Asteroids

Data and architecture design

By Matej Havranek & Teddy Gyabaah

Introduction and Rules

The player controls a spaceship which can move around the screen. The spaceship has to avoid the asteroids that pop up on the screen, and can destroy them by shooting at them, at which point they split into smaller asteroids.

Each asteroid has a chance of having an enemy spaceship inside. When freed by destroying the asteroid, it flies around and tries to shoot the player.

The goal of the player is to gain as much points as possible by destroying asteroids and killing the enemies in the game.

Design

Data structures

The full state of the game is represented using a GameState object that contains three fundamental states of our game, namely the state of the Game screen, the stats situation and the actual objects involved in the game in a precise moment.

```
data GameState = State { gameScreen :: GameScreen,
```

```
                          gameStats :: GameStats,
                          gameObjects :: GameObjects }

data GameStats = Stats { name :: String,
                         score :: Int }

data GameObjects = GameObjs { asteroids :: [Asteroid],
                              missiles :: [Missile],
                              enemies :: [Player],
                              player :: Player }
```

The player

The player type in our Asteroids game will be represented by an enumeration, in this definition we define that the player can be either controlled by a Human or by the Computer.

```
data PlayerType = Human | Computer
```

The actual player is modeled by the Spaceship data type that contains all the properties to handle it.

```
data Spaceship =   Ship { shipPosition :: Point,
                          shipSpeed :: Vector,
                          angle :: Float,
                          boundingBox :: BoundingBox,
                          bulletTimeout :: Int,
                          playerType :: PlayerType }
```

As we can see by design the enemies and the player differ only in the player type. The player has to defeat enemies (by launching missiles at them) and destroy asteroids while

avoiding being hit by the asteroids or enemy missiles to get a higher score. Missiles are defined as follows:

```
data Missile = Msl { lifetime :: Int,
                     position :: Point,
                     boundingBox :: BoundingBox,
                     speed :: Vector }
```

And the function that launches them is

```
-- Create a missile heading in the direction of the spaceship
shoot :: Spaceship -> Missile
```

Player movement

Asteroids is a game that has independently moving particles. This means that the movement is continuous and can vary in direction and speed for every object on screen. The player can decide where to move on the screen.
The move function takes the position of an object and a speed vector, and returns the new position of the object.

```
-- Moves an object in the direction of the specified vector
move :: Point -> Vector -> Point
```

Player adversary

As mentioned before the difference between the player and the enemies dwells in the PlayerType. The enemies movements are controlled by the AI function that will decide the speed and the direction of the ship in order to obtain a clear shot on the player.

Another fundamental adversary of the user are the Asteroids that float around the screen modeled using the following structure:

```
data Asteroid = As { size :: Int,
                     position :: Point,
                     boundingBox :: BoundingBox,
                     speed :: Vector }
```

An important property of Asteroids and Spaceships is the BoundingBox. The bounding box is a rectangular border that fully encapsulates an object. Ii is important because it allows us to detect collisions between the asteroids and the player by simply checking whether their bounding boxes intersect.

```
-- Checks whether two bounding boxes collide
checkCollision :: BoundingBox -> BoundingBox -> Bool
```

Interface

The players will interact with the game through keyboard keys to Turn Left/Right, Thrust forward and Shoot. Mouse controls will also be available, making the player always face in the direction of the mouse, thrust forward by pressing the left mouse button and shoot a missile with the right mouse button. The game will feature a black background with asteroids being polygons, spaceships being arrows and missiles being short lines that move across the screen. All objects will be represented by a white outline.
Propelling will cause more lines to appear behind the ship, these could be animated
We have designed four screens:

- MainScreen where the player can insert his name and start the game
- PauseScreen that appears when the user decides to pause the game and shows the current score
- GameScreen where the Asteroids game happens

- EndScreen showing a Game Over message and player's score

All these screens will be rendered by the view function

```
view :: GameState -> IO Picture

view gstate = pure $ case gameScreen gstate of
                MainScreen -> renderMainScreen gstate
                PlayScreen -> renderPlayScreen gstate
                PauseScreen -> renderPauseScreen gstate
                EndScreen -> renderEndScreen gstate
```

Implementation of the Minimum Requirements

- **Player:** The player can control his spaceship
- **Enemies:** The AI controls all the other spaceships in the game. It tries to shoot missiles at the player and moves around.
- **Randomness:** New asteroids spawn at random locations on the screen. Big asteroids have a random chance of having an enemy hidden "inside" and releasing it when they are destroyed.
- **Animation**: Flames behind the ship are animated. There can also be an animation of an asteroid being destroyed.
- **Pause**: In our game Pause is implemented by a Pause screen that appears whenever the user press the key 'P'
- **IO**: The top three highscores are saved as name-score pairs to a file and loaded from it whenever the game is run.

