Storeys

Brian Beckman

<2016-02-21 Sun>

Contents

1 Prelude

```
;; Copyright (c) 2016 Brian Beckman
```

"Storeys" is a pun on "story" and the levels of a dungeon. We avoid the word "level" because it's ambiguous between the level of advancement of a character and the level or storey of the dungeon.

2 Project Structure

2.1 Brute-Force Loads

The current system structure is just a sequence of load commands. Dependencies of one facility on another is expressed only implicitly by the sequence of those loads, and only in top-level, runnable lisp files. For example, *box* depends on *point*, but `box.lisp` doesn't load `point.lisp`. Instead, any file that wants to use *box* must load `point.lisp` before loading `box.lisp`. `box.lisp` *could* load `point.lisp`, but then `point.lisp` would be loaded more than once. Harmless but inefficient. If `box.lisp` doesn't load `point.lisp`, then `box.lisp` can't stand alone. That's the decision we made:

Facilities are not typically standalone, and their dependencies are implicit in the sequence of load commands.

Currently, there are two contexts in which this brute-force loading goes on:

1. any main code, such as `charms-test-5.lisp`
2. quickcheck code, like `test.lisp`

For example, in one intermediate version, `charms-test-5.lisp` loaded files in this order:

```
(load "point.lisp")
(load "box.lisp")
(load "glyph.lisp")
(load "world.lisp")
(load "storey.lisp")
(load "room.lisp")
(load "rendering.lisp")
```

The convention for quickcheck code is a little more subtle: it will load unit-test files out of the `test` directory in the same order as a main file does. For instance, the version of `test.lisp` corresponding to the version of `charms-test-5.lisp` noted above looks like this:

```
(load "~/quicklisp/setup.lisp")
```

```
(ql:quickload :cl-quickcheck)
(ql:quickload :defenum)
(ql:quickload :alexandria)
(ql:quickload :hash-set)

(let ((*random-state* (make-random-state t))
      (*print-length* 6)
      (*load-verbose* t))

  (shadow 'cl-quickcheck:report '#:cl-user)
  (shadow 'defenum:enum         '#:cl-user)
  (use-package :cl-quickcheck)
  (use-package :defenum)

  ;; This sequence of 'load' expressions should parallel the 'load's at the head
  ;; of any main like 'charms-test-5.lisp'.

  (load "test/point.lisp")
  (load "test/box.lisp")
  (load "test/glyph.lisp")
  (load "test/geo.lisp")
  (load "test/world.lisp")
  (load "test/storey.lisp")
  (load "test/room.lisp")
  (load "test/rendering.lisp")
  )
```

No doubt, you've deduced that main codes and test codes — anything that runs facilityy code — must also load *quickload* and any quickload dependencies like *alexandria*. Just as with our own facilities, you must do this by brute force.

Each file in the test directory must load its brother from the top-level directory. For example, `test/glyph.lisp` looks like this:

```
(load "glyph.lisp")

(quickcheck
 (is= 42 42))
```

Because we run `test.lisp` from the top level, `test/glyph.lisp` will load the top-level `glyph.lisp`. Of course, `test/glyph.lisp` must load any of its dependencies. It doesn't have any in the version we're talking about, so everything is fine. But `test/box.lisp` needs top-level `point.lisp`. So `test/box.lisp` looks like this:

```
(load "point.lisp")
(load "box.lisp")

;;; blah blah blah
```

```
(quickcheck
 (is= 42 42))
```

All this is cumbersome and brittle, so we will fix it with something like ASDF later.

2.2 TODO ASDF

3 Basics

3.1 Point

3.2 Box

3.3 Glyph

4 Storeys

4.1 World

The world has a sequence of *storeys*. There is a first one, but not a last one.

```
(defclass world ()
  ((storeys :accessor world-storeys :initform () :initarg :storeys)))
```

4.2 Storey

A storey has a matrix of *tiles*. It's likely that we will change the representation of a story to a sparse matrix in the future, so abstracting its representation is worthwhile prophylaxis (future-proofing).

The coordinate system of any storey has origin $(0, 0)$ so that array indices and tile coordinates are always identical.

```
(defconstant +storey-width+  256)
(defconstant +storey-height+ 256)

(defun make-storey (&key (width +storey-width+)
                         (height +storey-height+))
  (make-array `(,height ,width)
              :initial-element nil))

(let ((m (make-storey :width 10 :height 6)))
  (setf (aref m 2 1) :hi)
  m)
```

4.3 Tile

Each tile contains exactly one (possible nil) *geo*, zero or one *critters*, and a bag of *treasures*. We allow nil geos because most tiles will have nothing interesting in them. A nil tile means the same as a nil geo in a non-nil tile.

```
(defclass tile ()
  ((geo       :accessor tile-geo       :initform nil :initarg :geo)
   (critter   :accessor tile-critter   :initform nil :initarg :critter)
   (treasures :accessor tile-treasures :initform nil :initarg :treasures)
   (lighted   :accessor tile-lighted   :initform nil :initarg :lighted
  )))

  (let ((m (make-storey :width 10 :height 6)))
    (setf (aref m 2 1)
          (make-instance 'tile))
    m)
```

4.4 Room

A *room* is a rectangular region of tiles in a storey. It has a reference to its storey and a reference to a box specifying the top and left coordinates of the room with respect to the storey's origin $(0, 0)$, and a width and height.

4.5 Geo

A geo could be nil, meaning "nothing interesting here," or a wall, door, rock, or trap.

```
(defclass geo ()
  ((kind :accessor geo-kind :initform nil :initarg :kind)))
```

4.5.1 Wall

1. Inscribed

4.5.2 Door

1. Open
2. Closed
3. Locked
4. Spiked
5. Broken

4.5.3 Rock

1. Granite
2. Quartz
3. Magma
4. Lava

4.5.4 Trap

1. Gas
    (a) Poison
    (b) Drug
    (c) Blindness
    (d) Fear
2. DimMak
3. Dart
    (a) Poison
    (b) Drug
4. Fire
5. Boulder
6. Flood
7. Curse
8. Ice
9. Immoblization
10. Lightning
11. Pit
12. Hole
13. Teleport

4.6 Critter

4.6.1 Me

4.6.2 Monster

4.7 Treasure

4.7.1 Potion

4.7.2 Scroll

4.7.3 Armor

4.7.4 Weapon

5 Rendering

5.1 Screen

5.2 Window

5.3 Scroll-state

6 Me

6.1 Attributes

6.1.1 Dynamic

1. HitPoints
2. Mana
3. Energy
4. Rage
5. Focus

6.1.2 Static

1. Strength
2. Wisdom
3. Constitution
4. Stamina
5. Intellect
6. Charisma

7. Agility
8. Dexterity
9. Versatility
10. Mastery

6.2 Classes

6.3 Races

7 Treasures

7.1 Armor

7.1.1 Head

7.1.2 Shoulders

7.1.3 Chest

7.1.4 Arms

7.1.5 Wrists

7.1.6 Hands

7.1.7 Pants

7.1.8 Feet

7.1.9 Neck

7.1.10 Trinkets

7.1.11 Rings

7.2 Weapons

7.2.1 One-Handers

1. Swords
2. Daggers
3. Maces
4. Clubs
5. Fists

7.2.2 Two-Handers

1. Staves
2. Swords
3. Maces
4. Clubs

7.2.3 Ranged

1. Guns
2. Bows
3. Crossbows

7.3

8 Players

8.1 AI

8.2 Bots

Emacs 24.5.1 (Org mode 8.2.10)