Tutorial 8

11/29/2023

100/100 Points

Attempt 1

Review Feedback
11/29/2023

Attempt 1 Score:
100/100

View feedback

Anonymous grading: **no**

Unlimited Attempts Allowed
11/29/2023

Details

NOTE: You are not allowed to drop this assignment.

In this tutorial we'll do a simple re-implementation of the classic arcade game Asteroids. Since this is your first imperative programming activity, we've already implemented most of the core functionality for animation. Today we'll be working on the *rendering* aspect of the game and then you'll be focusing on the mechanics and extensions of the game in the Exercise.

This assignment will start here in the tutorial and then be finished in Exercise 8.

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 functions for redrawing it on the screen and for updating its position and status. The overarching structure of the game really just a loop that runs indefinitely: first calling the `update` function for every object and then calling the `render` function for every object. Once the rendering part is complete, the whole process repeats. We've already implemented the main game loop and drawing functionally for you so today you'll be focusing on the code for the update functions.

This will be similar in approach to our work on Snake. Except, now we have imperatives so we have the ability to update structs rather than having to make whole new ones and methods that will simplify our game design.

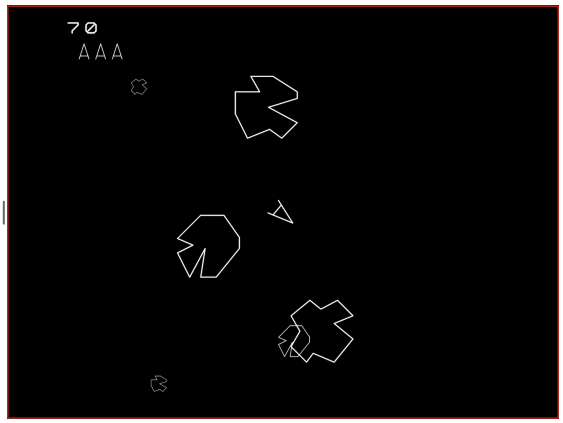
Asteroids

Asteroids was an incredibly popular multi-directional arcade game written by Lyle Rains and Ed Long and released by Atari in 1979. It's now considered one of the most influential video games of all time, both for its effect on players as well as game developers. The objective of the game is to shoot asteroids moving about in the game space while avoiding collisions with your ship and sometimes fending off attacks from enemy spaceships. If you've never played the game, there's a **fully functional version available here on the web (https://www.echalk.co.uk/amusements/Games/asteroidsClassic/ateroids.html)**.

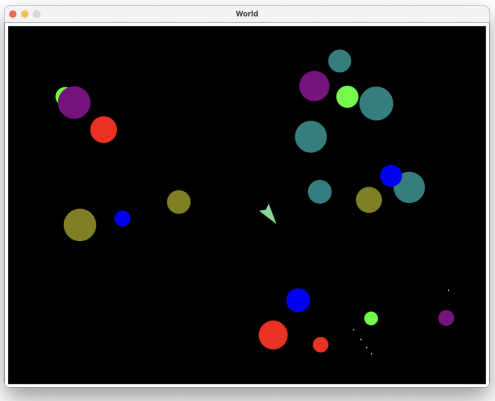
class…but for a very rough comparison of computing power, odds are your current laptop is running at a clock speed of around 3**G**Hz (or 3000MHz) (and probably has upwards of 4 cores running at this speed). That means your computer is roughly *twelve-thousand times* more powerful than the original computer than ran Asteroids. Really this is mostly just a fun fact and doesn't affect this assignment, but I like looking back at computer history to see how far we've come.

Today, you'll be finishing up an implementation of a simplified version of asteroids that isn't **feature complete** (i.e. it doesn't have all of the features of the original game) but does emulate its core functionality. Here's a comparison of what the real Asteroids looks like versus what ours will look like:

Real Asteroids



Simplified Asteroids



Our simplified game (in the tutorial) will consist of three kinds of on-screen objects:

- some `asteroid`s which are randomly sized obstacles that float around the screen
- a `player`, which tries to navigate the space without crashing into one of the asteroids
- `missile`s, which the player can shoot from the ship to blow up the asteroids in its path

And in the Exercise (8) you'll add:

- `ufo`s, which are enemy AIs that try to get the player
- `heat-seeker`s, which are special missiles that track objects

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 while the up arrow accelerates it in the direction it's pointing. The space bar fires a `missile`. The `s` key fires a `heat-seeker` (in the Exercise). The players's goal is just to

Getting Started

If you're unsure of what to do at any given point, *just come back to this section*. This section contains a description of most of the code in the `tutorial_8.rkt` and `asteroids_lib.rkt` file that you will need to complete the assignment. The **Your Job** section farther below describes what you'll need to do in the code.

