Meteor Strike

Rust Mini Midterm Project

Element System Programming

Software Engineering Program,

Department of Computer Engineering,

School of Engineering, KMITL

By

67011247 Payut Kapasuwan

67011352 Theepakorn Phayonrat

Game Concept

It's 20th April 2069, as you're driving along the road a mysterious looking meteor striked down. In shock, you drove toward the open field and that's how the story began...

Welcome to METEOR STRIKE game with low quality graphic based on Terminal, and coded in our beloved programming language "RUST". Your mission is to survive 60 second of HELL or DIE.

How to Play

In the terminal, you are the BLUE SQUARE while those meteors are the RED @ on the screen randomly spawn along the game. Your goal is to avoid all of the meteors.

Our game is simple, use the Universal movement keys (Aka, WASD) to move. You have 60 seconds to move and avoid those meteors or else it's GAME OVER. If you don't want to continue, you can press ESC to End the game.

Win/Lose Condition

Win

You survive 60 second without hitting any meteor. Your point will be a high score of 60 points.

Lose

You got hit by one of those meteors fallen down. Your point will be how long you had survived.

Code Explanation

The SetUp (Enum, Struct, Impl and other funtions)

Enum (Custom Types)

- enum Direction:

  ```
  enum Direction {
      North,
      South,
      East,
      West
  }
  ```

  - "North" will be used in impl Player when player moved North.
  - "East" will be used in impl Player when player moved East.
  - "South" will be used in impl Player when player moved South.
  - "West" will be used in impl Player when player moved West.

Struct

1. struct Player:

   ```
   struct Player {
       alive: bool,
       x: u16,
       y: u16,
       score: u16
   }
   ```

- "alive" is a Boolean type variable for checking if the player is alive or not.
- "x" is an unsigned type variable that represents the player's east and west movement.
- "y" is an unsigned type variable that represents the player's north and south movement.
- "score" is an integer variable that represents the player's score.

2. struct Meteor:

```
struct Meteor {
    x: u16,
    y: u16
}
```

- "x" is an unsigned type variable that represents the meteor's fall location.
- "y" is an unsigned type variable that represents the meteor's fall location.

Impl (Functions implement from struct)

1. impl Player:

- fn new: used to make new player and set: alive condition to true, score, and position.

```
fn new(max_x: u16, max_y: u16) -> Player {
    Player {
        alive: true,
        x: (max_x + 1) / 2,
        y: (max_y + 1) /2,
        score: 0
    }
}
```

- fn move_player: used to move the player by matching directions from enum Direction.

```
fn move_player(&mut self, direction: Direction, max_x: u16, max_y: u16) {
    match direction {
        Direction::North => {
            if self.y >= 1 {
                self.y -= 1;
            } else {
                self.y = 1;
            }
        }
        Direction::East => {
            if self.x <= max_x{
                self.x += 1;
            } else {
                self.x = max_x;
            }
        },
        Direction::South => {
            if self.y <= max_y {
                self.y += 1;
            } else {
                self.y = max_y
            }
        }
        Direction::West => {
            if self.x >= 1 {
                self.x -= 1;
            } else {
                self.x = 1
            }
        }
    }
}
```

- fn add_score: used to add player's score 1 point per seconds.

```
fn add_score(&mut self) {
    self.score += 1;
}
```

2. impl Meteor:

- fn new: used to make new meteors and random the spawn point of meteors.

```
fn new(max_x: u16, max_y: u16) -> Meteor {
    let mut rng = rand::thread_rng();
    let meteor_x = rng.gen_range(1..=max_x);
    let meteor_y = rng.gen_range(1..=max_y);
    Meteor {
        x: meteor_x,
        y: meteor_y,
    }
}
```

Other Functions

- fn show_entity: used to spawn player, meteors or other labels as a string.

```
fn show_entity(x: u16, y: u16, entity: &str, color: Color) {
    let mut stdout = stdout();
    stdout.execute(MoveTo(x, y)).unwrap();
    stdout.execute(SetForegroundColor(color)).unwrap();
    print!("{}", entity);
    stdout.execute(ResetColor).unwrap();
}
```

The Main Function

The SetUps in Main

1. The General SetUps

   - We declared a "stdout" variable to get access to the standard output and for the future to make it easier to write.
   - We then used the variable declared earlier to execute the crossterm and tell the program to hide the normal cursor.
   - After that, we use the command in the 3rd line to enable raw mode. Raw mode will stop the behavior of the normal terminal and start record our keystrokes and do what is programmed in the later part of the code.

