main.rs

```rust
mod draw;
mod food;
mod game;
mod snake;

use draw::BLOCK_SIZE;
use food::Food;
use game::{Color, Game};
use piston_window::*;
use snake::Snake;

fn main() {
    let mut window: PistonWindow = WindowSettings::new("Snake Game", [400, 400])
        .exit_on_esc(true)
        .build()
        .unwrap_or_else(|e| panic!("Failed to build PistonWindow: {}", e));

    let mut game = Game::new(
        Snake::new(
            (window.size().width / (BLOCK_SIZE * 2.0)) as i32,
            (window.size().height / (BLOCK_SIZE * 2.0)) as i32,
        ),
        Food::new(window.size()),
        window.size(),
    );

    while let Some(e) = window.next() {
        if let Some(args) = e.render_args() {
            game.draw_instructions(args);
        }

        if let Some(Button::Keyboard(_)) = e.press_args() {
            break;
        }
    }

    let mut x = Key::Q;

    while let Some(e) = window.next() {
        if let Some(Button::Keyboard(k)) = e.press_args() {
            if k != x {
                x = k;
            }
        }

        window.draw_2d(&e, |c, g, _| {
            clear(Color::BACKGROUND, g);
            game.draw(&c, g);
        });

        e.update(|args| {
            game.update(window.size(), args, x);
        });

        if game.over() {
            if let Some(args) = e.render_args() {
                game.draw_game_over(args);
            }
        }
    }
}
```

game.rs

```rust
use crate::food::Food;
use crate::snake::*;
use opengl_graphics::{GlGraphics, GlyphCache};
use piston_window::*;

#[non_exhaustive]
pub struct Color;

impl Color {
    pub const BACKGROUND: [f32; 4] = [0.3, 0.4, 0.2, 1.0];
    pub const SNAKE_BODY: [f32; 4] = [0.0, 0.0, 1.0, 1.0];
    pub const SNAKE_HEAD: [f32; 4] = [0.3, 0.6, 1.2, 1.0];
    pub const FOOD: [f32; 4] = [1.0, 0.6, 0.2, 1.0];
    pub const TEXT: [f32; 4] = [1.0, 0.99, 0.22, 1.0];
}

#[derive(Copy, Clone, Debug, PartialEq)]
pub struct Position {
    pub x: i32,
    pub y: i32,
}

#[derive(Debug, PartialEq)]
pub struct Game {
    pub snake: Snake,
    pub food: Food,
    pub window_size: Size,
}

impl Game {
    // make a new game
    pub fn new(snake: Snake, food: Food, size: Size) -> Game {
        Game {
            snake,
            food,
            window_size: size,
        }
    }
    // call a function to draw the instructions screen
    pub fn draw_instructions(&self, args: RenderArgs) {
        let mut gl = GlGraphics::new(OpenGL::V3_2);
        let font = include_bytes!("../assets/FiraSans-Regular.ttf");
        let mut glyphs = GlyphCache::from_bytes(font, (), TextureSettings::new()).unwrap();

        gl.draw(args.viewport(), |c, g| {
            clear(Color::BACKGROUND, g);
            text(
                Color::TEXT,
                (self.window_size.width / 25.0) as u32,
                "WASD or Arrow Keys: Move",
                &mut glyphs,
                c.transform
                    .trans(self.window_size.width / 4.0, self.window_size.height / 2.0),
                g,
            )
        })
        .expect("Failed to make end screen");

        gl.draw(args.viewport(), |c, g| {
            text(
                Color::TEXT,
                (self.window_size.width / 25.0) as u32,
                "Q: Pause",
                &mut glyphs,
                c.transform
                    .trans(self.window_size.width / 4.0, self.window_size.height / 2.35),
                g,
            )
        })
        .expect("Failed to make end screen");

        gl.draw(args.viewport(), |c, g| {
            text(
                Color::TEXT,
                (self.window_size.width / 25.0) as u32,
                "Esc: Quit",
                &mut glyphs,
                c.transform
                    .trans(self.window_size.width / 4.0, self.window_size.height / 1.75),
                g,
            )
        })
        .expect("Failed to make end screen");
    }

    // update the game by calling functions to move the snake
    // have the snake eat food and spawn new food
    pub fn update(&mut self, size: Size, args: &UpdateArgs, key: Key) {