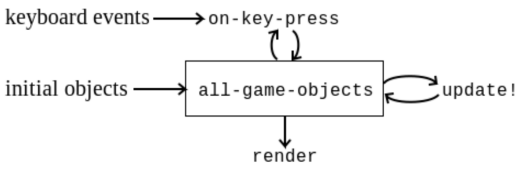
> **Note:** If you decide to become any sort of programmer or software engineer later in life, this sort of task will become very familiar. You're given a large existing *code base* and asked to add new functionality. Before you can add the new functionality, you need to spend some time familiarizing yourself with the existing code. **This is why it's so important to write readable and well-commented code because at some point…the roles will be reversed!** Make sure to watch the pre-recorded lecture videos as they'll give you insight into how the game engine for Asteroids actually works.

Tutorial 8 Starter Files (https://bain-cs111.github.io/course-files/tutorials/tutorial_8_template.zip)

Start Racket by opening the `tutorial_8.rkt` file. You can start the game by running the `(asteroids)` function in the REPL and stop it by closing the game's window. At the moment, much of the player's control doesn't work and the visuals leave something to be desired.

Overview of The Asteroids Library

The code of Asteroids is structured around the primary game state, the collection of all game-objects, and an event loop that continuously accepts keyboard events and updates the game state (this mostly happens in the `asteroids_lib.rkt` library).



When the game is running, the event loop calls an `update!` method on each object in the game state to update their positions and velocities; then uses the `render` method to repaint the screen at a rate of 30FPS (frames per second). The event loop also dispatches the keyboard events to the on-key-press function. The `on-key-press` function then adjusts the player's orientation and/or creates new missile objects accordingly

You'll be working on defining what the game objects are (properties/fields) and what they do (methods) in the `tutorial_8.rkt` file.

The `game-object` Struct

template for other structs and we won't be creating any `game-object`s directly.

The starter code provides three kinds of structs: the `player`, the `asteroid` and the `missile`. All three kinds of structs inherit the `game-object` struct:

```
;; A game-object is a (make-game-object posn posn number number)
(define-struct game-object
  (position velocity orientation rotational-velocity)
  #:methods
  ...)
```

Every instance of game-objects comes with (at least) four pieces of data that is available through the usual accessor functions:

- The `position` field: stores a `posn` struct representing the position of the object on the screen.
- The `velocity` field: stores a `posn` struct representing the velocity of the object. The velocity is a vector.
- The `orientation` field: stores a `number` representing the angle between the object and the x-axis in radians. Note that in the computer's coordinate system, the y-axis points **downwards** unlike in Snake.
- The `rotational-velocity` field: stores the `number` representing the angular velocity (think of it like spin) of the object (rad/s).

> Note: as we are working with imperative language features, all four fields come with mutators, e.g. `set-game-object-position!`, `set-game-object-velocity!`, etc., that lets you mutate the content of the fields in any `game-object` instance.

Additionally, a `game-object` has four methods defining its behaviors:

- `; update! : game-object -> void`: the `update!` method defines the behavior of each kind of game object. Each different sub-struct should implement its own `update!` method to update their state. The event loop calls the `update!` method on each game object 30 times per second.
- `; render : game-object -> image`: the `render` method defines the appearance of each kind of game object. It should return an `image`. The event loop calls the `render` method on each game object and draws the resulting images on the screen.
- `; radius : game-object -> number`: the `radius` method returns the size of each game object. The size must be a positive number. This method is used for collision detection.
- `; destroy! : game-object -> void`: the `destroy!` method implements destruction of a game object. The event loop calls the `destroy!` method of a game object when it collides with another game object. The default implementation of `destroy!` in game-object simply removes itself from the global `all-game-objects` variable that holds the list of all game objects.

