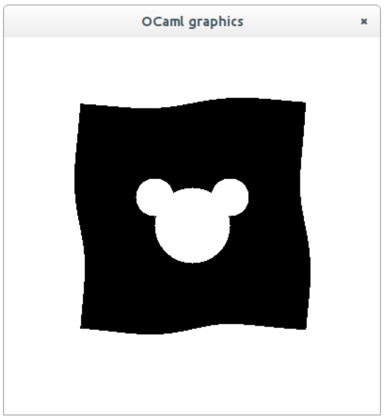
- Functions
- Pattern matching
- Numerical types and operations
- The `Graphics` module (code reading only)

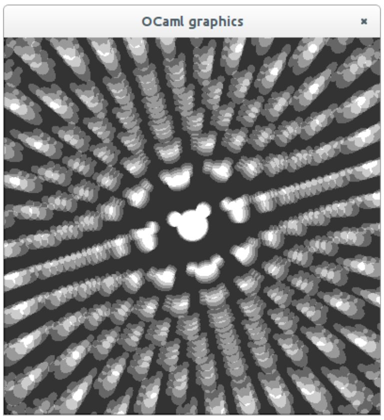
Exercise 1 – Raster Rendering using Graphics

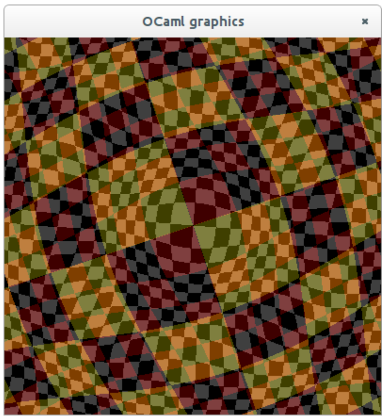
In this exercise, we will define color pictures as OCaml functions mapping points of the plane to colors. These functions will have type:

```
(float * float) -> (float * float * float * float)
```

The two tupled parameters are input coordinates x and y. The result is a quadruplet `(r, g, b, a)` giving the color at this point as *premultipled* RGBA components: the alpha component is in the interval $[0., 1.]$ and the color components are in the interval $[0., a.]$. Black is $(0., 0., 0., 1.)$, transparent is $(0., 0., 0., 0.)$, red $(1., 0., 0., 1.)$, and semi-transparent red is $(0.5, 0., 0., 0.5)$.







Question 1.1 – To render the image, we use the following function, written in imperative style using the module `Graphics`. Read it, deduce its type, and which part of the plane is rendered.

```
let render w h f =
  let dx = 2. /. float w and dy = 2. /. float h in
  let open Graphics in
  open_graph (Printf.sprintf " %dx%d" w h) ;
  auto_synchronize false ;
  for x = 0 to w - 1 do
    for y = 0 to h - 1 do
      let px = float x *. dx -. 1.
      and py = float y *. dy -. 1. in
      let (r, g, b, a) = f (px, py) in
      if a <> 0. then begin
        set_color @@ rgb
          (int_of_float ((r +. 1. -. a) *. 255.))
          (int_of_float ((g +. 1. -. a) *. 255.))
          (int_of_float ((b +. 1. -. a) *. 255.)) ;
        plot x y
      end
    done
  done ;
  synchronize () ;
  wait_next_event [ Button_down ; Key_pressed ] |> ignore ;
  close_graph ()
```

Question 1.2 – Open a file `funpics.ml` and write down this code.

Question 1.3 – Define a value `transparent` to be returned by image functions at points where they are not defined, and some color values `black`, `red`, etc.

Question 1.4 – Write a predicate `valid_color` that is valid when the four components of a color quadruplet fit in their respective range.

Question 1.5 – Write a function `clamp` that takes a color quadruplet and returns the same quadruplet when it represents a valid color. Otherwise, it should return a quadruplet where out of bound components have been replaced by their nearest bound.

Exercise 2 – Simple Shapes

Question 2.1 – Write a function background that takes a color c and produces an image that fills the plane with c.

Question 2.2 – Write a simple test `let () = render (background red)`, compile and run the program.

Question 2.3 – Write a function `disk` that takes a float r a color c and produces a c-colored disk of radius r centered at $(0., 0.)$.

Question 2.4 – Write a function square that takes a float s a color c and produces an c-colored square of side s centered at $(0., 0.)$.

