Exercise 5:
A Simple Video Game

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 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*** add new require command to the file, as they will break the autograder.
- 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 don't, your game will start running inside the autograder and never exit, freezing the autograder and leading to your getting a zero on the assignment.

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 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.

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 5.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[1] to represent positions in space and velocities of travel. The `posn` struct is built in, but behaves as if you had defined it as:

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

That is, `posn` object contains two fields, `x` and `y`, accessed with the `posn-x` and `posn-y` procedures, respectively, 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)`.[2]

When a `posn` is used to represent a point in space, the `x` and `y` fields hold the coordinates of the point on the screen, in units of pixels.

When a `posn` is used to represent a velocity, the `x` field holds the number of pixels per frame that the object is moving horizontally, and the `y` field holds the number of pixels per frame that it's moving vertically.

Adding a velocity to a point

If you want to shift a position by a specified velicity, you can use the `posn-+` procedure.

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

This returns a new posn whose `x` coordinate is the sum of the `x` coordinates of the inputs, and whose `y` coordinate is the sum of the `y` coordinates of the inputs. So if you have a location on the screen, represented by a `posn`, and an amount you want to shift it by, also represented by a `posn`, you can find the shifted

[1] Short for "position".

[2] If you've taken EA1 or MATH 240, A `posn` is a vector. It has both an x and a y component, and it can represent all of the things that a vector would.

position by adding them with `posn-+`. For example, if we shift the position `(make-posn 1 2)` right 3 units, and up 4 units, we get the position `(make-posn 4 6)`:

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

Scaling a velocity by multiplying it

You can scale a velocity, represented by a posn, to be faster or slower by multiplying it by a number. If you multiply it by 2, it moves twice as fast in the same direction. If you multiply it by a half, it moves half as fast, again, in the same direction.

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

Multiplies *num* by *posn*. 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. You can access them using the following procedures:

- **`; 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 to change the system's memory, rather than to generate a useful return value.

Sensing the Player's Input

When the player presses keys, the operating system sends messages to Racket called *events*. When Racket receives those messages, it calls procedures. For example, it calls the procedure `on-left-press` when the left arrow is pressed and it calls `on-left-release` when the left arrow is released. Part of your job for this assignment is to fill in those procedures to respond appropriately.

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 `on-left-press` and `on-left-release` to turn and stop turning the player's ship, respectively. So pressing the left arrow should start the ship turning counter-clockwise, and releasing it will stop the rotation. Remember that you can adjust the ship's turn rate using `set-game-object-rotational-velocity!` (see above). Note that the player's ship is in the variable `the-player`.

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

At this point, try playing your game by running the `(asteroids)` procedure! It won't do much at this stage since all you've implemented is turning. But it's a good sanity check to make sure you're code is doing what you expect.

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 (forward speed). 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), which will make 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 the forward-direction procedure to tell what direction is forward. It takes the player's ship or some other game object as an argument and returns a posn pointing the in ship's forward direction. However, it corresponds to a very slow velocity, which

isn't terribly useful. But you can use posn-* (multiplying a posn by a scalar) to make the velocity faster. This will give you a speed vector in the forward direction of the *the-ship* and a speed of *speed*:

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

Fill in *speed* with the speed you want.

Try again to run your game with `(asteroids)`. Confirm that your code is doing what you expect before moving on. Also this is your last hint so remember to keep testing your code at the ends of future sections.

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. In the code we wrote before, we set the player's velocity to a fixed value. But now let's add that fixed value to the player's velocity. Insert the following code in on-up-press:

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

Where *acceleration* is a pretty much what you had before:

```
(posn-* acceleration-rate
        (forward-direction the-player))
```

Then every time the game updates, this 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 while turning

If you run your game now you might notice that when you hold down the up key and then turn we continue to move in the same direction we started. We don't accelerate in the new direction we've turned. This issue happens because as soon as we rotate the new key we pressed (left or right) takes precedence over the up key. We won't fire the on-up-press event (and therefore accelerate) again until we release the up key and repress it.

We want to allow a player to accelerate while turning so we'll need to fix this. To do so, change your code to adjust the speed in `update-player!` (and not `on-up-press`). `Update-player!` is automatically called every time the game updates (30 times per second).

Controlling acceleration

Unfortunately, with that change the player accelerates indefinitely, whether you 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?
    … add the acceleration …)
```

The `when` special form is like `if`: it says only run the "... *add the acceleration* ..." code if the `firing-engines?` variable is true, otherwise `when` does nothing and returns `(void)`. We use `when` here simply for convenience. Otherwise we would write lots of code that looked like: `(if x? 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 holding down 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.

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. Notice there's also an `on-space-press` procedure you can fill in.

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:

```
; set-missile-lifetime! : missile number -> void`
; Effect: update the lifetime of the missile to the specified value (a number)
(set-missile-lifetime! missile lifetime)`
```

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

- Your player won't respawn when it dies. To play again, just close the window and rerun `(asteroids)`.
- You will also still be able to shoot after you're dead. Don't worry about it.

Congratulations! You now have a working video game. Turn it in! But if you added a call to `(asteroids)` to your file you **must** remove it before submitting. If you don't you'll crash the autograder and get a zero.