Engine

James Long

March 7, 2009

First we load up the required modules. We will be testing the main engine module.

```
(load "lib/engine")
(init-engine)
```

1 Images & Textures

1.1 image type

An image type is defined with the following fields:

- width
- height
- rgb-bytes
- gl-texture-id

1.2 (make-image filename)

Loads an image into memory, returning an image instance. FreeImage is used to load images. Only JPG support has been implemented.

```
(define test-image #f)

(define-test make-image
  (set! test-image (make-image "docs/test.jpg"))
  (assert-equal (image-width test-image) 100)
  (assert-equal (image-height test-image) 100)
  (assert-equal (char->integer
                 (u8*-ref (image-rgb-bytes test-image) 0)) 25)
  (assert-equal (char->integer
                 (u8*-ref (image-rgb-bytes test-image) 1)) 51)
  (assert-equal (char->integer
                 (u8*-ref (image-rgb-bytes test-image) 2)) 76))
```

1.3 (image-opengl-upload! image)

Creates an OpenGL texture and uploads the image data to video memory. Various OpenGL texture states are set to appropriate defaults. This function will set the image's gl-texture-id field to the generated OpenGL texture id.

TODO: need to implement testing under OpenGL.

1.4 (image-fetch-rgb image x y)

Retrieves the RGB color from the source image at the specified position. The bottom-left corner defines (0,0).

```
(define-test image-fetch-rgb
  (let ((color (image-fetch-rgb test-image 55 59)))
    (assert-equal (vec3-x color) 239)
    (assert-equal (vec3-y color) 255)
    (assert-equal (vec3-z color) 255)))
```

2 Selection Procedures

2.1 (selection-rws pop f)

Implements roulette wheel selection. Given a number f, where f is between 0 and the sum of all fitnesses, linearly search the genomes and find the genome occupying the specified slot, where slot lengths are the genome's fitness.

```
(define-test selection-rws
  (let ((pop (make-population 3)))
    (genotype-fitness-set! (car pop) 50)
    (genotype-fitness-set! (cadr pop) 25)
    (genotype-fitness-set! (caddr pop) 5)
    (assert-equal (selection-rws pop 25) (car pop))
    (assert-equal (selection-rws pop 50) (cadr pop))
    (assert-equal (selection-rws pop 70) (cadr pop))
    (assert-equal (selection-rws pop 76) (caddr pop))))
```

2.2 (selection-sus pop f)

Implements stochastic universal selection. This builds on top of roulette wheel selection: instead of passing a random value to **??**, select n genomes at once, dividing the fitness space evenly and selecting the genomes at those points. Typically, you would want to select a whole new population at once, which means the spread of your fitness scores controls the selection behaviour (higher spread, more variance, and vice versa).

```
(define-test selection-sus
  (let ((pop (make-population 3)))
```

```
  (genotype-fitness-set! (car pop) 50)
  (genotype-fitness-set! (cadr pop) 25)
  (genotype-fitness-set! (caddr pop) 5)
  (let ((vec-pop (list->vector (selection-sus pop 16))))
    (assert-equal (vector-ref vec-pop 0) (car pop))
    (assert-equal (vector-ref vec-pop 10) (cadr pop))
    (assert-equal (vector-ref vec-pop 15) (caddr pop)))))
```

3 Crossover procedures

3.1 (genotype-crossover gt1 gt2 #!optional rate-or-pt)

Implements the crossover operator. It simple takes two genotypes, randomly selects a point in the genotype with the longest solution, and cuts both genotypes at this point and switches the second parts from each genotype.

It takes an optional 3rd argument. If passed a floating point, it will override the default crossover rate which is .01. If passed an integer, it will be used as the point at which to cut, but if omitted will generate this randomly.

```
(define-test genotype-crossover
  (let ((gt1 (make-genotype '(1 2 3 4 5)))
        (gt2 (make-genotype '(6 7 8 9 10 11 12 13))))
    (receive (new-gt1 new-gt2) (genotype-crossover gt1 gt2 3)
      (assert-equal (genotype-data new-gt1) '(1 2 3 9 10 11 12 13))
      (assert-equal (genotype-data new-gt2) '(6 7 8 4 5)))
    (receive (new-gt1 new-gt2) (genotype-crossover gt1 gt2 7)
      (assert-equal (genotype-data new-gt1) '(1 2 3 4 5 13))
      (assert-equal (genotype-data new-gt2) '(6 7 8 9 10 11 12)))))
```

4 Mutation procedures

4.1 (genotype-mutate gt #!optional rate)

Implements the mutation operator. Currently, this simply shuffles each triangle a little bit and re-calculates its color (for those which are determined to be mutated according to the mutation rate). The default mutation rate is .01, which can be overridden by the second parameter.

5 Vectors

Basic functionality for 2d and 3d vectors.

TODO: add vector unit tests

6 Utility

6.1 (midpoint point1 point2 point3)

Returns a point which vaguely approximates the middle the triangle formed by the three points (2d vectors).

```
(define-test midpoint
  (let ((point (midpoint (make-vec2 0 10)
                         (make-vec2 10 0)
                         (make-vec2 0 0))))
    (assert-equal (vec2-x point) 2.5)
    (assert-equal (vec2-y point) 2.5)))
```

7 Benchmarks

```
(shutdown-engine)
```