Dealing Damage (and taking some!)

About this tutorial

This tutorial is free and open source, and all code uses the MIT license - so you are free to do with it as you like. My hope is that you will enjoy the tutorial, and make great games!

If you enjoy this and would like me to keep writing, please consider supporting my Patreon.

Now that we have monsters, we want them to be more interesting than just yelling at you on the console! This chapter will make them chase you, and introduce some basic game stats to let you fight your way through the hordes.

Chasing the Player

The first thing we need to do is finish implementing `BaseMap` for our `Map` class. In particular, we need to support `get_available_exits` - which is used by the pathfinding.

In our `Map` implementation, we'll need a helper function:

```
fn is_exit_valid(&self, x:i32, y:i32) -> bool {
    if x < 1 || x > self.width-1 || y < 1 || y > self.height-1 { return false; }
    let idx = self.xy_idx(x, y);
    self.tiles[idx as usize] != TileType::Wall
}
```

This takes an index, and calculates if it can be entered.

We then implement the trait, using this helper:

```
fn get_available_exits(&self, idx:usize) -> rltk::SmallVec<[(usize, f32); 10]> {
    let mut exits = rltk::SmallVec::new();
    let x = idx as i32 % self.width;
    let y = idx as i32 / self.width;
    let w = self.width as usize;

    // Cardinal directions
    if self.is_exit_valid(x-1, y) { exits.push((idx-1, 1.0)) };
    if self.is_exit_valid(x+1, y) { exits.push((idx+1, 1.0)) };
    if self.is_exit_valid(x, y-1) { exits.push((idx-w, 1.0)) };
    if self.is_exit_valid(x, y+1) { exits.push((idx+w, 1.0)) };

    exits
}
```

Providing exits without a distance heuristic will lead to some horrible behaviour (and a crash on future versions of RLTK). So also implement that for your map:

```
impl BaseMap for Map {
    ...
    fn get_pathing_distance(&self, idx1:usize, idx2:usize) -> f32 {
        let w = self.width as usize;
        let p1 = Point::new(idx1 % w, idx1 / w);
        let p2 = Point::new(idx2 % w, idx2 / w);
        rltk::DistanceAlg::Pythagoras.distance2d(p1, p2)
    }
```

Pretty straight-forward: we evaluate each possible exit, and add it to the `exits` vector if it can be taken. Next, we modify the main loop in `monster_ai_system`:

```
use specs::prelude::*;
use super::{Viewshed, Monster, Name, Map, Position};
use rltk::{Point, console};

pub struct MonsterAI {}

impl<'a> System<'a> for MonsterAI {
    #[allow(clippy::type_complexity)]
    type SystemData = ( WriteExpect<'a, Map>,
                        ReadExpect<'a, Point>,
                        WriteStorage<'a, Viewshed>,
                        ReadStorage<'a, Monster>,
                        ReadStorage<'a, Name>,
                        WriteStorage<'a, Position>);

    fn run(&mut self, data : Self::SystemData) {
        let (mut map, player_pos, mut viewshed, monster, name, mut position) =
data;

        for (mut viewshed,_monster,name,mut pos) in (&mut viewshed, &monster,
&name, &mut position).join() {
            if viewshed.visible_tiles.contains(&*player_pos) {
                console::log(&format!("{} shouts insults", name.name));
                let path = rltk::a_star_search(
                    map.xy_idx(pos.x, pos.y) as i32,
                    map.xy_idx(player_pos.x, player_pos.y) as i32,
                    &mut *map
                );
                if path.success && path.steps.len()>1 {
                    pos.x = path.steps[1] as i32 % map.width;
                    pos.y = path.steps[1] as i32 / map.width;
                    viewshed.dirty = true;
                }
            }
        }
    }
}
```

We've changed a few things to allow write access, requested access to the map. We've also added an `#[allow...]` to tell the linter that we really did mean to use quite so much in one type! The meat is the `a_star_search` call; RLTK includes a high-performance A* implementation, so we're asking it for a path from the monster's position to the player. Then we check that the path succeeded, and has more than 2 steps (step 0 is always the current location). If it does, then we move the monster to that point - and set their viewshed to be dirty.