The Game State

The game state consists of three variables:

- The variable `all-game-objects` is a `list` holding all of the objects in the game. Each object is an

- The variable `firing-engines?` is a `boolean` flag indicating whether the engine is currently on (`#true`) or off (`#false`), and hence indicates whether `the-player` should be accelerating or not. When the user presses the "up" key, the Asteroids library sets `firing-engines?` to `#true` (**you do not need to do this; it will do it for you**). When the user releases the "up" key, the Asteroids library sets `firing-engines?` to `#false` (**again it will do it for you**).

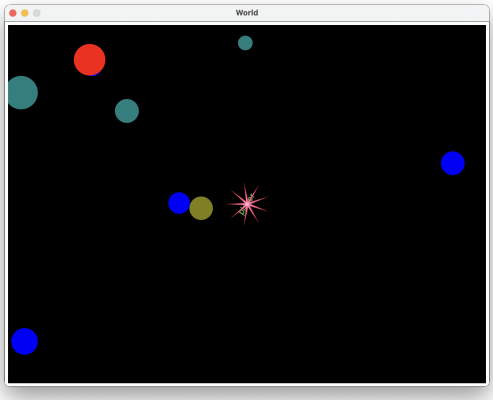
Testing while you work

After you add a feature, make sure to both write some `check-expect`s but also **actually run the game**.

> **Note**: Several of the built in tests will fail because they're for stuff we'll work on in the second half of this assignment in Exercise 8. Feel free to either comment them out or ignore them…but don't forget about them for the Exercise!

Remember, you can start the game by running the `(asteroids)` function in the REPL and stop it by closing the game's window.

Note that your player won't respawn when it dies. It'll just "explode" like in the below screenshot. To play again, just close the window and rerun `(asteroids)`.



Representing movement in Asteroids

The game code uses the `posn` (short for position) struct to represent positions in space along with velocities of travel. The `posn` struct is built in but behaves as if you had defined it as we normally do with custom `struct`s:

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

In other words, you **DON'T** need the above code and instead the `posn` object **already** contains two

`y` coordinates of the point, respectively, and you can create a new `posn` object by saying `(make-posn x y)`.

- 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 velocity, you can use the `posn-+` functions.

```
; 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 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))
```

Sensing the Player's Input

You won't need to worry about this as `asteroids_lib.rkt` takes care of all of this but it works the exact same way as Snake. When the user hits a key, the game "hears" the key and calls the appropriate function.

We'll start by completing this part in the tutorial today. **The further you get in this assignment today, the less work you have to do in Exercise 8. If you do not finish today, you're still responsible for the stuff in this assignment as Exercise 8 will assume you've already finished this part.**

Q1: The `player` Struct

Let's start by implementing the `player` struct. Right now, the player does not respond to the user's command because it inherits the default `update!` method from `game-object` which does nothing. Complete the `update!`, `render` and `radius` methods for the player:

- The `render` method returns the `image` of the player.
- The `radius` method returns the size (`number`) of the player. The size must be a positive number.

> Note: You may pick any design you like as the appearance of the player object in the render method and any size appropriate in the radius method. The size usually matches the image of the player object. For more information about how the Asteroids library uses the update! and render method, see the `game-object` struct and the Overview section. There are a TON of built-in image functions that you can choose from or you can feel free to make your own function to create an image: **2htdp/image library (https://docs.racket-lang.org/teachpack/2htdpimage.html)** .

- The `update!` method should speed up the player in forward direction (i.e. appropriately mutates the velocity field) **when** the engine is on.

> **Hint**: Review the Game State section to see how to check if the engine is currently firing or not. Let `v` denote the original velocity of the player and a denote the acceleration. Then the new velocity of the player should be `v+a`. If `p` is the player object, then the code will look like:

```
(set-game-object-velocity! p (posn-+ (game-object-velocity p) acceleration-expr))
```

- The velocity `v` is computed by the function call `(game-object-velocity p)`.
- The vector addition, `v+a`, is calculated using the `posn-+` function.
- `acceleration-expr` is an expression computing the acceleration, `a`, pointing in the forward direction.

