Hugh L
4/27/2019
Functional Programming in OCaml
Project Report – Space Destroyer (Space Invaders clone)

My project goal was to recreate the classic Space Invaders game in OCaml using immutable data structures and using non-imperative (pure) functions/code. The Tsdl library was used for this program for graphics and UI as well as movement of the sprites inside the game. I ran into a lot of difficulties while implementing this game, and learned a lot which will be discussed later on.

First, I will talk about how the game itself was implemented. I started with the sample code given to us for the Tsdl library. I took some time to look through the sample code and did my best in understanding what each function does and what each line of code was doing. I kept the State module and implemented the actual game functions within the module. I immediately declared the screen width and screen height in the module since that would be a static value. Then a record called Sprite was created as a template for all sprites in the game, with an x and y coordinates of type float, and a rectangle of type Sdl.rect.

I decided to make a record named “t” to store the actual game state itself. It has the default screen width/height, the player which is of type sprite (as mentioned before), and enemies and bullets were each a list of sprites. I also have a enemy offset that is of type float * float, to help determine the coordinates of the enemies inside the list. Lastly I have a bool called enemy_forward to help figure out if the enemies should be moving right or left on the screen.

The make function is called upon the creation of the game. It initializes all the starting values for the game, such as player position, enemy positions, an empty bullet list, etc. Since the enemies are stored in a list rather than an array matrix, I had to have a function to make their coordinates display them as a grid which is what the grid_coords function is used for.

Next, are the push and shoot functions. Both of these functions take a State.t as an argument and return type State.t as well. Push is responsible for the player ship movement from left to right as well as keeping the player ship from going off the screen. The shoot function adds a bullet to the bullet list, with the starting position being the same as the player position on the screen.

The last function within the state module is the most important function. Since update is called on every loop, the enemy movements is implemented here by determining the enemy offset x, y coordinates, and the boolean of enemy_forward. The enemy moves from left to right, then down, and right to left, and repeat until it hits the bottom of the screen. The update function also moves the bullets

upwards towards the enemies (to the top of the screen), by using the List.map function to update the bullet records with new y coordinates. Lastly, List.filter function is used to remove bullets that have reached the top of the screen, as well as removing bullets/enemies that have collided with each other. Update also takes a State.t as an argument and returns type State.t with updated values for the state.

Inside the draw function, I created a generic draw_sprite function that would take a sprite or sprites, and draw them into the game window. For the draw_sprite function it has the optional argument offset (to help determine where to draw enemy sprites), width and height of the sprite picture, and the sprite itself. After implementing the generic draw_sprite function, drawing the players and enemies onto the screen was a matter of passing the sprites itself into the function. List.iter was used to apply the draw_sprite function to the enemies and the bullets.

The loop function inside the run function was modified slightly to include a call to the shoot function when the user presses space. All the functions being called from state in the pattern matching pass a State.t (as well as other arguments if needed) and return a State.t for use. The State.t that is returned is used in the update function to update the game. Lastly, the draw function is called to draw the new updated state. These functions are called on each loop, and each loop represents one “frame” of the game, until the game ends.

The function get_event () was largely untouched, except for removing the up and down events, and adding a shoot event. The main function is also largely untouched except for removing unused arguments, in this case the screen width and height, which is already declared in the state module.

When beginning to implement this program, I had difficulty wrapping my head around how the sample program worked, and how I could modify it to include multiple “objects” instead of just one. I took the suggestion of creating a record to hold everything in the game state, and things slowly started to click. Figuring out how to use the SDL library proved difficult as well but luckily the sample program provided all that was needed in order to create the actual game and display the “objects” of the game. If I didn’t have the sample code, I don’t think I would have been able to complete the project.

The lack of resources and references specific to OCaml as well as the vague error messages made it hard to figure out where certain errors were coming from and difficult to debug. But in my opinion the most difficult issue I came across was switching my brain from object oriented programming logic into functional programming logic since most of my experience has been with C++, it was challenging to switch my way of thinking.

First and foremost, the major thing that will stick with me from doing this project, is the importance of types. In C++, everything is mutable, and we can modify specific parts since everything is treated like an object. In OCaml, it doesn't work that way, so I opted to make each function take in the entire game State.t and return the same State.t in order to pass along and modify certain arguments which took me a while to figure out. The lectures in class begin to make more sense after working on the project, I can see why the errors are often vague and can seem unhelpful, since the compiler is doing a lot of inferring of the code determining the types expected, etc. Since OCaml code seems very rigid, for the lack of a better word, I can see why functional programming is easier to walk through though the syntax can be difficult, and also why programs built this way are less problematic/inclined to fail.

My last comments on this project is that I feel I might have chose something too difficult for my level. I spent more time than I planned on building this project, and a lot of it was on specific OCaml functions and errors. Game design itself is interesting, but as mentioned before, if sample code was not provided, I may have spent the whole time trying to even implement the run + get_event + main functions that were given to us. I mistakenly assumed that it would not be too difficult with the help of the Sdl library, but obviously I was wrong. But in the end, I'm glad I was able to complete the core of the game, and I'm actually more motivated to learn more OCaml after the semester is over. This was definitely one of the more frustrating projects I've had, but also one of the more fun ones.