Yoo Min Cha

91.301

Professor Martin

11 December, 2014

Final Project Report

Michael Jannino and I wanted to work on creating a 2d game and we went through an exhaustive search of all the available libraries to us that enabled graphics. We both thought Clojure was the best dynamic programming language available to us to make our project a reality. After watching the tutorial on <https://www.youtube.com/watch?v=9ilUe7Re-RA>, and recreating the game on our own desktops, we decided that we would create a modularized version of this game in an object oriented manner. We also referenced <http://oakes.github.io/play-clj/> to guide us along the way and we used [www.codeshare.io/](http://www.codeshare.io/) to share code.

We both worked on most of the code together because we had to learn how to work with the syntax and data structures of this programming language. We both put substantial work into utils.clj, core.clj, and entities.clj. My responsibilities included working on the AI and creating the movement functions in utils.clj.

In utils.clj I worked on the movement of our player entities by creating get-x-velocity and get-y-velocity.

(defn get-x-velocity

[{:keys [me? npc? x-velocity]}]

(cond

me?

(cond

(or (key-pressed? :dpad-left) (touched? :left)) (\* -1 max-velocity)

(or (key-pressed? :dpad-right) (touched? :right))

max-velocity

:else

x-velocity)

npc?

(if (> 5 (rand 100))

(if (< 5 (rand 10))

(\* max-velocity-npc (- (rand-int 10)))

;else

(\* max-velocity-npc (rand-int 10)))

0)

:else x-velocity))

(defn get-y-velocity

[{:keys [me? y-velocity can-jump?]}]

(if me?

(cond

max-jump-velocity

:else

y-velocity)

y-velocity))

These functions both take in a sequence of parameters that get mapped to :keys and by testing each of these parameters, we return an altered set of parameters or return the original ones. We test to see what kind of player entity the object is and then we allow movement through key pressed events or if the object is a zombie we allow it to move freely on its own.

I was able to get all of our objects into the game by creating a vector of objects with [] and by calling the create player entity functions from the entities and zombies class.

[(apply zomb/create-zombie zombie-images)

(apply zomb/create-zombie zombie-images)

(apply zomb/create-zombie zombie-images)

(apply zomb/create-zombie zombie-images)

(apply zomb/create-zombie zombie-images) ;Generating some enemies!!

(apply zomb/create-zombie zombie-images)

(apply zomb/create-zombie zombie-images)

(apply ents/create-player player-images)]

I initially wanted to implement (take 8 (repeat (apply zomb/create-zombie zombie-images))), but this created the problem of creating the same zombie eight times instead of creating eight zombies with their own set of x parameters.

Here is the move function that moves the player/zombie entities around but the hitbox was not registering when calling my death function.

(defn move

[{:keys [delta-time]} {:keys [x y can-jump?] :as entity}]

(let [x-velocity (u/get-x-velocity entity)

y-velocity (+ (u/get-y-velocity entity) u/gravity)

x-change (\* x-velocity delta-time)

y-change (\* y-velocity delta-time)]

(if (or (not= 0 x-change) (not= 0 y-change))

(assoc entity

:x-velocity (u/decelerate x-velocity)

:y-velocity (u/decelerate y-velocity)

:x-change x-change

:y-change y-change

:x (+ x x-change)

:y (+ y y-change)

:can-jump? (if (> y-velocity 0) false can-jump?)

:hitbox (rectangle (:x (+ x x-change))

(:y (+ y y-change))

(:width entity)

(:height entity)))

entity)))

The move function takes in two discrete hash map parameters; first one for delta time, and second one for x,y, and can-jump?. We create a local binding for the x and y velocities and change. By calling the get x and y velocity functions, we can bind then to our x and y velocities. Y-velocity has to take into account the gravity defined in the utilities class. X and y change determines the amount of change by factoring the velocities with the change in time. Only if there is movement then we change the position of the entities on the screen, or else we return the entity as is. The hitbox parameters also gets updated with the rest of the x and y values.

(defn death

[entities]

(if-let [zombie (filter #(contains? % :npc?) entities)]

(let [player (some #(when (:me? %) %) entities)

touch-zombie (filter #(rectangle! (:hitbox player) :overlaps (:hitbox %)) zombie)]

(if (touch-zombie)

(set-screen! enclojure death-screen)

entities)

entities))

The death function takes in the sequence of entities in the game world and filters them based on the entity type. If it is an npc, we know that it is a zombies and we bind it with the zombie label. When we bind player, we don’t use the filter function because we don’t need to go through the entire vector of entities and because we only need the first entity that returns a logical true value of :me?, the player. We now have a list of zombies around our player and we can filter out which zombies is touching our player using the overlap function to determine if our player hitbox rectangle is touching the zombie hitbox rectangle. If we happen to find that a zombie is touching our player, we call the enable the death-screen window and game over.

If you would like to run our code, you can download the standalone IDEs Light Table, or NightCode. NightCode is easier to use, all you have to do is import the folder into the IDE and press ctrl+b to build the code. Inside the enclojure/target/ folder is the Standalone executable to launch the game. The direction keys direct movement of the player, and to win you must make it all the way across the map and touch the lamppost.

Included Folders/Files

-resources

--background.png, coin.png, FAILURE.png, fredm.png, level1.tmx, okay.png, SUCCESS.png, tileSet.png, zombie.png

-src

--enclojure

---core

----desktop\_launcer.clj

-src-common

--enclojure

---coins.clj, core.clj, entities.clj, utils.clj, zombies.clj

-target

--all play-clj wrapper functions

-project.clj