Joshua Brancale

CSCI Functional Programming in Ocaml

Professor Alexey Nikolaev

Project Report

My project is a version of the popular “snakes” game. The program runs a game where a player, using the directional arrow keys on the keyboard, controls the movement of a snake represented by a square on the screen. The goal of the player is to collect as many “points” (also represented by a white square) as possible. As the player collects more points, the tail of the snake grows longer. If the player should run the head of the snake into the tail, you lose and the game ends.

I started by trying to learn tsdl by modifying part of the example code in an attempt to understand what parts of the code did what exactly. It was difficult to find any tutorials for tsdl in ocaml, so this was essentially my only way of learning. I performed a lot of guess-and-check in order to get an idea about many things. In the end, I still utilize a big chunk of the example code in my final project, largely untouched, as that code was still a necessary part in order to get the program to work. What I did change was mostly situated in the State module, and I did some modifications to the rendering as well, which I will cover in more detail later.

In my first attempt, I tried to do an object oriented approach where I separated the snake, the point, and the field into different modules with their own functions and data members, however, this quickly became crowded and complicated, resulting in lots

of confusion and errors. After lots of thought I scrapped that idea and instead decided to combine all the different modules into a single State module for simplicity, and that helped move along progress considerably. The current State module contains all parts of the snake along with helper functions, the point location, x and y velocity, etc. I kept many of the names of functions the same as in the practice code for simplicity, but none of the functions in the State module have retained their original functionality.

Data members in the State module now include

- The snake head of type (float * float)
- The snake tail of type (float * float) list
- The point of type (float * float)
- The horizontal velocity vx of type float
- The vertical velocity vy of type float

Functions in the State module that have been modified include:

- make - modified to handle the new data members
- push - now only pushes at a constant velocity in a singular direction (up, down, left, right) at a time.
- update - keeps track of collision, point consumption, and boundaries. If the snake reaches the boundary of the field, it wraps around to the other side. If the snake collides with its tail, the game ends. If the snake collides with a point, it consumes it, grows, and a new point is randomly generated within the space of the field.

Additional functions added (helper functions)

- pop_back - removes the last element of a list (our snake tail is implemented as a list)
- snake_grow - if the snake should not grow, calls pop_back on the snake tail
- point_collision - checks if the snake head collides with a point
- collision - check if the snake head collides with a single node of the tail
- tail_collision - recursively checks if any of the snake tail nodes has collided with the snake head

Aside from the State module, most of the code is taken directly from the practice example. Exceptions include in the () function, where the image file was changed, and in the draw function, where a few changes were made - Two new helper functions were created, draw_point and draw_tail. Draw point accepts a (float * float) and uses it as coordinates for where to draw the position of a singular point on the field. Draw_tail recursively does the same thing as draw_point but for every node in the list (the snake tail). The draw function is modified to call draw_point for the snake head and the point, and draw_tail for the snake tail.

In run the delay was changed to 15 milliseconds, otherwise it functions the same, applying the push function to State as necessary according to get_event, and updating and drawing on each loop through.