COMP10001 Foundations of Computing

Advanced Lecture
Functional Graphics

Traditional representation of images

As in project 2, the traditional way to represent a computer image is using a rectangular grid of pixels.

```
image = [[(15, 103, 255), (0, 3, 19)],
         [(22, 200, 1), (8, 8, 8)],
         [(0, 0, 0), (5, 123, 19)]]
```

Traditional representation of images

Each pixel is accessed by its row and column coordinates, which are integers.

```
some_pixel = image[2][1]
```

This representation reflects the way that images are stored in files, but it is not necessarily the most convenient way to work with images.

Abstraction to the rescue!

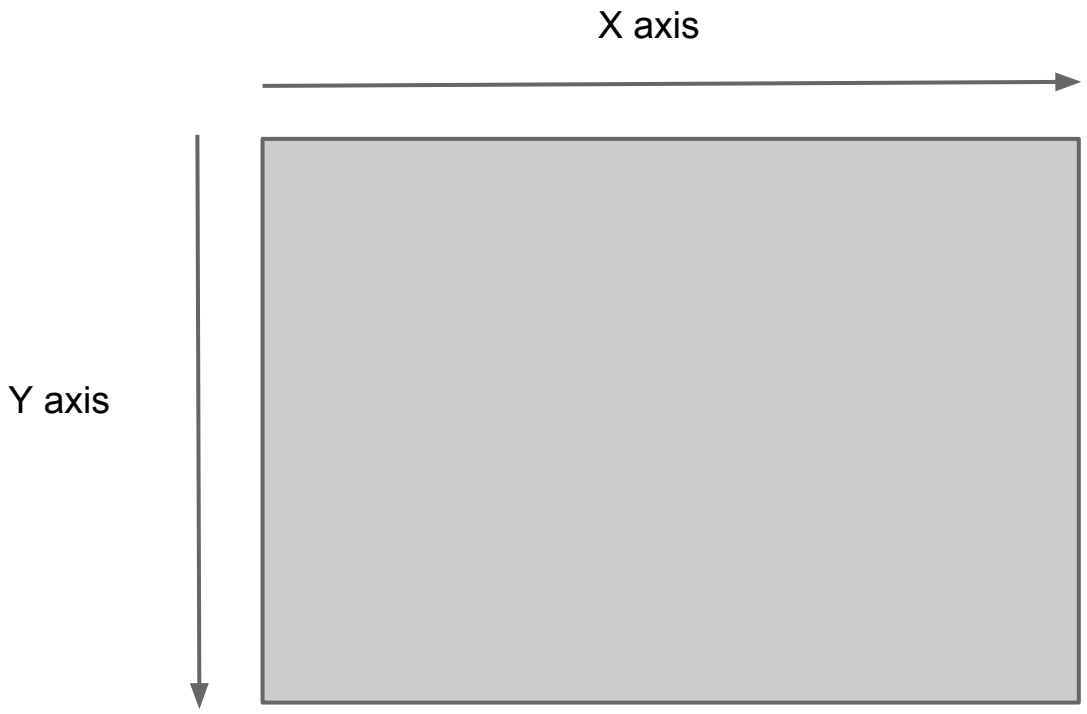
A more abstract way to represent an image is as a function from coordinates to pixel values:

pixel: (int, int, int)

image: (float, float) ⇒ pixel

Note that we allow pixel values at arbitrary floating point positions, rather than just integer coordinates.

Our coordinate system



Abstraction to the rescue!

Here is an infinite red image:

```
red_pixel = (255, 0, 0)

def red_image(x, y):

    return red_pixel
```

This function returns a red pixel for any (x,y) coordinate.

Turning functions into bitmaps

```
def reify(image_fun, row1, col1, row2, col2):
    image = []
    for y in range(row1, row2 + 1):
        image_row = []
        for x in range(col1, col2 + 1):
            image_row.append(image_fun(x, y))
        image.append(image_row)
    return image
```

Saving images

```
from SimpleImage import write_image

def save(image_fun, row1=0, col1=0, row2=299,
         col2=299, filename='out.png'):

    image = reify(image_fun, row1, col1,
                  row2, col2)
    write_image(image, filename)
```