Exercise 3 – Simple Transformations

Question 3.1 – Write a function `at` that takes two floats x and y, a picture f and produces a version of f centered at (x, y) (translated by $(-x, -y)$).

Question 3.2 – Write a function `scale` that takes a float s, a picture f and produces a version of f scaled by a factor s.

Question 3.3 – Write a function `rot` that takes a float a, a picture f and produces a version of f turned of a radians.

Question 3.4 – You can also write a function `rotozoom` instead that takes two floats a and s and performs both transformations.

Exercise 4 – Funnier Transformations

Question 4.1 – Write a function `repeat` that takes a float s, a picture f and produces an infinite repetition of the $(-s, -s)$ – (s, s) part of f.

Question 4.2 – Write a function `waves` that takes a float wl and transforms the input plane of f, adding to the radial component ρ a sinusoidal offset, function of ρ of period wl, creating a ripple effect.

Many other fun effects can be achieved by switching to polar coordinates. You can for instance create a nice effect by adding to ρ a sinusoidal offset function of α. Try writing `warp`, taking ph the phase of the effect, n the number of periods and amp the amplitude of the offset.

Exercise 5 – Grey image

In this exercise, we introduce two new kinds of images: black and white images, that are represented by a function of type `(float * float) -> bool`—black being `false`—, and grey-scale images, that are represented by a function of type `(float * float) -> float`—black being `0.` and white being `1.`.

Question 5.1 – Write a function `image_of_bw_image` that takes a BW image and returns an equivalent image as a function that may be passed to the function `render`.

Question 5.2 – Write a function `image_of_grey_image`.

Question 5.3 – Write a function `mask`, that takes an image `src` and returns a grey image corresponding to the alpha component of `src`.

Question 5.4 – Write a function `alpha`, that takes an image `src` and a grey image `mask` and returns an image corresponding to `src` where the alpha component has been multiplied by `mask`.

Exercise 6 – Alpha composition

Let's write a series of binary operators to build composite shapes using simple ones.

Question 6.1 (Compositing operators) – Write a function `compose_src_over` that takes two image `src` and `dest` and returns a image where `src` is composited over `dest`. The *composited* image may be defined with the following formula, where Ca_{src} and Ca_{dest} represent respectively one of the premultiplied RGB composant of `src` and `dest`, and a_{src} and a_{dest} represent their alpha composant:

- $Ca_{res} = Ca_{src} + Ca_{dest}(1 - a_{dest})$
- $a_{res} = a_{src} + a_{dest} - a_{src}a_{dest}$

You may also write any compositing operators found in the SVG specification[1], for instance the *source intersection* that could be defined by the following formula:

- $Ca_{res} = Ca_{src} + a_{dest}$
- $a_{res} = a_{src}a_{dest}$

Question 6.2 – Test all the compositing operators you defined by displaying sequentially the result of their application on the same source and destination images. *Hint: you may iterate a function over the list of all compositing functions.*

Question 6.3 – Write a function `combine_right` taking a compositing operator o and a list of images l and returning the composition with o of all the images of l, starting with the last two images. In other words, for some point (x, y), `combine_right op [ f1 ; f2 ; ... ; fn ]` is color `(op f1 (op f2 (... (op fn-1 fn)))`.

Exercise 7 – A bit of action

Question 7.1 – Write an `animate` derivative of the `render` function. Instead of taking an image, in now takes a function of type `float -> image`, to which it provides the rendering time. Use functions `Unix.gettimeofday` and `Graphics.clear_graph` to do so.

Question 7.2 – Improve `animate` so that it also takes the mouse coordinates as parameters. Read the section about events of the `Graphics` documentation.

[1] http://www.w3.org/TR/2009/WD-SVGCompositing-20090430/#containerElementCompositingOperators

Functional Raster Image Synthesis

SOLUTIONS – SOLUTIONS – SOLUTIONS – SOLUTIONS

Solution to question 1.3

```
let transparent = (0., 0., 0., 0.)
let black = (0., 0., 0., 1.)
let white = (1., 1., 1., 1.)
let red = (1., 0., 0., 1.)
let green = (0., 1., 0., 1.)
let blue = (0., 1., 0., 1.)
let yellow = (1., 1., 0., 1.)
let violet = (1., 0., 1., 1.)
let cyan = (0., 1., 1., 1.)
```

Solution to question 1.4

```
let valid_color (ra, ga, ba, a) =
  0. <= a && a <= 1. &&
  0. <= ra && ra <= a &&
  0. <= ga && ga <= a &&
  0. <= ba && ba <= a
```

Solution to question 1.5

```
let clamp ra ga ba a =
  let a = max 0. (min a 1.) in
  let ra = max 0. (min ra a) in
  let ga = max 0. (min ga a) in
  let ba = max 0. (min ba a) in
  (ra, ga, ba, a)

let rbg r g b = (r, g, b, 1.)
let rgba r g b a = (r *. a, g *. a, b *. a, a)
```

Solution to question 2.1

```
let background color = fun (x, y) -> color
```

Alternatively:

```
let background color (x, y) = color
```

Solution to question 2.2

```
ocamlopt graphics.cmxa funpics.ml -o funpics
./funpics
```

Solution to question 2.3

```
let disk radius color = fun (x, y) ->
  if x *. x +. y *. y < radius *. radius then color else transparent
```

Solution to exercise 2

```
let square side color =
  let radius = side /. 2. in
  fun (x, y) ->
    if abs_float x < radius && abs_float y < radius
    then color else transparent
```

Solution to question 3.1

```
let at sx sy f = fun (x, y) ->
  f (x -. sx, y -. sy)
```

Solution to question 3.2

```
let scale s f = fun (x, y) ->
  f (x /. s, y /. s)
```

Solution to question 3.3

```
let rotate da f = fun (x, y) ->
  let a = atan2 y x -. da in
  let p = sqrt (x *. x +. y *. y) in
  f (p *. cos a, p *. sin a)
```

Solution to question 3.4

```
let rotozoom da z f = fun (x, y) ->
  let a = atan2 y x in
  let p = sqrt (x *. x +. y *. y) in
  let a = a -. da in
  let p = p /. z in
  f (p *. cos a, p *. sin a)
```

Solution to question 4.1

```
let repeat w h f = fun (x, y) ->
  let mod_float x m =
    let r = mod_float x m in
    if r >= 0. then r else m +. r in
  f (mod_float (x +. w /. 2.) w -. w /. 2.,
     mod_float (y +. h /. 2.) h -. h /. 2.)
```

Solution to question 4.2

```
let waves wl f = fun (x, y) ->
  let wl = wl /. 6.28 in
  let a = atan2 y x in
  let p = sqrt (x *. x +. y *. y) in
  let p = p -. sin (p /. wl) *. 0.5 *. wl in
  f (p *. cos a, p *. sin a)

let warp ph n amp f = fun (x, y) ->
  let a = atan2 y x in
  let p = sqrt (x *. x +. y *. y) in
  let p = p +. sin (ph +. a *. float n) *. amp in
  f (p *. cos a, p *. sin a)
```

Solution to question 5.1

```
let image_of_gray_image f = fun (x, y) ->
  if f (x, y) then black else transparent
```

Solution to question 5.2

```
let image_of_gray_image f = fun (x, y) ->
  let lvl = f (x, y) in
  (lvl, lvl, lvl, 1.)
```

Solution to question 5.3

```
let mask f = fun (x, y) ->
  let (_, _, _, a) = f (x, y) in a
```

Solution to question 5.4

```
let alpha src mask = fun (x, y) ->
  let (r, g, b, a) = src (x, y) in
  let a' = mask (x, y) in
  (r, g, b, a *. a')
```

Solution to question 6.1

```
let compose_src_over src dest = fun (x, y) ->
  let (ra_src, ga_src, ba_src, a_src as c) = src (x, y) in
  if a_src >= 1. then c else
  let (ra_dest, ga_dest, ba_dest, a_dest) = dest (x, y) in
  let f ca_src ca_dest =
    ca_src +. ca_dest *. (1. -. a_src) in
  (f ra_src ra_dest, f ga_src ga_dest, f ba_src ba_dest,
   a_src +. a_dest -. a_src *. a_dest)

let compose_src_in src dest = fun (x, y) ->
  let (ra_dest, ga_dest, ba_dest, a_dest) = dest (x, y) in
  if a_dest <= 0. then
    transparent
  else
    let (ra_src, ga_src, ba_src, a_src) = src (x, y) in
    let f ca_src ca_dest = ca_src *. a_dest in
    (f ra_src ra_dest, f ga_src ga_dest, f ba_src ba_dest,
     a_src *. a_dest)
```

