Extracted from:

Hands-on Rust

Effective Learning through 2D Game Development and Play

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Hands-on Rust

Effective Learning through 2D Game Development and Play

Herbert Wolverson

edited by Tammy Coron

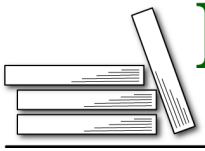
Hands-on Rust

Effective Learning through 2D Game Development and Play

Herbert Wolverson

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Tammy Coron
Copy Editor: Vanya Wong
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-816-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2021

Badly wounded by the goblin, the hero reaches into their backpack. Uncorking a terrible smelling potion and drinking it, their wounds close. Refreshed, the hero is ready to return to the good fight.

This scene is a staple of modern fantasy games: the adventurer finds an item on the map, stores it in their inventory, and then uses it later when needed. Substitute the smelly potion for a healing system controlled by sympathetic nanobots, and you have a similar game mechanic for a sci-fi game.

With items and inventory systems, you can add variety to your game. Players can now make tactical decisions based on item use—they'll need to consider whether it's worth the potential injuries to reach an item despite the number of monsters who might be guarding it.

However, before you can offer items for pickup, you first need to design and create them.

Designing Items

Item design is fun. You probably have some ideas for the items you want to include in your game. Taken alongside Chapter 15, Combat Systems and Loot, on page ?, this chapter will give you the experience you need to add items to your game and build a unique experience.

Here, you're going to add two items: *Healing Potions* and *Dungeon Maps*. Healing potions restore the player's hit points, and dungeon maps reveal the entire map, allowing the player to carefully plan a route to the Amulet of Yala. The graphics for these items are included in the `dungeonfont.png` file.

Let's start by giving the new items some component definitions.

Describing the Items with Components

You already have most of the components you need to describe the healing potion and dungeon map items—they need a name, appearance, and position on the map, which means you can re-use the following existing components:

- `Item`: Potion or Scroll
- `Name`: The name that appears in tooltips and the player's inventory list
- `Point`: The item's position on the map
- `Render`: The visual display component for the item

You also need new component types to describe what each item does.

The healing potion restores the drinker's hit points up to their maximum. You can describe this by creating a ProvidesHealing component. Open components.rs, and add a new component type:

InventoryAndPowerUps/potions_and_scrolls/src/components.rs
```
#[derive(Clone, Copy, Debug, PartialEq)]
pub struct ProvidesHealing{
    pub amount: i32
}
```

The amount field specifies how much health a given potion will restore. You could use this value to differentiate between types of healing potion, allowing you to make potions of varying strength.

The dungeon map reveals the entire level map when activated. You need to create a component indicating that this effect occurs:

InventoryAndPowerUps/potions_and_scrolls/src/components.rs
```
#[derive(Clone, Copy, Debug, PartialEq)]
pub struct ProvidesDungeonMap;
```

Why Not Use an Enum?

Each item you're adding provides only one effect. It's tempting to create a generic UseEffect component containing an enumeration. Enums can only have one value—if you want to make an item with multiple effects, you'd be out of luck. It's a good idea to separate effects into their own components in case you decide to create an item that does more than one thing.

Now that you have described your items with components, it's time to spawn these items on the map.

Spawning Potions and Maps

Open spawner.rs, and add two item spawning functions, one for each item type:

InventoryAndPowerUps/potions_and_scrolls/src/spawner.rs
```
pub fn spawn_healing_potion(ecs: &mut World, pos: Point) {
    ecs.push(
        (Item,
            pos,
            Render{
                color: ColorPair::new(WHITE, BLACK),
                glyph : to_cp437('!')
            },
            Name("Healing Potion".to_string()),
            ProvidesHealing{amount: 6}
        )
    );
```

```
}

pub fn spawn_magic_mapper(ecs: &mut World, pos: Point) {
    ecs.push(
        (Item,
            pos,
            Render{
                color: ColorPair::new(WHITE, BLACK),
                glyph : to_cp437('{')
            },
            Name("Dungeon Map".to_string()),
            ProvidesDungeonMap{}
        )
    );
}
```

The item spawning code is similar to the code you used to spawn the Amulet of Yala in Spawning the Amulet, on page ?, but with a different set of components.

Calling spawn_healing_potion() adds a healing potion to the map at the specified location. Likewise, spawn_magic_mapper() adds a dungeon map to the game level.

Now that you can spawn items, you also need to add the items to the list of things that might spawn in a designated tile. The spawn_monster() function handles this for monsters. Make a new function named spawn_entity() that can spawn your new items as well as monsters:

InventoryAndPowerUps/potions_and_scrolls/src/spawner.rs

```
pub fn spawn_entity(
    ecs: &mut World,
    rng: &mut RandomNumberGenerator,
    pos: Point
) {
    let roll = rng.roll_dice(1, 6);
    match roll {
        1 => spawn_healing_potion(ecs, pos),
        2 => spawn_magic_mapper(ecs, pos),
        _ => spawn_monster(ecs, rng, pos)
    }
}
```

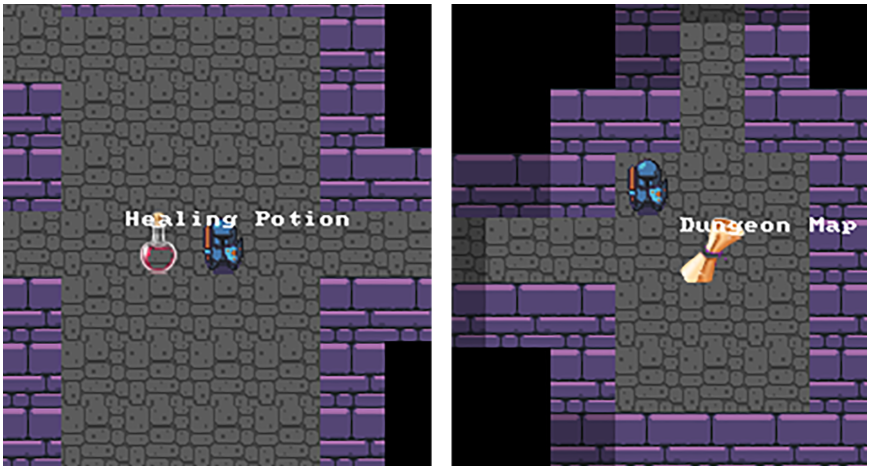
The function rolls a six-sided dice. If the dice roll results in a 1, a healing potion is spawned. A roll of 2 spawns a dungeon map. Otherwise, the spawn_monster() function is called.

Most items are liberally scattered throughout the game levels, but it's up to you how random you want to be versus how deliberate you are when placing these items. Managing the probabilities for spawns is a good way to tweak

the balance of your game level. You might decide to make items less frequent or adjust the weighting of monster types. Probability tables for spawns are discussed in Chapter 15, Combat Systems and Loot, on page ?.

Open main.rs, and replace all calls to spawn_monster() with spawn_entity(). Use your editor's search and replace function to replace all instances of spawn_monster with spawn_entity.

Run your game now, and you'll find potions and dungeon maps scattered throughout the dungeon level:



Now that the items are in the game, it's time to permit the player to pick them up.