        self.snake.update(size, args.dt, self.key_direction(key));
        self.window_size = size;
        if self.snake.position == self.food.position {
            self.snake.eat();
            self.food.spawn(size, &self.snake);
        }
    }

    // change the direction the snake is moving based on the players
    // keyboard input
    fn key_direction(&self, key: Key) -> Direction {
        return {
            match key {
                Key::Right | Key::D => Direction::Right,
                Key::Left | Key::A => Direction::Left,
                Key::Down | Key::S => Direction::Down,
                Key::Up | Key::W => Direction::Up,
                Key::Q => Direction::Still,
                _ => self.snake.direction,
            }
        };
    }

    // call functions to draw the snake and the food
    pub fn draw(&mut self, c: &Context, g: &mut G2d) {
        self.snake.draw(c, g);
        self.food.draw(c, g);
    }

    // return is the game is over or not (if the snake is dead)
    pub fn over(&self) -> bool {
        !self.snake.is_alive()
    }

    // draw the game over screen and show the final length of the snake
    pub fn draw_game_over(&self, args: RenderArgs) {
        let mut gl = GlGraphics::new(OpenGL::V3_2);
        let font = include_bytes!("../assets/FiraSans-Regular.ttf");
        let mut glyphs = GlyphCache::from_bytes(font, (), TextureSettings::new()).unwrap();

        gl.draw(args.viewport(), |c, g| {
            clear(Color::BACKGROUND, g);
            text(
                Color::TEXT,
                (self.window_size.width / 13.3) as u32,
                format!("Final Length: {}", self.snake.length).as_str(),
                &mut glyphs,
                c.transform
                    .trans(self.window_size.width / 4.0, self.window_size.height / 2.0),
                g,
            )
        })
        .expect("Failed to make end screen");
    }
}
```

snake.rs

```rust
use crate::draw::{draw, BLOCK_SIZE};
use crate::game::{Color, Position};
use piston_window::{Context, G2d, Size};
use std::collections::VecDeque;

const SNAKE_WAIT: f64 = 0.2;

#[derive(Debug, PartialEq, Copy, Clone)]
pub enum Direction {
    Left,
    Right,
    Up,
    Down,
    Still,
}

#[derive(Clone, Debug, PartialEq)]
pub struct Snake {
    pub position: Position,
    pub length: u32,
    pub direction: Direction,
    pub tail: VecDeque<Position>,
    is_alive: bool,
    wait: f64,
}

impl Snake {
    // make a new snake
    pub fn new(x: i32, y: i32) -> Snake {
        Snake {
            position: Position { x, y },
            length: 1,
            direction: Direction::Still,
            tail: vec![].into_iter().collect(),
            is_alive: true,
            wait: 0.0,
        }
    }

    // check if the position of the snake is out of bounds
    fn is_valid(&self, size: Size) -> bool {
        let x = self.position.x;
        let y = self.position.y;
        x >= 0
            && y >= 0
            && x <= (size.width / BLOCK_SIZE) as i32
            && y <= (size.height / BLOCK_SIZE) as i32
    }

    // move the snake in the direction it is facing
    pub fn mv(&mut self, size: Size, direction: Direction) {
        self.wait = 0.0;
        if !self.is_valid(size) {
            self.is_alive = false;
            return;
        }

        match self.direction {
            Direction::Left => {
                if direction != Direction::Right {
                    self.direction = direction;
                }
            }
            Direction::Right => {
                if direction != Direction::Left {
                    self.direction = direction;
                }
            }
            Direction::Up => {
                if direction != Direction::Down {
                    self.direction = direction;
                }
            }
            Direction::Down => {
                if direction != Direction::Up {
                    self.direction = direction;
                }
            }
            Direction::Still => self.direction = direction,
        }

        // Note: I am using 2 match cases here for visibilty (I could have put this in the one up above).