   ```
   let mut stdout = stdout();
   stdout.execute(Hide).unwrap();
   terminal::enable_raw_mode().unwrap();
   ```

2. Screen, Player and Meteors SetUps

   - Screen Setups: Declare a size of a screen of the game.
   - Player SetUps: Create a player which spawn in the middle of the screen.
   - Meteors SetUps: Create a vector which record every meteors fallen down.

   ```
   let max_x: u16 = 35;
   let max_y: u16 = 21;

   let mut player = Player::new(max_x, max_y);

   let mut meteor_vec: Vec<Meteor> = Vec::new();
   ```

3. Time SetUps: Declare a timestamp for each of the function in a loop in the next part of code.

   ```
   let mut score_time = Instant::now();
   let mut summon_met_time = Instant::now();
   let mut met_time = Instant::now();
   ```

The main loop of the game

1. End Game Conditions

   - Check whether the player is alive or not. If not, the game ends with the score of time passed in seconds.

   ```
   if player.alive == false {
       stdout.execute(Clear(ClearType::All)).unwrap();
       show_entity(5, 10, "You DEAD ", Color::DarkRed);
       show_entity(5, 11, &format!("Score: {}", player.score), Color::White);
       break;
   }
   ```

   - Check whether the player win the game or not. If yes, the game ends with the highest score possible (60 points for the 60 seconds survived).

   ```
   if player.score == 60 {
       stdout.execute(Clear(ClearType::All)).unwrap();
       show_entity(5, 10, "You SURVIVED ", Color::Cyan);
       show_entity(5, 11, &format!("Score: {}", player.score), Color::White);
       break;
   }
   ```

2. Summon Meteors

   - Summon a meteor every 1 second and push the meteor data into the meteor vector created earlier.

   ```
   if summon_met_time.elapsed() >= Duration::new(1, 0) {
       meteor_vec.push(Meteor::new(max_x, max_y));
       summon_met_time = Instant::now();
   }
   ```

3. Entities Collision Checker

   - Check whether the player hit one of the meteor or not. If yes, the player turns dead, else, the game continue by rendering the meteors as red @ as normal.

   ```
   for met in &mut meteor_vec {
       if player.x == met.x && player.y == met.y {
           player.alive = false;
       }

       if met_time.elapsed() >= Duration::new(0, 0) && met_time.elapsed() <=
   Duration::new(1, 0) {
   ```

```
            show_entity(met.x, met.y, "@", Color::Red);
        }
        met_time = Instant::now();
    }
```

4. Show Player

    - Show player as a blue square.

```
show_entity(player.x, player.y, "■", Color::Blue);
```

5. Show Score

    - Show player's score as a white text on the top.

```
show_entity(0, 0, &format!("Score: {}", player.score), Color::White);
```

6. Movement Dectector

    - Detect keystrokes and move player like a normal game via WASD keys. Else if player press ESC key, the game will end by showing the player's score.

```
if let Ok(true) = event::poll(Duration::from_millis(1)) {
    if let Ok(event::Event::Key(KeyEvent { code, .. })) = event::read() {
        match code {
            KeyCode::Char('w') => {
                player.move_player(Direction::North, max_x, max_y);
            }
            KeyCode::Char('s') => {
                player.move_player(Direction::South, max_x, max_y);
            }
            KeyCode::Char('d') => {
                player.move_player(Direction::East, max_x,  max_y);
            }
            KeyCode::Char('a') => {
                player.move_player(Direction::West, max_x, max_y);
            }
            KeyCode::Esc => {
                stdout.execute(Clear(ClearType::All)).unwrap();
                show_entity(5, 10, "You QUIT", Color::DarkRed);
                show_entity(5, 11, &format!("Score: {}", player.score),
Color::White);
                break;

            }

            _ => {}
            }
        }
    }
```

7. Flush

    - This command help the output go to the terminal right away without being held waiting in buffer.

```
stdout.flush().unwrap();
```

8. Add Player Score

    - Add player 1 score each second pass.

```
if score_time.elapsed() >= Duration::new(1,0) {
    player.add_score();
    score_time = Instant::now();
}
```

9. Clear the Terminal

    - Clear the whole terminal and end the current loop to start rendering in the next loop.

```
stdout.execute(Clear(ClearType::All)).unwrap();
```

Link to the presentation