A 300 x 300 red image

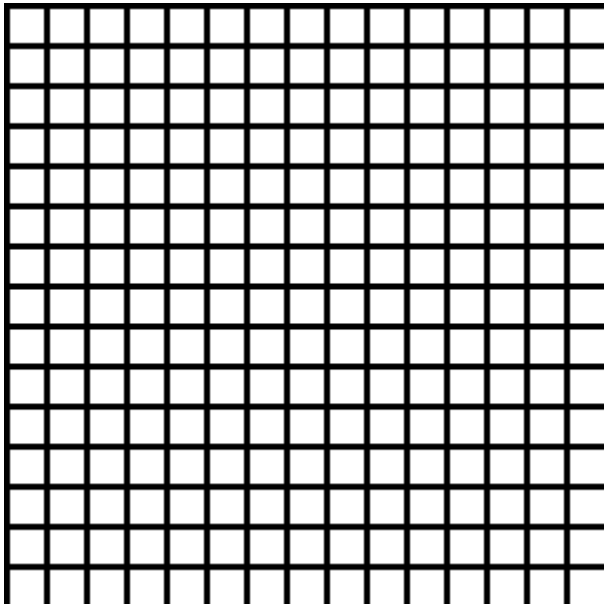
```
>>> save(red_image)
```

Drawing a grid

```
def grid(cell_size, line_thickness):
    def do_grid(x, y):
        if (x % cell_size) < line_thickness or \
            (y % cell_size) < line_thickness:
                return black_pixel
            else:
                return white_pixel
     return do_grid
```

Drawing a grid

```
>>> save(grid(20, 3))
```

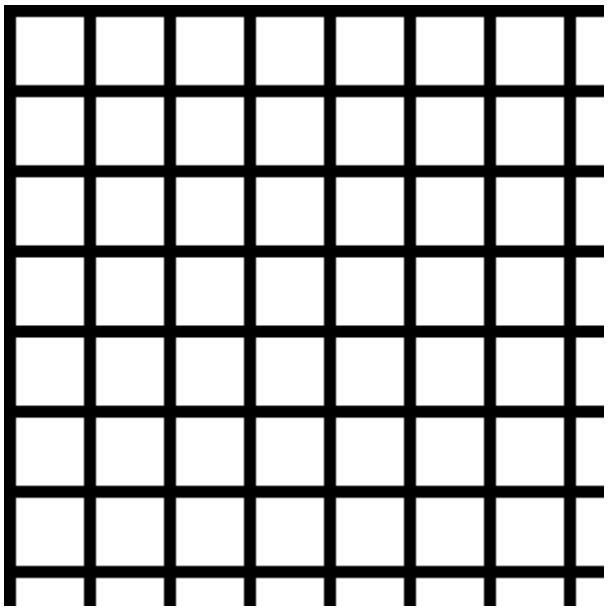


Scaling an image about a point

```
def scale(image_fun, factor, c_x, c_y):
    def do_scale(x, y):
        x_t = x - c_x
        y_t = y - c_y
        new_x = x_t / factor
        new_y = y_t / factor
        x_t = new_x + c_x
        y_t = new_y + c_y
        return image_fun(x_t, y_t)
    return do_scale
```

Scaling an image about a point

```
>>> grid_image = grid(20, 3)
>>> save(scale(grid_image, 2, 0, 0))
```



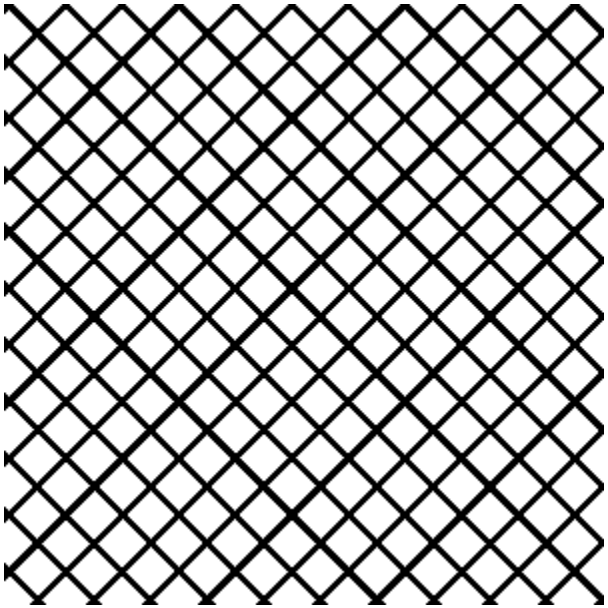
Rotating an image about a point

```
from math import cos, sin, radians

def rotate(f, angle, c_x, c_y):
    angle = radians(angle)
    def do_rotate(x, y):
        x_t = x - c_x
        y_t = y - c_y
        new_x = x_t * cos(angle) - y_t * sin(angle)
        new_y = x_t * sin(angle) + y_t * cos(angle)
        x_t = new_x + c_x
        y_t = new_y + c_y
        return f(x_t, y_t)
    return do_rotate
```