Solution to question 6.2

```
let all_compositions =
  [ compose_src_over;
    compose_src_in;
  ]

let sq_src = square yellow 1.33 |> at ~-.0.33 (-.0.33)
let sq_dest = square blue 1.33 |> at 0.33 0.33

let () =
  List.iter
    (fun f -> render (f sq_src sq_dest))
    all_compositions
```

Solution to question 6.3

```
let rec combine_right binop fs =
  match fs with
  | [] -> plane transparent
  | [f] -> f
  | f :: fs -> binop f (combine_right binop fs)
let combine_left binop fs = combine_right binop (List.rev fs)
```

Solution to exercise 7

```
let animate (f : float -> point -> image) : unit =
  let dx = 2. /. float w and dy = 2. /. float h in
  let open Graphics in
  open_graph (Printf.sprintf " %dx%d" w h) ;
  auto_synchronize false ;
  let tzero = Unix.gettimeofday () in
  let rec loop mx my =
    clear_graph () ;
    for x = 0 to w - 1 do
      for y = 0 to h - 1 do
        let px = float x *. dx -. 1.
        and py = float y *. dy -. 1. in
        match f (Unix.gettimeofday () -. tzero) (mx, my) (px, py) with
        | None -> ()
        | Some (r, g, b) ->
          set_color @@ rgb
            (int_of_float (r *. 255.))
            (int_of_float (g *. 255.))
            (int_of_float (b *. 255.)) ;
          plot x y
      done
    done ;
    synchronize () ;
    let st = wait_next_event [ Button_down ; Key_pressed ;
                                Mouse_motion ; Poll ] in
    if not (st.button || st.keypressed) then
      let mx = float (st.mouse_x - w / 2) *. dx
      and my = float (st.mouse_y - h / 2) *. dy in
      loop mx my
    else
      close_graph ()
  in loop 0. 0.
```

Solution

```
(*-- Simple Examples --*)

let () =
  render
    (rem
       (square 0.6 |> at (0., 0.05)
        |> warp 0. 4 0.03)
       (combine join transparent
          [ disk 0.1 |> at (-0.2, 0.15) ;
```

```
            disk 0.2 |> at (0., 0.) ;
            disk 0.1 |> at (0.2, 0.15) ])
     |> fill black)

let () =
  animate @@ fun t (mx, my) ->
  exclude
    (square 0.6 |> at (0., 0.05)
     |> rotozoom 0. (1. +. sin t *. 0.5)
     |> warp t 4 0.03)
    (rem
       (combine join transparent
          [ disk 0.2 |> at (-0.4, 0.3) ;
            disk 0.4 |> at (0., 0.) ;
            disk 0.2 |> at (0.4, 0.3) ])
       (combine join transparent
          [ disk 0.05 |> at (-0.2, 0.) ;
            disk 0.05 |> at (0.2, 0.) ;
            disk 0.1 |> at (0., -0.2) ])
     |> rotozoom 0. (1. -. sin t *. 0.5))
  |> fill red
  |> at (-. mx /. 3., -. my /. 3.)
```

Solution

```
(*-- Complex Examples --*)

let () =
  let pattern =
    combine join transparent
      [ square 0.04 |> at (-0.04, -0.04) ;
        square 0.04 |> at (0.04, +0.04) ]
    |> repeat 0.08 0.08
    |> waves 0.4 in
  animate @@ fun t (mx, my) ->
  let t = t /. 10. in
  blend
    [ square 1. |> fill black ;
      pattern |> fill red |> rotozoom (0.4 -. t) 4. ;
      pattern |> fill yellow |> rotozoom (0.8 +. t /. 2.) 8. ;
      pattern |> fill white |> rotozoom t 2. ] |>
  let ma = atan2 my mx in
  rotozoom ma (max 0.2 (abs_float mx ** 2.))

let () =
  let pattern =
    waves 0.5
      (repeat 0.08 0.08
         (combine join transparent
            [ disk 0.01 |> at (-0.02, 0.015) ;
              disk 0.02 |> at (0., 0.) ;
              disk 0.01 |> at (0.02, 0.015) ]))
    |> fill white in
  render
```

```
    (blend
       [ square 1. |> fill white ;
         pattern |> rotozoom 0.3 2.0 ;
         pattern |> rotozoom 0.3 2.1 ;
         pattern |> rotozoom 0.3 2.2 ;
         pattern |> rotozoom 0.3 2.3 ])
```