Kristi Nikolla
BU CS 520
Project Report

Type Invaders

Repository: https://github.com/knikolla/TypeInvaders

Type Invaders is a game similar to Space Invaders. You control a spaceship using the Left and Right arrow keys, and can fire bullets using the Up key of the keyboard. On the top of the screen there are several rows of alien enemies which also fire bullets at random intervals. If one of your bullets hits an enemy, it is destroyed. If one of the enemy bullets hits you, you are destroyed and the game is over. The game is won when all the enemies are destroyed.

The game was initially implemented entirely in JavaScript, using the Easel.js game framework for graphics rendering. The way this framework works is by attaching graphical objects to a stage object, and calling the update method every time the screen needs to be updated. Also, we can pass a function reference to Easel.js creating a timer which gets called periodically (in our case 20 times per second).

The program is structured into 3 files.

- game_sats.sats - Header file with function specifications
- game_cats.js - Implementation of the JS functions wrapping Easel.js
- game.dats - Implementation of all the other functions in ATS.

Wrapping JS

To communicate with Easel.js I needed to wrap the fundamental functions of creating a stage, updating the stage and adding and removing objects from the stage. The following is the wrapper specification I created around these fundamental functions. The game_tick function gets called 20 times per second from Easel.js.

```
abstype gameobject

// Init
fun init_stage(): void = "mac#"
fun init_resources(): void = "mac#"

// Stage
fun stage_add(gameobject): void = "mac#"
fun stage_remove(gameobject): void = "mac#"
fun stage_update(): void = "mac#"

fun game_tick(dt: int): void = "mac#"
```

GameObject is an abstract type which gets passed to the framework to be drawn in the screen. Init Resources is a helper function required to load the images representing the player and the aliens, while Init Stage initializes the stage object.

We can't create a GameObject directly in ATS, therefore we needed to implement the function to create the players and the enemies in JS. Since GameObject is an abstract type, we don't really care about its implementation because JS will handle that part behind the scenes. So functions which manipulate the gameobject (such as Get X and Set X) are also implemented in JS. In ATS, we only need the reference to such an object and pass it to these functions.

```
function init_player(x, y) {
      player = new createjs.BitmapAnimation(spritesSpaceship);
      player.gotoAndPlay("idle");
      player.currentFrame = 0;
      player.x = x;
      player.y = y;

      return player;
}
```

There are several such functions: to initialize the player GameObject, enemy GameObject and bullet GameObject. These are all defined as GameObject types to remove the need to define the same range of functions over all these similar types, since they are treated the same way by the framework.

```
fun init_player(x: int, y: int): gameobject = "mac#"
fun enemy_create(x: int, y: int): gameobject = "mac#"
fun player_create_bullet(p: gameobject): gameobject = "mac#"
fun enemy_create_bullet(gameobject): gameobject = "mac#"
```

We needed a quick and easy way to store and access the enemies and bullets present in the screen at any given time. To do this, I created an abstract type called store which I implemented as nothing more than a JavaScript array, and then wrapped the JS function over the array.

```
abstype store
fun store_init(): store = "mac#"
fun store_add(store, gameobject): void = "mac#"
fun store_get(store, int): gameobject = "mac#"
fun store_remove(store, int): void = "mac#"
fun store_size(store): intGte(0) = "mac#"
```

Initialization

To store the player, enemy store, and other necessary variables I used references inside a local space. This ensures that these variables are only accessed through the appropriate getter and setter functions which are implemented inside the local space. Player is an optional type, because many operations would fail without the player present, ex. after death. The init function calls the appropriate initializations defined above, filling the enemy store with the defined number of enemies. Since writing games requires many constant values (player speed, size, screen size, etc.), I filled the top of the sats header file with all the necessary value definitions.

```
local
  val player = ref(None)
  val p_cooldown = ref{int}(0)
  val e = store_init()
  val enemies = ref{store}(e)
  ...
  val pb = store_init()
  val player_bullets = ref{store}(pb)
  ...
  val e_dir = ref{int}(1)
  fun init(): void = …
in
  val () = init()
end
```

