EECS-111 Exercise 4: A Simple Video Game

Out: Saturday, October 31
Due: Friday, November 6, noon

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.

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**'s ship
  Which tries to navigate the space without crashing into one of the obstacles
- **Missiles**
  Which the player can shoot from the ship to destroy asteroids in its path.

In our game, the player will pilot the ship using the **arrow key**s 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.

The game code

Start Racket by clicking on the Exercise 4.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. But for the assignment, you'll be implementing player control of the ship.

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 *object*)
  The location on the screen where the object should appear. It returns a posn object, like we used in tutorial 2. 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 *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 *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 *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 *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.

The player's ship will always be stored in the global variable the-player, so you can get its position by saying (game-object-position the-player). You can change its attributes using the imperative procedures, set-game-object-velocity!, set-game-object-rotational-velocity!, etc.:

-
  ```
  (set-game-object-velocity!  the-player
                              (make-posn 0 0))
  ```
  Set's the player's ship's speed to 0.

-
  ```
  (set-game-object-velocity! the-player
                             (posn-+ (game-object-velocity the-player)
  ```

(make-posn 1 2)))

Adds the vector (1, 2) to the player's velocity. In other words, it accelerates the player's ship horizontally by 1 pixel per second and vertically by 2 pixels per second.

- (set-game-object-rotational-velocity! the-player 2)
  Sets the player's ship to rotate clockwise at 2 radians per second. -2 will make it rotate counter-clockwise.

Sensing the player's input

When the player presses keys, the operating system sends messages to Racket called **events**. Racket processes those message and will calls a set of procedures with names like on-left-press (the left arrow was pressed) and on-left-release (the left arrow was released). 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 start and stop the turning of 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.

Part 2: Moving (easy version)

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).

When the up arrow key is pressed, it should move in the direction the ship is pointed. You can use (forward-direction the-player) to get a vector pointing in the direction the player 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-* (multiply a posn by a number) to make it longer. So if you say:

```
(posn-* speed (forward-direction the-player))
```

You will get a vector pointing in the right direction that will move the player *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. And 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 we *add* a fixed value to the player's velocity:

```
(set-game-object-velocity! the-player
                                   (posn-+ (game-object-velocity the-player)
                                           acceleration))
```

Where *acceleration* is some vector pointing in the direction of the player, such as the one you used in the previous section. Then that every time the game updates, the value of *acceleration*, gets added to the player's velocity.

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 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 null, meaning do nothing and just return the magic value null (nothing).

Continuous acceleration

Another issue is that if we put that update in on-up-press, then we only accelerate at the moment the user first presses the key; we don't keep accelerating. So the player would have to keep hitting the key over and over. So move the update from on-up-press to the update-player! procedure. Update-player! gets called every time the game updates (30 times a second), so the player will keep accelerating.

Controlling acceleration

Unfortunately, now the player accelerates whether you push 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.

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 she isn't.

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 ship accelerates faster or slower depending on how old your computer is. Not good.

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.

Part 5: suicide 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. And 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.

You now have a working video game!