Registry-driven spawning

This repo now uses a data-first spawning system:

- `data/spawn_registry.lua` : declarative types, defaults, and variants
- `spawning/spawner.lua` : reads the registry, resolves type/variant for each Tiled object, merges configs, and calls factories
- `spawning/factories/*.lua` : small adapters to construct live entities

See also: Writing factories

How it works

1. In Tiled, put spawnable objects in the `entity` object layer.
2. For each object:

- Name-only matching is used. The object name must start with a known prefix (box/ball/bell) and the full name must match an existing variant (e.g., `box1` , `ball2` , `bell1`).
- If the name doesn't match a declared variant, the object is skipped.

3. The spawner merges config in this order:

- per-type defaults (from `registry.types[type].defaults`)
- per-variant presets (from `registry.variants[type][variant]`)
- `object.properties` (from Tiled)

4. The resolved config is passed to the factory `create(world, x, y, obj, cfg, ctx)` .

Extend with a new type

- Add a factory: `spawning/factories/spike.lua` exporting `create(world, x, y, obj, cfg, ctx)`

- Register the type in `data/spawn_registry.lua` :

```
types = {
  spike = { factory = 'spawning.factories.spike', defaults = { lethal = true } },
}
variants = {
  spike = {
    spike1 = { w = 16, h = 8 },
  }
}
rules = {
  { when = { namePrefix = 'spike' }, type = 'spike', variant = { fromName = true } },
}
```

Now drop an object named `spike1` or set `type=spike` in Tiled.

Integration details

- `map.lua` now calls `Spawner.spawn(world, layer.objects, { registry, level, map })`
- Results are assigned to `state.boxes` , `state.balls` , `state.bells` for compatibility
- The old presets table in `map.lua` was removed; use `data/spawn_registry.lua` instead

Factory contract

A factory module returns a table with `create(world, x, y, obj, cfg, ctx)` .

- `world` : love.physics World
- `x, y` : center coordinates computed from the Tiled object
- `obj` : full Tiled object (with `.name` , `.type` , `.properties` , `.width` , `.height`)
- `cfg` : merged configuration from registry + variant + obj.properties

- `ctx` : extra context: `{ registry, level, map }`

Return the live instance (table/object). If your project already has an entity module, require and delegate to it from the factory.

Notes

- SaveState overlays are applied before spawning, so factories can read updated properties
- You can keep using `obj.name` to distinguish specific singles (e.g., `bell1`)
- To debug, print from factories or extend the registry with additional flags