To simplify your job, the `asteroids_lib.rkt` library provides the `forward-direction` function that takes a game-object and returns a unit vector (a `posn` struct) pointing in the forward direction of the given object by inspecting its orientation field.

```
; forward-direction: game-object -> posn
; Returns a unit vector in the forward direction of the game object.
```

Q2: The `asteroid` Struct

In the next step, let's enrich the game with asteroids of different sizes and different colors. To achieve this goal, we define the asteroid struct with two additional fields storing the radius and the color information:

```
(define-struct (asteroid game-object) (radius color)
  #:methods
  ...)
```

- The `radius` field stores a positive number representing the size of an asteroid.
- The `color` field stores the color of an asteroid. It is a string or an RGB color struct.

Your task is to implement the `radius` method and the `render` method for asteroids. The Asteroids library takes on the job of creating asteroids with varying radius and colors and will also take care of updating the asteroids state (i.e. you don't need an `update!` method for asteroids).

- The `render` method returns an image of the asteroid.
- The `radius` method should return the size stored in the radius field.

Note: The `asteroid` struct purposefully has both a `radius` field and a `radius` method. There isn't a name conflict here, because in order to access the property `radius` of an `asteroid` we use the accessor `asteroid-radius` which is distinct from the `radius` method which we'd call by saying: `(radius an-asteroid)`.

Q3: The `missile` Struct

Finally, finish the base game by implementing the `missile` object and limiting a missile's `lifetime` on the screen. A missile self-destructs when it reaches the end of its lifetime.

```
(define-struct (missile game-object) (lifetime)
  #:methods
  ...)
```

- The `lifetime` field stores an integer between 0 and 100 representing the current remaining life of a missile. (**Hint**: There is nothing to be defined for the lifetime of the missiles. `lifetime` is a field, not a method like `radius`.)
- The `update!` method should decrement the lifetime of a missile by 1 if its remaining lifetime is positive before the decrement. **When** the lifetime of a missile is already 0 (so decrementing it would have ended up with a negative number), call `destroy!` to destroy the missile. It does NOT need to move the missile.
- The `render` method returns an image of the missile.
- The `radius` method returns the size of the missile. It should be a positive number.

Note that the game engine will take care of actually moving the missile across the screen (which it will NOT do for the `heat-seekers` in Exercise 8).

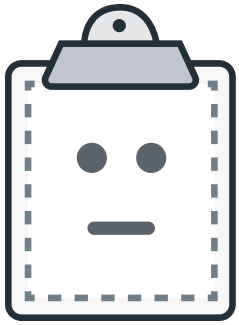
There are some built-in tests that we've provided, but also consider other ways you might test your completed work.

> **Note**: Several of the built in tests will fail because they're for stuff we'll work on in the second half of this assignment in Exercise 8. Feel free to either comment them out or ignore them…but don't forget about them for the Exercise!

Make sure to actually try playing the game (remember, you can just enter `(asteroids)` in the interactions window)! Does it work as expected? If everything was defined correctly than you should have a working simple asteroids complete with moving asteroids, moving and missle-ing and losing player, and destroyable asteroids.

Getting Credit for Your Work

Everyone must submit their `tutorial_8.rkt` file to Canvas regardless if you're in-person or remote. As long as you complete 80% of the tasks, you will receive a 100 for the assignment, but remember you'll need to complete the parts you didn't finish before doing Exercise 8. You'll receive automated feedback on this tutorial from the autograder to make sure you're on track for Exercise 8. It will run once an hour starting on Tuesday afternoon at 5pm. **If you receive less than a 100 but higher than a 80 on the tutorial submission, your Canvas assignment grade will be manually updated on Thursday morning**. If you score below a 80, your grade will be final. You can resubmit up until



Preview Unavailable

tutorial_8.rkt

Download

(https://canvas.northwestern.edu/files/17887458/download?download_frd=1&verifier=eq6EzM9T76R6tH9XrC4Tpz5tYVEcfkZ3AgyKFhMe)

(https://canvas.northwestern.edu/courses/201068/modules/items/2863026)

(https://canvas.northwestern.edu/courses/201068/modules