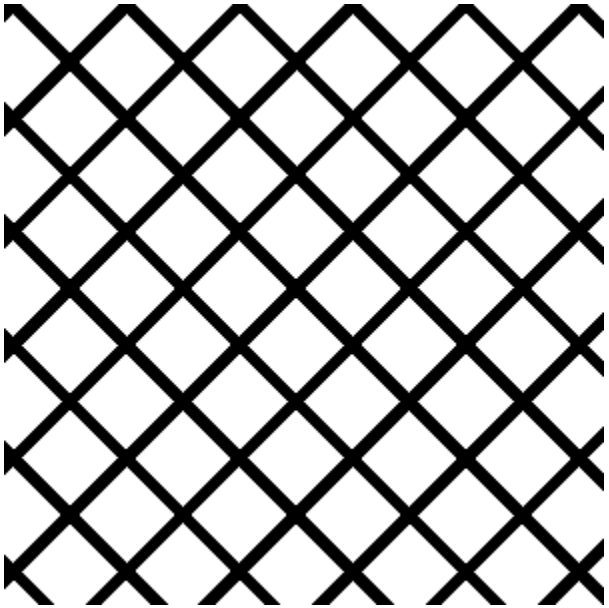
Rotating an image about a point

```
>>> grid_image = grid(20, 3)
>>> save(rotate(grid_image, 45, 0, 0))
```



Combining transformations

```
>>> i = grid(20, 3)
>>> i = scale(i, 2, 0, 0)
>>> i = rotate(i, 45, 0, 0)
>>> save(i)
```



Bitmaps to functions

```
def from_bitmap(bitmap, tiled=True):
    def do_from_bitmap(x, y):
        num_rows = len(bitmap)
        if num_rows > 0:
            num_cols = len(bitmap[0])
            if num_cols > 0:
                if tiled:
                    x %= num_cols
                    y %= num_rows
                x = int(round(x))
                y = int(round(y))
                if y > 0 and x > 0 and \
                   y < num_rows and \
                   x < num_cols:
                    return bitmap[y][x]
        return black_pixel
    return do_from_bitmap
```

Bitmaps to functions

```
>>> from SimpleImage import read_image
>>> b = read_image('floyd.png')
>>> save(from_bitmap(b))
```

Bitmaps to functions

```
>>> i = from_bitmap(read_image('floyd.png'))
>>> i = scale(i, 0.2, 0, 0)
>>> save(rotate(i, 45, 0, 0))
```

Distorting an image by a wave

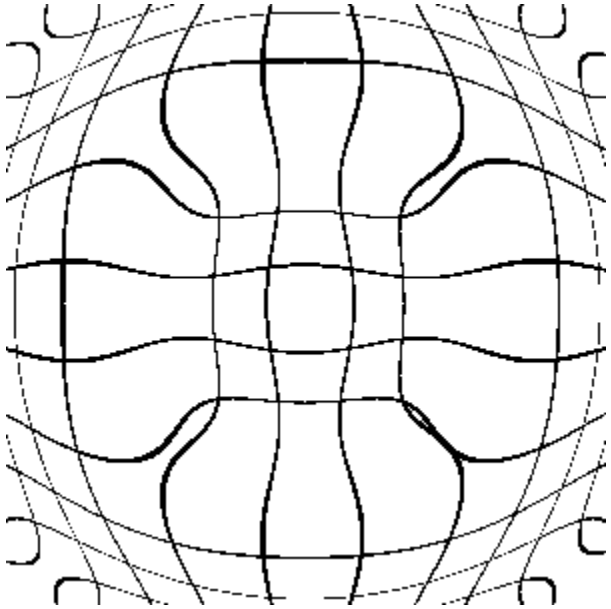
```
from math import sqrt

def distance(x1, y1, x2, y2):
    return sqrt((y1 - y2) ** 2 + (x1 - x2) ** 2)

def wave(image_fun, amp, period, c_x, c_y):
    def do_wave(x, y):
        d = distance(x, y, c_x, c_y)
        s = 2 + cos(radians(d * period))/2.0 * amp
        return scale(image_fun, s, c_x, c_y)(x, y)
    return do_wave
```

Distorting an image by a wave

```
>>> i = grid(20, 1)
>>> save(wave(i, 1, 3, 149, 149))
```



Distorting an image by a wave

```
>>> i = from_bitmap(read_image('floyd.png'))
>>> i = scale(i, 0.2, 0, 0)
>>> i = wave(i, 3, 1, 250, 250)
>>> save(i, col2 = 500, row2 = 500)
```

Distorting an image by a wave

Fuzzify

```
from random import randint

def random_neighbour(image_fun, window_size):
    half_window = window_size / 2
    def do_random_neighbour(x, y):
        new_x = x + randint(-half_window, half_window)
        new_y = y + randint(-half_window, half_window)
        return image_fun(new_x, new_y)
    return do_random_neighbour
```

Fuzzify

```
>>> i = from_bitmap(read_image('floyd.png'))
>>> save(random_neighbour(i, 10))
```

Challenge

Write a program to make images like this: