アルゴリズムとデータ構造入門 必修課題2

1029-24-9540 山崎啓太郎

February 7, 2013

1 painterを1種類作成

```
(define perorin (lambda (part)
        (cond
                ((eq? part 'frame) (vertexes->painter
                        (list
                                (make-vect .05 .1) (make-vect .95 .1)
                                (make-vect .95 .1) (make-vect .5 .85)
                                (make-vect .5 .85) (make-vect .05 .1))
                        #t ))
                ((eq? part 'left-eye) (vertexes->painter
                        (list
                                (make-vect .35 .5) (make-vect .4 .5)
                                (make-vect .4 .5) (make-vect .4 .55)
                                (make-vect .4 .55) (make-vect .35 .55)
                                (make-vect .35 .55) (make-vect .35 .5))
                        #t ))
                ((eq? part 'right-eye) (vertexes->painter
                        (list
                                (make-vect .65 .5) (make-vect .6 .5)
                                (make-vect .6 .5) (make-vect .6 .55)
                                (make-vect .6 .55) (make-vect .65 .55)
                                (make-vect .65 .55) (make-vect .65 .5))
                        #t ))
                ((eq? part 'mouth) (vertexes->painter
                        (list
                                (make-vect .2 .25) (make-vect .8 .25)
                                (make-vect .8 .25) (make-vect .8 .28)
                                (make-vect .8 .28) (make-vect .2 .28)
                                (make-vect .2 .28) (make-vect .2 .25))
                        #t ))
                ((eq? part 'tongue) (vertexes->painter
                        (list
                                (make-vect .6 .25) (make-vect .74 .25)
                                (make-vect .74 .25) (make-vect .67 .4)
                                (make-vect .67 .4) (make-vect .6 .25))
                        #t )) ) ))

(define draw-perorin (lambda (frm)
        (clear-picture)
        (set-color #x0B6A4A)
        ((perorin 'frame) frm)
        (set-color #x0ffffff)
        ((perorin 'left-eye) frm)
        ((perorin 'right-eye) frm)
        ((perorin 'mouth) frm)
        (set-color #xCB361F)
        ((perorin 'tongue) frm)
))

(define draw-perorin-square-limit (lambda (n frm)
        (clear-picture)
        (set-color #x0B6A4A)
        ((square-limit (perorin 'frame) n) frm)
        (set-color #x0ffffff)
        ((square-limit (perorin 'left-eye) n) frm)
        ((square-limit (perorin 'right-eye) n) frm)
        ((square-limit (perorin 'mouth) n) frm)
        (set-color #xCB361F)
        ((square-limit (perorin 'tongue) n) frm)
))
```

2 square-limit に適用

```
(define draw-perorin-square-limit (lambda (n frm)
        (clear-picture)
        (set-color #x0B6A4A)
        ((square-limit (perorin 'frame) n) frm)
        (set-color #x0ffffff)
        ((square-limit (perorin 'left-eye) n) frm)
        ((square-limit (perorin 'right-eye) n) frm)
        ((square-limit (perorin 'mouth) n) frm)
        (set-color #xCB361F)
        ((square-limit (perorin 'tongue) n) frm)
))

(load "init.lsp")
(load "painter.scm")
(load "square-limit.scm")
(draw-perorin-square-limit 4 frm1)
```

画像はperorin.pngに保存してあります。

3　空間充填曲線を1種類作成

```
(define hilbert-a (lambda (x0 y0 x1 y1 i)
        (let ((xs (/ (+ (* 3.0 x0) x1) 4.0))
                (ys (/ (+ (* 3.0 y0) y1) 4.0))
                (xm (/ (+ x0 x1) 2.0))
                (ym (/ (+ y0 y1) 2.0))
                (xl (/ (+ x0 (* 3.0 x1)) 4.0))
                (yl (/ (+ y0 (* 3.0 y1)) 4.0)) )

                (if (= i 0)
                        (list (make-vect xl yl) (make-vect xs yl)
                                (make-vect xs ys) (make-vect xl ys))
                        (append (hilbert-d xm ym x1 y1 (- i 1))
                                (hilbert-a x0 ym xm y1 (- i 1))
                                (hilbert-a x0 y0 xm ym (- i 1))
                                (hilbert-b xm y0 x1 ym (- i 1))) ))))
(define hilbert-b (lambda (x0 y0 x1 y1 i)
        (let ((xs (/ (+ (* 3.0 x0) x1) 4.0))
                (ys (/ (+ (* 3.0 y0) y1) 4.0))
                (xm (/ (+ x0 x1) 2.0))
                (ym (/ (+ y0 y1) 2.0))
                (xl (/ (+ x0 (* 3.0 x1)) 4.0))
                (yl (/ (+ y0 (* 3.0 y1)) 4.0)) )

                (if (= i 0)
                        (list (make-vect xs ys) (make-vect xs yl)
                                (make-vect xl yl) (make-vect xl ys) )
                        (append (hilbert-c x0 y0 xm ym (- i 1))
                                (hilbert-b x0 ym xm y1 (- i 1))
                                (hilbert-b xm ym x1 y1 (- i 1))
                                (hilbert-a xm y0 x1 ym (- i 1))) ))))
(define hilbert-c (lambda (x0 y0 x1 y1 i)
        (let ((xs (/ (+ (* 3.0 x0) x1) 4.0))
                (ys (/ (+ (* 3.0 y0) y1) 4.0))
                (xm (/ (+ x0 x1) 2.0))
                (ym (/ (+ y0 y1) 2.0))
                (xl (/ (+ x0 (* 3.0 x1)) 4.0))
                (yl (/ (+ y0 (* 3.0 y1)) 4.0)) )

                (if (= i 0)
                        (list (make-vect xs ys)(make-vect xl ys)
                                 (make-vect xl yl) (make-vect xs yl) )
                        (append (hilbert-b x0 y0 xm ym (- i 1))
                                (hilbert-c xm y0 x1 ym (- i 1))
                                (hilbert-c xm ym x1 y1 (- i 1))
                                (hilbert-d x0 ym xm y1 (- i 1))) ))))
(define hilbert-d (lambda (x0 y0 x1 y1 i)
        (let ((xs (/ (+ (* 3.0 x0) x1) 4.0))
                (ys (/ (+ (* 3.0 y0) y1) 4.0))
                (xm (/ (+ x0 x1) 2.0))
                (ym (/ (+ y0 y1) 2.0))
                (xl (/ (+ x0 (* 3.0 x1)) 4.0))
                (yl (/ (+ y0 (* 3.0 y1)) 4.0)) )

                (if (= i 0)
                        (list (make-vect xl yl) (make-vect xl ys)
                                (make-vect xs ys) (make-vect xs yl))
                        (append (hilbert-a xm ym x1 y1 (- i 1))
                                (hilbert-d xm y0 x1 ym (- i 1))
                                (hilbert-d x0 y0 xm ym (- i 1))
                                (hilbert-c x0 ym xm y1 (- i 1))) ))))

(define vectors->segments (lambda (vectors)
        (define vector->segment (lambda (vector-list)
                (define a (car vector-list))
                (define b (cadr vector-list))
                (if (eq? nil b)
                        '()
                        (append (list (make-segment
                                (make-vect (car a) (cdr a))
                                (make-vect (car b) (cdr b))))
                                (vector->segment (cdr vector-list))) )))
        (vector->segment vectors) ))

(define hilbert (lambda (n)
        (segments->painter
                (vectors->segments (hilbert-a 0.0 0.0 1.0 1.0 n))) ))

; 以下は使用例です
(load "init.lsp")
(clear-picture)
((hilbert 4) frm1)
```

Hilbert circleを作成しました。
画像はhilbert.pngにあります。

4　フラクタルを１種類作成

```
(define kock-line (lambda (p0 q0 p1 q1 r i)
        (if (= i 0)
                (list (make-vect p0 q0) (make-vect p1 q1))
                (let* ((r1 (/ r 3.0))
                        (x3 (/ (- p1 p0) 3.0))
                        (y3 (/ (- q1 q0) 3.0))
                        (xs (/ (+ (* 2.0 p0) p1) 3.0))
                        (ys (/ (+ (* 2.0 q0) q1) 3.0))
                        (xl (/ (+ p0 (* 2.0 p1)) 3.0))
                        (yl (/ (+ q0 (* 2.0 q1)) 3.0))
                        (xm (+ (* 0.5 x3) (* 0.866 y3) xs))
                        (ym (+ (* 0.5 y3) (* -0.866 x3) ys)) )
                        (append (kock-line p0 q0 xs ys r1 (- i 1))
                                (kock-line xs ys xm ym r1 (- i 1))
                                (kock-line xm ym xl yl r1 (- i 1))
                                (kock-line xl yl p1 q1 r1 (- i 1)) )))))

(define vectors->segments (lambda (vectors)
        (define vector->segment (lambda (vector-list)
                (define a (car vector-list))
                (define b (cadr vector-list))
                (if (eq? nil b)
                        '()
                        (append (list (make-segment
                                (make-vect (car a) (cdr a))
                                (make-vect (car b) (cdr b))))
                                (vector->segment (cdr vector-list))) )))
        (vector->segment vectors) ))

(define kock (lambda (n)
        (let* ((h (/ 0.75 0.86))
                (p0 (/ (- 1.0 h) 2))
                (p1 (- 1.0 p0)) )
                (segments->painter
                        (vectors->segments
                                (append
                                        (kock-line p0 0.25 p1 0.25 1 n)
                                        (kock-line p1 0.25 0.5 1.0 1 n)
                                        (kock-line 0.5 1.0 p0 0.25 1 n) )))

; 以下は使用例
(load "init.lsp")
((kock 3) frm1)
```

Kock Snowflake を作成しました。
画像は snowflake.png にあります。