If you `cargo run` the project, monsters will now chase the player - and stop if they lose line-of-sight. We're not preventing monsters from standing on each other - or you - and we're not having them *do* anything other than yell at your console - but it's a good start. It wasn't too hard to get chase mechanics in!

Blocking access

We don't want monsters to walk on top of each other, nor do we want them to get stuck in a traffic jam hoping to find the player; we'd rather they are willing to try and flank the player! We'll accompany this by keeping track of what parts of the map are blocked.

First, we'll add another vector of bools to our `Map`:

```
#[derive(Default)]
pub struct Map {
    pub tiles : Vec<TileType>,
    pub rooms : Vec<Rect>,
    pub width : i32,
    pub height : i32,
    pub revealed_tiles : Vec<bool>,
    pub visible_tiles : Vec<bool>,
    pub blocked : Vec<bool>
}
```

We'll also initialize it, just like the other vectors:

```
let mut map = Map{
    tiles : vec![TileType::Wall; 80*50],
    rooms : Vec::new(),
    width : 80,
    height: 50,
    revealed_tiles : vec![false; 80*50],
    visible_tiles : vec![false; 80*50],
    blocked : vec![false; 80*50]
};
```

Lets introduce a new function to populate whether or not a tile is blocked. In the `Map` implementation:

```
pub fn populate_blocked(&mut self) {
    for (i,tile) in self.tiles.iter_mut().enumerate() {
        self.blocked[i] = *tile == TileType::Wall;
    }
}
```

This function is very simple: it sets `blocked` for a tile to true if its a wall, false otherwise (we'll expand it when we add more tile types). While we're working with `Map`, lets adjust `is_exit_valid` to use this data:

```
fn is_exit_valid(&self, x:i32, y:i32) -> bool {
    if x < 1 || x > self.width-1 || y < 1 || y > self.height-1 { return false; }
    let idx = self.xy_idx(x, y);
    !self.blocked[idx]
}
```

This is quite straightforward: it checks that `x` and `y` are within the map, returning `false` if the exit is outside of the map (this type of *bounds checking* is worth doing, it prevents your program from crashing because you tried to read outside of the the valid memory area). It then checks the *index* of the tiles array for the specified coordinates, and returns the *inverse* of `blocked` (the `!` is the same as `not` in most languages - so read it as "not blocked at `idx`").

Now we'll make a new component, `BlocksTile`. You should know the drill by now; in `Components.rs`:

```
#[derive(Component, Debug)]
pub struct BlocksTile {}
```

Then register it in `main.rs`: `gs.ecs.register::<BlocksTile>();`

We should apply `BlocksTile` to NPCs - so our NPC creation code becomes:

```
gs.ecs.create_entity()
    .with(Position{ x, y })
    .with(Renderable{
        glyph,
        fg: RGB::named(rltk::RED),
        bg: RGB::named(rltk::BLACK),
    })
    .with(Viewshed{ visible_tiles : Vec::new(), range: 8, dirty: true })
    .with(Monster{})
    .with(Name{ name: format!("{} #{}", &name, i) })
    .with(BlocksTile{})
    .build();
```

Lastly, we need to populate the blocked list. We'll probably extend this system later, so we'll go with a nice generic name `map_indexing_system.rs`:

```
use specs::prelude::*;
use super::{Map, Position, BlocksTile};

pub struct MapIndexingSystem {}

impl<'a> System<'a> for MapIndexingSystem {
    type SystemData = ( WriteExpect<'a, Map>,
                        ReadStorage<'a, Position>,
                        ReadStorage<'a, BlocksTile>);

    fn run(&mut self, data : Self::SystemData) {
        let (mut map, position, blockers) = data;

        map.populate_blocked();
        for (position, _blocks) in (&position, &blockers).join() {
            let idx = map.xy_idx(position.x, position.y);
            map.blocked[idx] = true;
        }
    }
}
```

This tells the map to setup blocking from the terrain, and then iterates all entities with a `BlocksTile` component, and applies them to the blocked list. We need to register it with `run_systems`; in `main.rs`:

```
impl State {
    fn run_systems(&mut self) {
        let mut vis = VisibilitySystem{};
        vis.run_now(&self.ecs);
        let mut mob = MonsterAI{};
        mob.run_now(&self.ecs);
        let mut mapindex = MapIndexingSystem{};
        mapindex.run_now(&self.ecs);
        self.ecs.maintain();
    }
}
```

If you `cargo run` now, monsters no longer end up on top of each other - but they do end up on top of the player. We should fix that. We can make the monster only yell when it is adjacent to the player. In `monster_ai_system.rs`, add this above the visibility test:

```
let distance = rltk::DistanceAlg::Pythagoras.distance2d(Point::new(pos.x, pos.y),
*player_pos);
if distance < 1.5 {
    // Attack goes here
    console::log(&format!("{} shouts insults", name.name));
    return;
}
```

Lastly, we want to stop the player from walking over monsters. In `player.rs`, we replace the `if` statement that looks for walls with:

```
if !map.blocked[destination_idx] {
```

Since we already put walls into the blocked list, this should take care of the issue for now. `cargo run` shows that monsters now block the player. They block them *perfectly* - so a monster that wants to be in your way is an unpassable obstacle!

Allowing Diagonal Movement

It would be nice to be able to bypass the monsters - and diagonal movement is a mainstay of roguelikes. So lets go ahead and support it. In `map.rs`'s `get_available_exits` function, we add them:

```
fn get_available_exits(&self, idx:usize) -> rltk::SmallVec<[(usize, f32); 10]> {
    let mut exits = rltk::SmallVec::new();
    let x = idx as i32 % self.width;
    let y = idx as i32 / self.width;
    let w = self.width as usize;

    // Cardinal directions
    if self.is_exit_valid(x-1, y) { exits.push((idx-1, 1.0)) };
    if self.is_exit_valid(x+1, y) { exits.push((idx+1, 1.0)) };
    if self.is_exit_valid(x, y-1) { exits.push((idx-w, 1.0)) };
    if self.is_exit_valid(x, y+1) { exits.push((idx+w, 1.0)) };

    // Diagonals
    if self.is_exit_valid(x-1, y-1) { exits.push(((idx-w)-1, 1.45)); }
    if self.is_exit_valid(x+1, y-1) { exits.push(((idx-w)+1, 1.45)); }
    if self.is_exit_valid(x-1, y+1) { exits.push(((idx+w)-1, 1.45)); }
    if self.is_exit_valid(x+1, y+1) { exits.push(((idx+w)+1, 1.45)); }

    exits
}
```

We also modify the `player.rs` input code:

```
pub fn player_input(gs: &mut State, ctx: &mut Rltk) -> RunState {
    // Player movement
    match ctx.key {
        None => { return RunState::Paused } // Nothing happened
        Some(key) => match key {
            VirtualKeyCode::Left |
            VirtualKeyCode::Numpad4 |
            VirtualKeyCode::H => try_move_player(-1, 0, &mut gs.ecs),

            VirtualKeyCode::Right |
            VirtualKeyCode::Numpad6 |
            VirtualKeyCode::L => try_move_player(1, 0, &mut gs.ecs),

            VirtualKeyCode::Up |
            VirtualKeyCode::Numpad8 |
            VirtualKeyCode::K => try_move_player(0, -1, &mut gs.ecs),

            VirtualKeyCode::Down |
            VirtualKeyCode::Numpad2 |
            VirtualKeyCode::J => try_move_player(0, 1, &mut gs.ecs),

            // Diagonals
            VirtualKeyCode::Numpad9 |
            VirtualKeyCode::Y => try_move_player(1, -1, &mut gs.ecs),

            VirtualKeyCode::Numpad7 |
            VirtualKeyCode::U => try_move_player(-1, -1, &mut gs.ecs),

            VirtualKeyCode::Numpad3 |
            VirtualKeyCode::N => try_move_player(1, 1, &mut gs.ecs),

            VirtualKeyCode::Numpad1 |
            VirtualKeyCode::B => try_move_player(-1, 1, &mut gs.ecs),

            _ => { return RunState::Paused }
        },
    }
    RunState::Running
}
```

You can now diagonally dodge around monsters - and they can move/attack diagonally.

Giving monsters and the player some combat stats

You probably guessed by now that the way to add stats to entities is with another component! In `components.rs`, we add `CombatStats`. Here's a simple definition:

```
#[derive(Component, Debug)]
pub struct CombatStats {
    pub max_hp : i32,
    pub hp : i32,
    pub defense : i32,
    pub power : i32
}
```

As usual, don't forget to register it in `main.rs`!

We'll give the `Player` 30 hit points, 2 defense, and 5 power:

```
.with(CombatStats{ max_hp: 30, hp: 30, defense: 2, power: 5 })
```

Likewise, we'll give the monsters a weaker set of stats (we'll worry about monster differentiation later):

```
.with(CombatStats{ max_hp: 16, hp: 16, defense: 1, power: 4 })
```

Indexing what is where

When traveling the map - as a player or a monster - it's really handy to know what is in a tile. You can combine it with the visibility system to make intelligent choices with what can be seen, you can use it to see if you are trying to walk into an enemy's space (and attack them), and so on. One way to do it would be to iterate the `Position` components and see if we hit anything; for low numbers of entities that would be plenty fast. We'll take a different approach, and make the `map_indexing_system` help us. We'll start by adding a field to the map:

```
#[derive(Default)]
pub struct Map {
    pub tiles : Vec<TileType>,
    pub rooms : Vec<Rect>,
    pub width : i32,
    pub height : i32,
    pub revealed_tiles : Vec<bool>,
    pub visible_tiles : Vec<bool>,
    pub blocked : Vec<bool>,
    pub tile_content : Vec<Vec<Entity>>
}
```

And we'll add a basic initializer to the new map code:

```
tile_content : vec![Vec::new(); 80*50]
```

While we're in `map`, there's one more function we are going to need:

```
pub fn clear_content_index(&mut self) {
    for content in self.tile_content.iter_mut() {
        content.clear();
    }
}
```

This is also quite simple: it iterates (visits) every vector in the `tile_content` list, mutably (the `iter_mut` obtains a *mutable iterator*). It then tells each vector to `clear` itself - remove all content (it doesn't actually guarantee that it will free up the memory; vectors can keep empty sections ready for more data. This is actually a *good* thing, because acquiring new memory is one of the slowest things a program can do - so it helps keep things running fast).

Then we'll upgrade the indexing system to index all entities by tile:

```
use specs::prelude::*;
use super::{Map, Position, BlocksTile};

pub struct MapIndexingSystem {}

impl<'a> System<'a> for MapIndexingSystem {
    type SystemData = ( WriteExpect<'a, Map>,
                        ReadStorage<'a, Position>,
                        ReadStorage<'a, BlocksTile>,
                        Entities<'a>,);

    fn run(&mut self, data : Self::SystemData) {
        let (mut map, position, blockers, entities) = data;

        map.populate_blocked();
        map.clear_content_index();
        for (entity, position) in (&entities, &position).join() {
            let idx = map.xy_idx(position.x, position.y);

            // If they block, update the blocking list
            let _p : Option<&BlocksTile> = blockers.get(entity);
            if let Some(_p) = _p {
                map.blocked[idx] = true;
            }

            // Push the entity to the appropriate index slot. It's a Copy
            // type, so we don't need to clone it (we want to avoid moving it out
of the ECS!)
            map.tile_content[idx].push(entity);
        }
    }
}
```

Letting the player hit things

Most roguelike characters spend a lot of time hitting things, so let's implement that! Bump to attack (walking into the target) is the canonical way to do this. We want to expand `try_move_player` in `player.rs` to check to see if a tile we are trying to enter contains a target.

We'll add a reader for `CombatStats` to the list of data-stores, and put in a quick enemy detector:

```
let combat_stats = ecs.read_storage::<CombatStats>();
let map = ecs.fetch::<Map>();

for (_player, pos, viewshed) in (&mut players, &mut positions, &mut
viewsheds).join() {
    let destination_idx = map.xy_idx(pos.x + delta_x, pos.y + delta_y);

    for potential_target in map.tile_content[destination_idx].iter() {
        let target = combat_stats.get(*potential_target);
        match target {
            None => {}
            Some(t) => {
                // Attack it
                console::log(&format!("From Hell's Heart, I stab thee!"));
                return; // So we don't move after attacking
            }
        }
    }
```

If you `cargo run` this, you'll see that you can walk up to a mob and try to move onto it. *From Hell's Heart, I stab thee!* appears on the console. So the detection works, and the attack is in the right place.

Player attacking and killing things

We're going to do this in an ECS way, so there's a bit of boilerplate. In `components.rs`, we add a component indicating an intent to attack:

```
#[derive(Component, Debug, ConvertSaveload, Clone)]
pub struct WantsToMelee {
    pub target : Entity
}
```

We also want to track incoming damage. It's possible that you will suffer damage from more than one source in a turn, and Specs doesn't like it at all when you try and have more than one component of the same type on an entity. There are two possible approaches here: make the damage an entity itself (and track the victim), or make damage a *vector*. The latter seems the easier approach; so we'll make a `SufferDamage` component to track the damage - and attach/implement a method to make using it easy:

```
#[derive(Component, Debug)]
pub struct SufferDamage {
    pub amount : Vec<i32>
}

impl SufferDamage {
    pub fn new_damage(store: &mut WriteStorage<SufferDamage>, victim: Entity,
amount: i32) {
        if let Some(suffering) = store.get_mut(victim) {
            suffering.amount.push(amount);
        } else {
            let dmg = SufferDamage { amount : vec![amount] };
            store.insert(victim, dmg).expect("Unable to insert damage");
        }
    }
}
```

(Don't forget to register them in `main.rs`!). We modify the player's movement command to create a component indicating the intention to attack (attaching a `wants_to_melee` to the *attacker*):

```
let entities = ecs.entities();
let mut wants_to_melee = ecs.write_storage::<WantsToMelee>();

for (entity, _player, pos, viewshed) in (&entities, &players, &mut positions, &mut
viewsheds).join() {
    if pos.x + delta_x < 1 || pos.x + delta_x > map.width-1 || pos.y + delta_y < 1
|| pos.y + delta_y > map.height-1 { return; }
    let destination_idx = map.xy_idx(pos.x + delta_x, pos.y + delta_y);

    for potential_target in map.tile_content[destination_idx].iter() {
        let target = combat_stats.get(*potential_target);
        if let Some(_target) = target {
            wants_to_melee.insert(entity, WantsToMelee{ target: *potential_target
}).expect("Add target failed");
            return;
        }
    }
}
```

We'll need a `melee_combat_system` to handle Melee. This uses the `new_damage` system we created to ensure that multiple sources of damage may be applied in one turn:

```
use specs::prelude::*;
use super::{CombatStats, WantsToMelee, Name, SufferDamage};

pub struct MeleeCombatSystem {}

impl<'a> System<'a> for MeleeCombatSystem {
    type SystemData = ( Entities<'a>,
                        WriteStorage<'a, WantsToMelee>,
                        ReadStorage<'a, Name>,
                        ReadStorage<'a, CombatStats>,
                        WriteStorage<'a, SufferDamage>
                      );

    fn run(&mut self, data : Self::SystemData) {
        let (entities, mut wants_melee, names, combat_stats, mut inflict_damage) =
data;

        for (_entity, wants_melee, name, stats) in (&entities, &wants_melee,
&names, &combat_stats).join() {
            if stats.hp > 0 {
                let target_stats = combat_stats.get(wants_melee.target).unwrap();
                if target_stats.hp > 0 {
                    let target_name = names.get(wants_melee.target).unwrap();

                    let damage = i32::max(0, stats.power - target_stats.defense);

                    if damage == 0 {
                        console::log(&format!("{} is unable to hurt {}",
&name.name, &target_name.name));
                    } else {
                        console::log(&format!("{} hits {}, for {} hp.",
&name.name, &target_name.name, damage));
                        SufferDamage::new_damage(&mut inflict_damage,
wants_melee.target, damage);
                    }
                }
            }
        }

        wants_melee.clear();
    }
}
```

And we'll need a `damage_system` to apply the damage (we're separating it out, because damage could come from any number of sources!). We use an iterator to *sum* the damage, ensuring that it is all applied:

```
use specs::prelude::*;
use super::{CombatStats, SufferDamage};

pub struct DamageSystem {}

impl<'a> System<'a> for DamageSystem {
    type SystemData = ( WriteStorage<'a, CombatStats>,
                        WriteStorage<'a, SufferDamage> );

    fn run(&mut self, data : Self::SystemData) {
        let (mut stats, mut damage) = data;

        for (mut stats, damage) in (&mut stats, &damage).join() {
            stats.hp -= damage.amount.iter().sum::<i32>();
        }

        damage.clear();
    }
}
```

We'll also add a method to clean up dead entities:

```
pub fn delete_the_dead(ecs : &mut World) {
    let mut dead : Vec<Entity> = Vec::new();
    // Using a scope to make the borrow checker happy
    {
        let combat_stats = ecs.read_storage::<CombatStats>();
        let entities = ecs.entities();
        for (entity, stats) in (&entities, &combat_stats).join() {
            if stats.hp < 1 { dead.push(entity); }
        }
    }

    for victim in dead {
        ecs.delete_entity(victim).expect("Unable to delete");
    }
}
```

This is called from our `tick` command, after the systems run: `damage_system::delete_the_dead(&mut self.ecs);`.

If you `cargo run` now, you can run around the map hitting things - and they vanish when dead!

Letting the monsters hit you back

Since we've already written systems to handle attacking and damaging, it's relatively easy to use the same code with monsters - just add a `WantsToMelee` component and they can attack/kill the player.

We'll start off by making the *player entity* into a game resource, so it can be easily referenced. Like the player's position, it's something that we're likely to need all over the place - and since entity IDs are stable, we can rely on it existing. In `main.rs`, we change the `create_entity` for the player to return the entity object:

```
let player_entity = gs.ecs
    .create_entity()
    .with(Position { x: player_x, y: player_y })
    .with(Renderable {
        glyph: rltk::to_cp437('@'),
        fg: RGB::named(rltk::YELLOW),
        bg: RGB::named(rltk::BLACK),
    })
    .with(Player{})
    .with(Viewshed{ visible_tiles : Vec::new(), range: 8, dirty: true })
    .with(Name{name: "Player".to_string() })
    .with(CombatStats{ max_hp: 30, hp: 30, defense: 2, power: 5 })
    .build();
```

