Midi Maze: Simulation and Data

Data

It's easier to understand the simulation if you understand the data. The format of the input data and how to read it is covered in the input document. The format of the output data is covered in the output document.

Input Data

The stored data we initialize from the input consists of:

- The code (to be read and stored in a 16 bit unsigned short)
- The initial X and Y positions (double)
- The initial direction (double)

The code contains the color, speed, and type of an object. It also contains some error detection data to make sure that the code is valid.

Internal Data

The simulation also keeps the simulated elapsed time for each run. This is different than the real time of the run, though in graphics mode they will be close.

- Elapsed time, in seconds (double)

Constants

We also keep the value of how much time each clock tick represents. Only the simulation code should know this value and it should pass it as needed, notably to output.

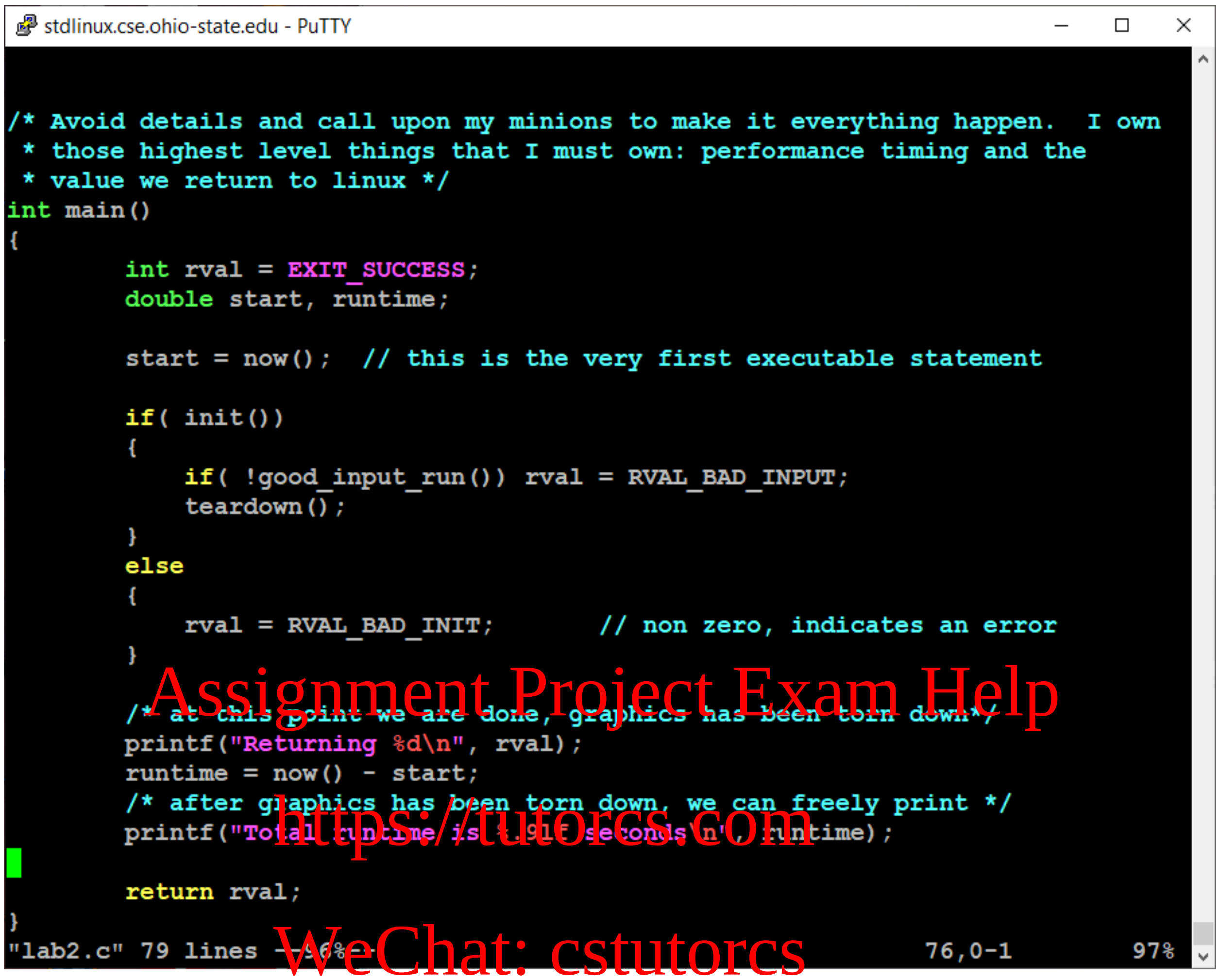
- DELTA_T (a manifest constant owned by the simulation and passed where needed). It has the value (1.0 / 32.0). So our clock rate is 32 ticks per second.

