Exercise 7: A Simple Video Game

EECS 111

Due Friday, March 9th by Midnight

Introduction

In this exercise we'll do a simple reimplementation of the classic arcade game *Asteroids*. Since this is your first imperative programming assignment, we've already implemented most of the core functionality for animation. All you need to do is the implement the procedures that update the individual objects in the game. **Note:** if you have never played the game *Asteroids* or it's been long enough that you've forgotten how it looks here is an online example.

IMPORTANT: Before beginning your assignment, you need to change Racket:

- We have graduated from the intermediate student language to the advanced student language. Go to the Language menu in DrRacket, select Choose Language, and then click on Advanced Student. Now click Okay. Note that this will change the Step button into a Debug button. We'll talk about this later in class.

ALSO IMPORTANT:

- Do *NOT* require libraries other than those from `2htdp/•` and `cs111/•`. If you really need some function from other libraries, please make a piazza post and we can discuss how to resolve it.
- This exercise works by incrementally building small pieces to create a simple video game. If you don't follow the instructions step-by-step or don't read all of the instructions, this exercise will be way more difficult than it should be. There's a lot of code that we've already written for you, and you'll have to make references to that code. It's all in the directions. **Keep calm and follow along** :)
- If you add a call to `(asteroids)` to your file you **MUST** remove it before submitting. If you do not the hand-in pane will hang with the message "Checking submission. . . " and it will never change to "Handin successful." If you see the message "Handin successful." you don't have to worry about this.

Basic Structure of a Computer Game

Most computer games consist of a set of objects that appear on screen. For each object on screen, there is a data object in memory that represents it, along with procedures for redrawing it on screen and for updating its position and status. The basic structure of the game is therefore a loop that runs indefinitely: first calling the update procedure for every object, and then calling the redraw procedure for every object. Once redraw is complete, the whole process repeats. We've already implemented the main game loop and drawing functionally, so you need only fill in the code for the update procedures.

The *Asteroids* game

The *Asteroids* game consists of three kinds of on-screen objects:

- the **asteroids**

  Randomly sized obstacles that float around the screen

- the **player**

  Which tries to navigate the space without crashing into one of the obstacles

- **Missiles**

  Which the player can shoot from the ship to the asteroids in its path.

In our game, the player will pilot the ship using the **arrow keys** on the keyboard. The left and right keys turn the player's ship, and the up arrow accelerates it in the direction it's pointing. The **space bar** fires a missile. The player's goal is just to survive and not get hit by an asteroid.

From here on out, assume that the "ship" or "player" in prose and `a-player` or `p` in code are equivalent.

The Game Code

If you're unsure of what to do at any given point, it's probably worth it to revisit this section. It probably has the answer. This section contains a description of the code in this file you will need to complete the assignment. The *Your Job* section farther below describes what you'll need to do in the code.

Start Racket by clicking on the `exercise_7.rkt` file. You can start the game by running the `(asteroids)` procedure and stop it by closing the game's window. At the moment, none of the player's controls work, so it's not much of a game. For the assignment, you'll be implementing player control of the ship.

Representing Points and Velocities in Space

The game code uses the `posn` struct to represent points and vectors. The `posn` struct is build in, but behaves as if you defined it as:

```
;; a posn is a (make-posn number number)
(define-struct posn (x y))
```

That is, `posn` object contains two fields, `posn-x` and `posn-y`, which store the x and y coordinates of the point, respectively, and you can create a new `posn` object by saying `(make-posn x y)`

There are two important procedures you will use with posns:

The First - Posn Addition

```
;; posn-+: posn posn -> posn
(posn-+ a b)
```

Returns the sum of the two posns - the new posn's x coordinate is the sum of the x coordinates of the two, and its y coordinate is the sum of the y coordinates of the two. For example:

```
(check-expect (posn-+ (make-posn 1 2) (make-posn 3 4))
              (make-posn 4 6))
```

The Second - Posn Multiplication by a Scalar

```
;; posn-*: number posn -> posn
(posn-* num posn)
```

Multiplies num (a scalar) by posn (a vector). The x and y coordinates of the original posn are each multiplied by num. For example:

```
(check-expect (posn-* 2 (make-posn 3 4))
              (make-posn 6 8))
```

Game Objects in Memory

Each of the on-screen objects is represented by a data object with a set of built-in fields used by the animation system:

- `;; game-object-position: game-object -> posn`

  `(game-object-position object)` The location on the screen where the object should appear. It returns a posn object. It's just a simple object that contains two fields, `(posn-x p)` and `(posn-y p)` representing its x- and y-coordinates.