        match self.direction {
            Direction::Left => {
                if self.overlap_tail(self.position.x - 1, self.position.y) {
                    self.is_alive = false;
                    return;
                }
                self.position.x -= 1;
                self.tail.pop_back();
                self.tail.push_front(self.position);
            }
            Direction::Right => {
                if self.overlap_tail(self.position.x + 1, self.position.y) {
                    self.is_alive = false;
                    return;
                }
                self.position.x += 1;
                self.tail.pop_back();
                self.tail.push_front(self.position);
            }
            Direction::Up => {
                if self.overlap_tail(self.position.x, self.position.y - 1) {
                    self.is_alive = false;
                    return;
                }
                self.position.y -= 1;
                self.tail.pop_back();
                self.tail.push_front(self.position);
            }
            Direction::Down => {
                if self.overlap_tail(self.position.x, self.position.y + 1) {
                    self.is_alive = false;
                    return;
                }
                self.position.y += 1;
                self.tail.pop_back();
                self.tail.push_front(self.position);
            }
            Direction::Still => {}
        }
    }

    // return if the snake is over laping its tail
    fn overlap_tail(&self, x: i32, y: i32) -> bool {
        self.tail.contains(&Position { x, y })
    }

    // have the snake eat food updating the snakes length
    pub fn eat(&mut self) {
        match self.direction {
            Direction::Left => {
                self.tail.push_back(Position {
                    x: self.position.x + 1,
                    y: self.position.y,
                });
            }
            Direction::Right => {
                self.tail.push_back(Position {
                    x: self.position.x - 1,
                    y: self.position.y,
                });
            }
            Direction::Up => {
                self.tail.push_back(Position {
                    x: self.position.x,
                    y: self.position.y + 1,
                });
            }
            Direction::Down => {
                self.tail.push_back(Position {
                    x: self.position.x,
                    y: self.position.y - 1,
                });
            }
            Direction::Still => {}
        }
        self.length += 1;
    }

    // draw the snake
    pub fn draw(&self, c: &Context, g: &mut G2d) {
        draw(
            Color::SNAKE_HEAD,
            self.position.x as u32,
            self.position.y as u32,
            1,
            1,
            c,
            g,
        );
        for seg in self.tail.iter().skip(1) {
            draw(Color::SNAKE_BODY, seg.x as u32, seg.y as u32, 1, 1, c, g);
        }
    }

    // return if the snake is alive
    pub fn is_alive(&self) -> bool {
        self.is_alive
    }

    // move the snake after a set amount of 'wait time' so the snake isnt too fast
    pub fn update(&mut self, size: Size, dt: f64, direction: Direction) {
        self.wait += dt;
        if self.wait > SNAKE_WAIT {
            self.mv(size, direction);
        }
    }
}
```

food.rs

```rust
use crate::draw::{draw, BLOCK_SIZE};
use crate::game::{Color, Position};
use crate::snake::Snake;
use piston_window::{Context, G2d, Size};
use rand::{thread_rng, Rng};

#[derive(Debug, PartialEq)]
pub struct Food {
    pub position: Position,
}

impl Food {
    // make a new food
    pub fn new(size: Size) -> Food {
        Food {
            position: Position {
                x: thread_rng().gen_range(0..=(size.width / (BLOCK_SIZE * 2.0)) as i32),
                y: thread_rng().gen_range(0..=(size.height / (BLOCK_SIZE * 2.0)) as i32),
            },
        }
    }

    // spawn the food on the screen in a valid location
    pub fn spawn(&mut self, size: Size, snake: &Snake) {
        loop {
            let mut thing: bool = true;
            for i in snake.tail.iter() {
                if i == &self.position {
                    thing = false;
                    break;
                } else {
                    thing = true;
                }
            }

            if thing {
                break;
            } else {
                self.position = Position {
                    x: thread_rng().gen_range(0..=(size.width / (BLOCK_SIZE * 2.0)) as i32),
                    y: thread_rng().gen_range(0..=(size.height / (BLOCK_SIZE * 2.0)) as i32),
                };
            }
        }
    }

    // draw the food on screen
    pub fn draw(&self, c: &Context, g: &mut G2d) {
        draw(
            Color::FOOD,
            self.position.x as u32,
            self.position.y as u32,
            1,
            1,
            c,
            g,
        );
    }
}
```

draw.rs

```rust
use piston_window::types::Color;
use piston_window::{rectangle, Context, G2d};

pub const BLOCK_SIZE: f64 = 25.0;

// return input pixel to the size of a block
// (the head or 1 body segment of the snake)
pub fn to_block_size(size: u32) -> f64 {
    f64::from(size) * BLOCK_SIZE
}

// draw a rectangle on the screen with the parameters inputted
pub fn draw(color: Color, x: u32, y: u32, width: u32, height: u32, c: &Context, g: &mut G2d) {
    rectangle(
        color,
        [
            to_block_size(x),
            to_block_size(y),
            to_block_size(width),
            to_block_size(height),
        ],
        c.transform,
        g,
    );
}
```