We then insert it into the world:

```
gs.ecs.insert(player_entity);
```

Now we modify the `monster_ai_system`. There's a bit of clean-up here, and the "hurl insults" code is completely replaced with a single component insert:

```
use specs::prelude::*;
use super::{Viewshed, Monster, Map, Position, WantsToMelee, RunState};
use rltk::{Point};

pub struct MonsterAI {}

impl<'a> System<'a> for MonsterAI {
    #[allow(clippy::type_complexity)]
    type SystemData = ( WriteExpect<'a, Map>,
                        ReadExpect<'a, Point>,
                        ReadExpect<'a, Entity>,
                        ReadExpect<'a, RunState>,
                        Entities<'a>,
                        WriteStorage<'a, Viewshed>,
                        ReadStorage<'a, Monster>,
                        WriteStorage<'a, Position>,
                        WriteStorage<'a, WantsToMelee>);

    fn run(&mut self, data : Self::SystemData) {
        let (mut map, player_pos, player_entity, runstate, entities, mut viewshed,
monster, mut position, mut wants_to_melee) = data;

        for (entity, mut viewshed,_monster,mut pos) in (&entities, &mut viewshed,
&monster, &mut position).join() {
            let distance =
rltk::DistanceAlg::Pythagoras.distance2d(Point::new(pos.x, pos.y), *player_pos);
            if distance < 1.5 {
                wants_to_melee.insert(entity, WantsToMelee{ target: *player_entity
}).expect("Unable to insert attack");
            }
            else if viewshed.visible_tiles.contains(&*player_pos) {
                // Path to the player
                let path = rltk::a_star_search(
                    map.xy_idx(pos.x, pos.y),
                    map.xy_idx(player_pos.x, player_pos.y),
                    &mut *map
                );
                if path.success && path.steps.len()>1 {
                    let mut idx = map.xy_idx(pos.x, pos.y);
                    map.blocked[idx] = false;
                    pos.x = path.steps[1] as i32 % map.width;
                    pos.y = path.steps[1] as i32 / map.width;
                    idx = map.xy_idx(pos.x, pos.y);
                    map.blocked[idx] = true;
                    viewshed.dirty = true;
                }
            }
        }
    }
}
```

If you `cargo run` now, you can kill monsters - and they can attack you. If a monster kills you - the game crashes! It crashes, because `delete_the_dead` has deleted the player. That's obviously not what we intended. Here's a non-crashing version of `delete_the_dead`:

```
pub fn delete_the_dead(ecs : &mut World) {
    let mut dead : Vec<Entity> = Vec::new();
    // Using a scope to make the borrow checker happy
    {
        let combat_stats = ecs.read_storage::<CombatStats>();
        let players = ecs.read_storage::<Player>();
        let entities = ecs.entities();
        for (entity, stats) in (&entities, &combat_stats).join() {
            if stats.hp < 1 {
                let player = players.get(entity);
                match player {
                    None => dead.push(entity),
                    Some(_) => console::log("You are dead")
                }
            }
        }
    }

    for victim in dead {
        ecs.delete_entity(victim).expect("Unable to delete");
    }
}
```

We'll worry about ending the game in a later chapter.

Expanding the turn system

If you look closely, you'll see that enemies can fight back even after they have taken fatal damage. While that fits with some Shakespearean dramas (they really should give a speech), it's not the kind of tactical play that roguelikes encourage. The problem is that our game state is just `Running` and `Paused` - and we aren't even running the systems when the player acts. Additionally, systems don't know what phase we are in - so they can't take that into account.