- `;; game-object-velocity: game-object -> posn`

  `(game-object-velocity object)` The speed and direction in which it's moving. It's also a posn (i.e. a vector). On each update cycle, the animation system will automatically move the object based on its velocity. A velocity of `(make-posn 10 5)` means the object moves 10 pixels per second horizontally, and 5 pixels per second vertically.

- `;; game-object-orientation: game-object -> number`

  `(game-object-orientation object)` The direction the object is pointing (a number, expressed in radians). This only matters for the player, since the other objects are circles and so don't have any meaningful orientation.

- `;; game-object-rotational-velocity: game-object -> number`

  `(game-object-rotational-velocity object)` The speed at which the object is turning, in radians per second. Again, the animation system automatically updates the orientation field based on the rotational-velocity field.

- `;; game-object-radius: game-object -> number`

  `(game-object-radius object)` This is how near another object can come to this object without hitting and destroying it. It's used internally by the physics code; you don't have to worry about it.

With this assignment, we'll be starting to use imperatives. In particular, we will use the imperative procedures `set-game-object-velocity!` and `set-game-object-rotational-velocity!` to change the speeds of the game objects:

- `;; set-game-object-velocity! : game-object posn -> void`

  `;; Effect: update the velocity of gameobject to the specified value`

  `(set-game-object-velocity! gameobject velocity)`

- `;; set-game-object-rotational-velocity! : game-object number -> void`

  `;; Effect: update the rotational velocity of gameobject to the specified value`

  `(set-game-object-rotational-velocity! gameobject rotation-rate)`

Note that these return the `void` type - i.e. they don't return a meaningful value. Instead, they're imperatives: they're called because of the way they change the system's memory, rather than because they generate a useful return value.

Sensing the Player's Input

When the player presses keys, the operating system sends messages to Racket called *events.* Racket processes those messages and will call a set of procedures with names like `on-left-press` (pressed the left arrow) and `on-left-release`(released left arrow). You will fill in those procedures.

Your Job

In this assignment, you'll write the update logic for the player's ship and its missiles. This consists of filling in the procedures `update-player!`, `update-missile`, and the keyboard event handler procedures.

Part 1: Steering

Start by adding code to the `on-left-press` and `on-left-release` to turn and stop turning the ship (counterclockwise), respectively. Remember that you can adjust its turn rate using `set-game-object-rotational-velocity!` (see above).

Now fill in `on-right-press` and `on-right-release` to turn the ship in the opposite (clockwise) direction.

A Quick Interlude on the "posn" Struct

Member when we talked about the posn struct earlier? I member!

Like we said before, a posn is essentially just a vector - it has both an x and a y component, and it can represent all of the things that a vector would. If you haven't done any work with vectors, here's a condensed overview:

- As you've seen them so far posns (like vectors) are able to represent positions by simply viewing them as translations from an origin.
- Posns are also able to represent velocities with x and y components. For example, if my velocity is (make-posn 1 2), I'll move 1 unit in the positive x direction and 2 units in the positive y direction every second.

From here on out, it's important to keep in mind what your "posn" represents. It'll make everything easier :)

Part 2: Moving

Now we want the ship to be able to move around. That means we need to be able to set its velocity. The player will use the up-arrow key to control forward motion.

Add code to `on-up-press` and `on-up-release` to set the player's velocity. When the key is released, the velocity should be set to `(make-posn 0 0)`, i.e. to zero, meaning that the ship will stop moving.

When the up arrow key is pressed, it should move in the direction the ship is pointed. You can use provided `(forward-direction p)` to get a vector pointing in the direction the player's ship is pointing. However, the vector is very small, so if you set the player's velocity to that vector, the player will only move one pixel per second, which isn't terribly useful. But you can use posn-* (multiplying a posn by a scalar) to make it larger. So if you write the following (**Note**: `some-speed` here is something you need to fill in with a reasonable value):

```
(posn-* some-speed (forward-direction p))
```

You will get a vector pointing in the right direction that will move the player at *some-speed* pixels per second.

Part 3: Making moving harder

One of the things makes Asteroids challenging is that you can't just stop and go. Instead, pressing the up arrow key accelerates you in the forward direction. Letting go doesn't stop you; it just stops the acceleration (that is, after all, how movement really works in space). In order to stop, you have to turn around and accelerate in the opposite direction, hopefully just enough to exactly cancel out your original acceleration.