Implementation of the optional Requirements

- **Use JSON to save the full game state:** The GameState class and all classes contained within will be made serializable into JSON. There will be a “save” option, allowing the player to save the current game to a file and load it later.
- **Mouse input:** The player can either use the keyboard or the Mouse to move the spaceship and shoot

Complete model description

```
--------------------
------ Model -------
--------------------

--------------------
-- Defines the game screen we are currently on
--------------------
data GameScreen = MainScreen | PlayScreen | PauseScreen | EndScreen

--------------------
-- Defines whether an entity is controlled by a human or by AI (further AI
types can be added)
--------------------
data PlayerType = Human | AI

--------------------
-- Defines the bounding box for an entity, simplifies collision detection
--------------------
data BoundingBox = Bound { leftUpper :: Point,
                           rightBottom :: Point }
```

```haskell
--------------------
-- Defines a spaceship (player or enemy) and its properties
--------------------
data Spaceship = Ship { shipPosition :: Point,
                        shipSpeed :: Int,
                        angle :: Float,
                        shipBoundingBox :: BoundingBox,
                        bulletTimeout :: Int,
                        playerType :: PlayerType }

--------------------
-- Defines an asteroid and its properties
--------------------
data Asteroid = As { asSize :: Int,
                     asPosition :: Point,
                     asBoundingBox :: BoundingBox,
                     asSpeed :: Vector }

--------------------
-- Defines a missile (a single shot) and its properties
--------------------
data Missile = Msl { mslLifetime :: Int,
                     mslPosition :: Point,
                     mslBoundingBox :: BoundingBox,
                     mslSpeed :: Vector }

--------------------
-- Contains the players name and score in the current game
--------------------
data GameStats = Stats { name :: String,
                         score :: Int }

--------------------
-- Encapsulates all the game objects (spaceships, asteroids, missiles) into
one container
--------------------
data GameObjects = GameObjs { asteroids :: [Asteroid],
                              missiles :: [Missile],
                              enemies :: [Spaceship],
                              player :: Spaceship }

--------------------
```

```haskell
-- Contains all information about the game and its state
--------------------
data GameState = State { gameScreen :: GameScreen,
                         gameStats :: GameStats,
                         gameObjects :: GameObjects }
--------------------
-- Returns the initial state of the game
--------------------
initialState :: GameState

--------------------
-- Creates a new ship (player/enemy) at the given point
--------------------
createShip :: PlayerType -> Point -> Spaceship

--------------------
-- Creates an asteroid of the given size at the given point
--------------------
createAsteroid :: Int -> Point -> Vector -> Asteroid

--------------------
-- Create a missile heading in the direction of the spaceship
--------------------
createMissile :: Spaceship -> Missile

--------------------
-- Adds a missile to the list of missiles in the gameObject
--------------------
addMissile :: GameObjects -> Missile -> GameObjects

--------------------
-- Sets the ships speed to propel it forward
--------------------
propelShip :: Spaceship -> Spaceship

--------------------
-- Moves an object in the direction of the specified vector
--------------------
move :: Point -> Vector -> Point

--------------------
-- Checks whether two bounding boxes collide
```

```haskell
--------------------
checkCollision :: BoundingBox -> BoundingBox -> Bool

--------------------
-- Changes the spaceship angle and rotates the bounding box (graphic
rotation is handled separately)
--------------------
rotateShip :: Spaceship -> Float -> Spaceship

--------------------
-- Whenever an asteroid is hit, explode it into smaller asteroids (or none
if its too small)
--------------------
explodeAsteroid :: Asteroid -> IO [Asteroid]

--------------------
-- Updates the position of the asteroids
--------------------
updateAsteroids :: [Asteroid] -> [Asteroid]

--------------------
-- Updates the position of the missiles and remove them if their lifetimes
are over
--------------------
updateMissiles :: [Missile] -> [Missile]


--------------------
------- View -------
--------------------

--------------------
-- Main rendering method. Based on game state determines the correct screen
to render
--------------------
view :: GameState -> IO Picture

--------------------
-- Renders the first screen with game logo and player name input
--------------------
renderMainScreen :: GameState -> Picture
```

```haskell
--------------------
-- Renders the game itself - player, enemies, asteroids, missiles and score
--------------------
renderPlayScreen :: GameState -> Picture

--------------------
-- Pause screen - shows the score and a message that the game is paused
--------------------
renderPauseScreen:: GameState -> Picture

--------------------
-- End screen - displays "game over", players name and final score
--------------------
renderEndScreen :: GameState -> Picture

--------------------
-- Converts current game stats (name and score) into a picture
--------------------
getStatsText :: GameStats -> Picture


--------------------
---- Controller ----
--------------------

--------------------
-- Update loop
--------------------
step :: Float -> GameState -> IO GameState

--------------------
-- Handle user input
--------------------
input :: Event -> GameState -> IO GameState

--------------------
-- Handle keypresses
--------------------
inputKey :: Event -> GameState -> GameState

--------------------
-- Handle mouse events (for the mouse input bonus)
```

```haskell
--------------------
inputMouse :: Event -> GameState -> GameState

--------------------
-- AI determines the actions for enemy spaceships
--------------------
updateAI :: GameObjects -> GameObjects

--------------------
-- Performs collisions on all game entities
--------------------
performCollisions :: GameState -> GameState
```