Incidentals

You might want to store the two parts of any proposed new position as well.

Simulation

The lab code should initialize anything that needs it, such as checking the real-time clock and graphics, before dealing with input. If those initializations fail, main should return the value 1 to the system. To make life easier, the code shown here is the main function of the reference code and you can use it.



```
/* Avoid details and call upon my minions to make it everything happen.  I own
 * those highest level things that I must own: performance timing and the
 * value we return to linux */
int main()
{
        int rval = EXIT_SUCCESS;
        double start, runtime;

        start = now();  // this is the very first executable statement

        if( init())
        {
            if( !good_input_run()) rval = RVAL_BAD_INPUT;
            teardown();
        }
        else
        {
            rval = RVAL_BAD_INIT;       // non zero, indicates an error
        }
        /* at this point we are done, graphics has been torn down*/
        printf("Returning %d\n", rval);
        runtime = now() - start;
        /* after graphics has been torn down, we can freely print */
        printf("Total runtime is %.9lf seconds\n", runtime);

        return rval;
}
"lab2.c" 79 lines --96%--                                   76,0-1        97%
```

The two now calls are used to access the high precision real-time system clock so that we can time how long it takes our program to run. Performance awareness is a key teaching in this class.

The simulation is driven out of the input loop that reads all of the data.

For each object we read in, the simulation will run a loop to output the object data and update the object data. The loop should only run on valid positions. The first thing the loop does is output the object data. Then it moves time forward and computes the proposed new position for the object. The loop should not change the X,Y values to an invalid position. If the next position is invalid, the loop stops and the final output is done from the last valid position.

In short, go in the direction it's pointed until it hits a wall.

Your code should have a separate C code file that owns the simulation loop and all helper functions that know how to compute next values. It also owns code that determines if a proposed position is valid.

Your sim code will have to ask the library about the validity of any particular position via the **`midi_touches_wall`** function.

Equations of motion

The X and Y axes are oriented in the normal way with 0,0 at the lower left corner of the playing field.

Angles are measured with 0 degrees meaning “heading due north” or “moving straight up.” 90 degrees means “moving to the right.” This is not how mathematics usually does it, but it is how maps treat angles.

The C math library has both sin and cos functions. They take angles in radians. Our angles are measured in degrees. The #define for M_PI is in the pi.h file that is part of the libmidi.zip file.

Motion requires velocity and time. The code data item has the speed encoded into it. Use a function in your bits.c file to extract the speed from the code. Our time is a fixed amount, one thirty second of a second. This time value should be a constant known only to the simulation code and passed where need (to output most notably).

Putting that all together, the change in position in X or Y is given as either

```
speed * time * sin(direction)
speed * time * cos(direction)
```

Due to the way we orient angles, X uses sin and Y uses cos. If you add those changes in X and Y to the current values of X and Y you get the proposed new values of X and Y.

Your code should avoid changing the current position to an invalid position. Use the **midi_touches_wall** function to tell if a position is OK. It returns true if the position given touches a wall and thus is invalid.