Implementing acceleration

Implementing gradual acceleration is easy. Before, we set the player's velocity to a fixed value. But now let's add a fixed value to the player's velocity. Insert the following code in `on-up-press`:

```
(set-game-object-velocity! a-player
                           (posn-+ (game-object-velocity a-player)
                                   ; you need to replace acceleration here
                                   ; with some vector pointing in the direction
                                   ; of the ship
                                   acceleration))
```

Where acceleration is some vector pointing in the direction of the ship, such as the one you used in the previous section (this could just be the forward direction of the ship, you can also scale this forward direction to accelerate faster or slower). Then every time the game updates, the value of acceleration gets added to the player's velocity. When you run this you should see that every time you press the `up` key the ship will accelerate a little more.

There are a number of problems with this, though. So let's fix them.

Turning off braking

One problem is that if we still zero out the velocity in `on-up-release`, then the player will stop dead whenever they let go of the key. So, you'll want to start by removing that code from `on-up-release`. For now, you can just change the body of `on-up-release` to be `(void)`, meaning do nothing. Yes we're going back and removing stuff we've already done. We set up the assignment this way to show how changing one thing can lead you to rethink another part of your code. Hopefully this semi-contrived process helps convey a better understanding of the movement mechanics.

Continuous acceleration

Another issue is that `on-up-press` is called only once when the user presses the key. If we update our velocity in that function then we will only accelerate when the user presses the key. Instead we want to accelerate continuously while the key is held down. To fix this change your code to adjust the speed in `update-player` instead of `on-up-press` which is automatically called every time the game updates (30 times per second).

Controlling acceleration

Unfortunately, with that change the player accelerates indefinitely based on whether you have pushed the key or not. So we need to change it so that it only accelerates when you press the key. So we want something like:

```
(when firing-engines?
    ... do the acceleration ...)
```

The `when` special form is like `if`: it says only run the `... do the acceleration ...` code if the `firing-engines?` variable is true, otherwise it returns `(void)`. We use when here simply for convience: Otherwise we would write lots of code that looked like: `(if x? (... do-something ...) (void))`.

But how does the system know if the player is firing the engines or not? That's where `on-up-press` and `on-up-release` come in. Modify them to update `firing-engines?` so that it's always true when the player is pressing the up-arrow key and false when they aren't. Remember that `firing-engines?` is just a variable, not part of a struct, so we use `set!` to change it.

Note: Once again we're going back to change things that we'd previously called finished. Also note here that these changes make both `on-up-press` and `on-up-release` much simpler functions than they were before. This is not a regression, just a different structuring of the code to give us the acceleration properties we're looking for.

Pro-rating the acceleration

The last problem is that a fixed acceleration gets added in every time the game calls your `update-player!` procedure, but since different computers run at different speeds, that happens more often on faster computers and less often on slower computers, with the result that the player accelerates faster or slower depending on how old your computer is. Not good - we want everyone to get the same experience!

So what you need to be able to do is to pro-rate the acceleration by how long it's been since the last time you did an update. Use `posn-*` to multiply the acceleration by the value in the variable `inter-frame-interval`, which holds the fraction of a second that it's been since the last update. Then add that pro-rated acceleration into the velocity as before. Note that this will make the player accelerate much more slowly, so you may want to increase the multiplier on the acceleration accordingly.

Part 4: Blowing stuff up

Now modify the code so that a missile is fired each time the player presses the space bar. You can create and fire a missile using the `(fire-missile!)` procedure.

Self-destructing Missiles

The one remaining issue with the game is that if a missile misses its target, it will continue to move until it does hit something. That could very easily be the player. To prevent that, it's common to have missiles self-destruct after a specified period of time. Missiles in the game have a field called *lifetime* that has the number of updates (number of calls to `update-missile!`) the missile should continue to move before self-destructing. You can get the current lifetime of the missile using the `missile-lifetime` procedure and set it using the `set-missile-lifetime!` procedure.

Find the `update-missile!` procedure and modify it so that it decreases of the lifetime of the missile by 1 each time it's updated. If the lifetime is zero, you should destroy it by calling `destroy!` on it.

A few things

- The way the game is right now, your player won't respawn when it dies
- You'll have to close the window and rerun `(asteroids)`to start again. If you add a call to `(asteroids)` to your file you **MUST** remove it before submitting. If you do not the hand-in pane will hang with the message "Checking submission..." and it will never change to "Handin successful." If you see the message "Handin successful." you don't have to worry about this.
- You will also still be able to shoot after you're dead

These are intended - don't worry! Your game is working fine.

Congratulations, you now have a working video game! Turn it in :)