Let's replace `RunState` with something more descriptive of each phase:

```
#[derive(PartialEq, Copy, Clone)]
pub enum RunState { AwaitingInput, PreRun, PlayerTurn, MonsterTurn }
```

If you're running Visual Studio Code with RLS, half your project just turned red. That's ok, we'll refactor one step at a time. We're going to *remove* the `RunState` altogether from the main `GameState`:

```
pub struct State {
    pub ecs: World
}
```

This makes even more red appear! We're doing this, because we're going to make the `RunState` into a resource. So in `main.rs` where we insert other resources, we add:

```
gs.ecs.insert(RunState::PreRun);
```

Now to start refactoring `Tick`. Our new `tick` function looks like this:

```
fn tick(&mut self, ctx : &mut Rltk) {
        ctx.cls();
        let mut newrunstate;
        {
            let runstate = self.ecs.fetch::<RunState>();
            newrunstate = *runstate;
        }

        match newrunstate {
            RunState::PreRun => {
                self.run_systems();
                newrunstate = RunState::AwaitingInput;
            }
            RunState::AwaitingInput => {
                newrunstate = player_input(self, ctx);
            }
            RunState::PlayerTurn => {
                self.run_systems();
                newrunstate = RunState::MonsterTurn;
            }
            RunState::MonsterTurn => {
                self.run_systems();
                newrunstate = RunState::AwaitingInput;
            }
        }

        {
            let mut runwriter = self.ecs.write_resource::<RunState>();
            *runwriter = newrunstate;
        }
        damage_system::delete_the_dead(&mut self.ecs);

        draw_map(&self.ecs, ctx);

        let positions = self.ecs.read_storage::<Position>();
        let renderables = self.ecs.read_storage::<Renderable>();
        let map = self.ecs.fetch::<Map>();

        for (pos, render) in (&positions, &renderables).join() {
            let idx = map.xy_idx(pos.x, pos.y);
            if map.visible_tiles[idx] { ctx.set(pos.x, pos.y, render.fg,
render.bg, render.glyph) }
        }
    }
}
```

Notice how we now have a state machine going, with a "pre-run" phase for starting the game! It's much cleaner, and quite obvious what's going on. There's a bit of scope magic in use to keep the borrow-checker happy: if you declare and use a variable inside a scope, it is dropped on scope exit (you can also manually drop things, but I think this is cleaner looking).

In `player.rs` we simply replace all `Paused` with `AwaitingInput`, and `Running` with `PlayerTurn`.

Lastly, we modify `monster_ai_system` to only run if the state is `MonsterTurn` (snippet):

```
impl<'a> System<'a> for MonsterAI {
    #[allow(clippy::type_complexity)]
    type SystemData = ( WriteExpect<'a, Map>,
                        ReadExpect<'a, Point>,
                        ReadExpect<'a, Entity>,
                        ReadExpect<'a, RunState>,
                        Entities<'a>,
                        WriteStorage<'a, Viewshed>,
                        ReadStorage<'a, Monster>,
                        WriteStorage<'a, Position>,
                        WriteStorage<'a, WantsToMelee>);

    fn run(&mut self, data : Self::SystemData) {
        let (mut map, player_pos, player_entity, runstate, entities, mut viewshed,
monster, mut position, mut wants_to_melee) = data;

        if *runstate != RunState::MonsterTurn { return; }
```

Don't forget to make sure that all of the systems are now in `run_systems` (in `main.rs`):

```
impl State {
    fn run_systems(&mut self) {
        let mut vis = VisibilitySystem{};
        vis.run_now(&self.ecs);
        let mut mob = MonsterAI{};
        mob.run_now(&self.ecs);
        let mut mapindex = MapIndexingSystem{};
        mapindex.run_now(&self.ecs);
        let mut melee = MeleeCombatSystem{};
        melee.run_now(&self.ecs);
        let mut damage = DamageSystem{};
        damage.run_now(&self.ecs);
        self.ecs.maintain();
    }
}
```

If you `cargo run` the project, it now behaves as you'd expect: the player moves, and things he/she kills die before they can respond.

Wrapping Up

That was quite the chapter! We added in location indexing, damage, and killing things. The good news is that this is the hardest part; you now have a simple dungeon bash game! It's not particularly fun, and you *will* die (since there's no healing at all) - but the basics are there.

The source code for this chapter may be found here

Run this chapter's example with web assembly, in your browser (WebGL2 required)
