Sleep Well/handmade-clojure planning document

Joshua Suskalo

January 25th, 2018

Contents

1 Game Design

The game is currently planned to be the game Cassie and I have planned on before, "Sleep Well"

2 Art Design

3 Sound Design

4 Game Development

4.1 Architecture

4.1.1 Entity/Component System

1. Example Usage

```
(defcomponent health-comp [num] number?)

(defcomponent player-input [x y jump]
        (s/keys :req-un [::x ::y ::jump])
        {:x 0 :y 0 :jump false})

(register-entity (create-health-comp 5)
                 (create-position 2 2)
                 (create-image-texture (sprite "Monster.png")))

(register-entity (create-player-input)
                 (create-position 0 0)
                 (create-image-texture (sprite "Player.png")))

(defsystem damage-over-time
  :on-update
  [health :health-comp]
  (fn [entity-id]
      (update-component-state! health dec)))

(defsystem player-movement
  :on-update
  [input :player-input
   position :position]
  (fn [entity-id]
    (let [x (:x input)
          y (:y input)]
      (update-component-state! position move x y)))
```

```
(defsystem render-sprites
  :post-update
  [texture :image-texture]
  (fn ...))

(defsystem kill-all
  :on-call
  [health :health-comp]
  (fn [entity-id]
    (destroy-entity! entity-id)))
```

2. Internal Structure

   (a) Systems

   Systems run over either all entities, or components of a specific set of types. When running over a set of types, it selects the shortest list of entities from the component types, and then iterates over them, filtering by only the entities which have the other required components. This requires minimal iteration and makes use of the hashmap lookup speed granted for looking up entities.

   Systems are kicked off by a simple function call, and some systems are put in a near-infinite loop. Systems are permitted to fire off other systems, however in these cases the functions that are used to fire off the other systems should do so concurrently and return immediately to prevent the first system from being blocked.

   (b) Entities

   Entities are simple unique ids, generated for each entity to identify them. Once you can identify an entity, all its data and associated behaviour can be found in components and the systems that operate on them.

   (c) Components

   Components are where all the data for the entire game's high-level systems will be stored. Systems can operate over components.

   A component has several parts, a component id, which is a namespaced keyword to uniquely identify it by name, and a spec (tied to the same namespaced keyword), which declares the data which will be stored inside the component.

(d) Data

Internal storage will be as follows:

- An atom which is simply a list of all entity ids
- An atom which maps entity ids to component ids and component data ids
  - This atom should have a two-way mapping, from entities to components, and vice-versa (Maybe don't have it two way, and only do it as a mapping from components to entities? Having it be two way can make it more efficient though for systems to do lookups.)
- An atom which maps from component ids to refs that store component data
  - A ref per component type (stored in the above atom) which maps component data ids to data stored

(e) Assemblages

Assemblages or entity-templates are mappings of template-ids to lists of components and their initial data, such that a generic method could be created to take an assemblage and return an entity with all the required components and initial data.

4.1.2 Physics

4.1.3 Renderer

The renderer should be using recent OpenGL features so that GLSL can be used instead of software rendering. OpenGL 4 should most likely be the feature set that's used, though I can pick what subversion later.

4.2 Tasks