Game Tick

The Game Tick function is called 20 times per second with dt as an argument. Delta Time is the time in milliseconds that has passed from the last time Game Tick was called, and is useful for keeping track of time. The Game Tick function does anything only if a player is returned from the Get Player function. If there is a player it calls update on the player, enemies, and enemy bullets and player bullets, and then calls stage update which updates the screen.

```
implement game_tick(dt) =
  let
      val opt = player_get()
  in
      case opt of
      | None() => ()
      | Some(p) =>
      let
        val () = enemy_update(dt)
        val () = player_update(p, dt)
        val () = player_bullets_update(dt)
        val () = enemy_bullets_update(dt)
      in
        stage_update()
      end
  end
```

Enemy update checks if any of the enemies reached the border of the screen, turning them if necessary, then checks if they can shoot and moves them in the specified direction.

```
implement enemy_update(dt) =
  let
      fun boundaries{n:pos}(s: store, n: int(n), i: natLt(n)): void = ...
      fun shoot(s: store, n:int, dt: int): void = ...
      fun move{n:pos}(s: store, n: int(n), i: natLt(n), d: int): void = ...

      val e = enemies_get()
      val s = e.size()
  in
      if s > 0 then
        let
          val () = boundaries(e, s, 0)
          val d = enemies_direction()
          val () = shoot(e, s, dt)
        in
          move(e, s, 0, d)
        end
    else ()
  end
```

Player update is similar, but it also check for collision with enemy bullets. In that case it call Player Death which alerts the user that the game is over and updates the player reference to None. Therefore the next time Game Tick is called player will be None and the loop will do nothing, essentially stopping the game.

```
implement player_update(p, dt) =
{
  val () = player_cooldown(dt)
  val () = player_input(p)

  val b = enemy_bullets_get()
  val n = b.size()
  val i = object_store_collision(p, PLAYER_HALFSIZE, b, BULLET_HALFSIZE, n)

  val () = if i > 0 then player_death(p)
}
```

Checking if the enemies or the player can shoot are both similar. There is a function called Player Can Fire or Enemy Can Fire which return an optional type. They return Some(1) if the cooldown of either player or enemy is zero, incrementing the cooldown back again. In the case of the player the cooldown is always incremented by 1000 milliseconds, therefore the player can shoot exactly 1 bullet per second. The enemy cooldown is random but disproportional to the amount of enemies on the screen. The less enemies there are, the smaller the maximum random cooldown. This makes the game harder as more enemies are destroyed.

Player Input checks if Left, Right, or Up is called, and calls the appropriate functions called Translate (for moving) or Fire.

Object Store Collision is an abstraction which checks for intersection between a gameobject and a store of gameobjects. On collision it returns 1 and removes the gameobject from the list which collided. It uses the Rectangle Intersection algorithm.

Both bullet update functions essentially only move each of the bullets in the required direction (up or down) and remove them from the store if they are outside of the boundaries. The Player Bullet Update function though, has an additional function for checking the collision between each bullet and the store of aliens. If there was a collision it removes the bullet from the store and check the size of the store of enemies. If it was 1 (and now is zero) it congratulates the user for winning the game. An additional consideration was that these loops have been implemented starting from n-1 and going to zero, because there is the possibility of removing elements from the array inside the loop. This simplifies the indexing if such case occurs. Below is the code for Player Bullets Update.

```
implement player_bullets_update(dt) =
  let
      fun move{n:pos}(s: store, n: int(n), i: natLt(n)): void =
        let
          val e = s.get(i)
          val () = e.translate(BULLET_HALFSIZE, 0, ~BULLET_SPEED)
          val () = // remove the bullet if it exited the screen
      if e.x + BULLET_HALFSIZE < 0 then s.remove(i)
          val es = enemies_get()
          val en = es.size()
          val co = object_store_collision
      (e, BULLET_HALFSIZE, es, ENEMY_HALFSIZE, en)
          val () =
      if co = 1 then // if there was a collision
      {
              val () = s.remove(i) // destroy the bullet
              val () = stage_remove(e)
              val () = if en = 1 then congrats_alert()
                 // if this was the last enemy, game is won
      }
        in
          if i > 0 then move(s, n, i - 1)
          else ()
        end

      val s = player_bullets_get()
      val e = enemies_get()
      val n = s.size()
  in
      if n > 0 then move(s, n, n - 1)
      else ()
  end
```