RPG Engine

Write a game engine for an RPG game

Tibo De Peuter

23 december 2022

Contents

RPG-Engine

RPG-Engine is a game engine for playing and creating your own RPG games.

If you are interested in the development side of things, development notes can be found here.

This README serves as both documentation and project report, so excuse the details that might not be important for the average user.

Playing the game

These are the keybinds *in* the game. All other keybinds in the menus should be straightforward.

Action	Primary	Secondary
Move up	`Arrow Up`	`w`
Move left	`Arrow Left`	`a`
Move down	`Arrow Down`	`s`
Move right	`Arrow Right`	`d`
Interaction	`Space`	`f`
Show inventory	`i`	`Tab`
Restart level	`r`	
Quit game	`Esc`	

Example playthrough

TODO

- An example playthrough, with pictures and explanations

Writing your own stages

A stage description file, conventionally named `<stage_name>.txt` is a file with a JSON-like format. It is used to describe everything inside a single stage of your game, including anything related to the player, the levels your game contains and what happens in that level. It is essentially the raw representation of the initial state of a single game.

> Note: At the moment, every game has a single stage description file. Chaining several files together is not possible yet.

A stage description file consists of several elements.

Element	Short description
`Block`	optionally surrounded by `{ ... }`, consists of several `Entry`'s, optionally separated by commas `,`
`Entry`	is a `Key` - `Value` pair, optionally separated by a colon `:`
`Key`	is a unique, predefined `String` describing `Value`
`Value`	is either a `Block` or a `BlockList` or a traditional value, such as `String` or `Int`
`Block-List`	is a number of `Block`'s, surrounded by `[ ... ]`, separated by commas, can be empty

We'll look at the following example to explain these concepts.

```
player: {
    hp: 50,
    inventory: [
        {
            id: "dagger",
            x: 0,
            y: 0,
            name: "Dagger",
            description: "Basic dagger you found somewhere",
            useTimes: infinite,
            value: 10,

            actions: {}
        }
    ]
}

levels: [
    {
        layout: {
            | * * * * * *
            | * s . . e *
            | * * * * * *
        },
        items: [],
        entities: []
    },
    {
        layout: {
            | * * * * * * * *
            | * s . . . . e *
            | * * * * * * * *
        },
        items: [
            {
                id: "key",
                x: 3,
                y: 1,
                name: "Door key",
                description: "Unlocks a secret door",
                useTimes: 1,
                value: 0,
                actions: {
                    [not(inventoryFull())] retrieveItem(key),
                    [] leave()
                }
            }
        ],
        entities: [
            {
                id: "door",
                x: 4,
                y: 1,
                name: "Secret door",
                description: "This secret door can only be opened with a
                 ↪  key",
                direction: left,
                actions: {
                    [inventoryContains(key)] useItem(key),
                    [] leave()
                }
            }
        ]
    }
]
```

This stage description file consists of a single `Block`. A stage description file always does. This top level `Block` contains two `Values` `player` and `levels`, not separated by commas.

`player` describes a `Block` that represents the player of the game. Its `Entrys` are hp (a traditional value) and `inventory` (a `BlockList` of several other `Blocks`). They are both separated by commas this time. It is possible for the inventory to be an empty list `[]`.

`levels` is a `BlockList` that contains all the information to construct your game.

`layout` syntax

If `Key` has the value `layout`, `Value` is none of the types discussed so far. Instead `Layout` is specifically made to describe the layout of a level. This object is surrounded by `{ ... }` and consists of multiple lines, starting with a vertical line `|` and several characters of the following:

- `x` is an empty tile a.k.a. void.
- `.` is a tile walkable by the player.
- `*` is a tile not walkable by the player.
- `s` is the starting position of the player.
- `e` is the exit.

All characters are interspersed with spaces.

`actions` syntax

If `Key` has the value `actions`, the following changes are important for its `Value`, which in this case is a `Block` with zero or more `Entrys` like so:

- `Key` has type `ConditionList`.

  A `ConditionList` consists of several `Conditions`, surrounded by `[ ... ]`, separated by commas. A `ConditionList` can be empty. If so, the conditional is always fulfilled.

  A `Condition` is one of the following:

  - `inventoryFull()`: the players inventory is full.
  - `inventoryContains(objectId)`: the players inventory contains an object with id `objectId`.
  - `not(condition)`: logical negation of `condition`.

- `Value` is an `Action`.

  An `Action` is one of the following:

  ```
  - leave()
  - retrieveItem(objectId)
  - useItem(objectId)
  - decreaseHp(entityId, objectId)
  - increasePlayerHp(objectId)
  ```

Back to the example

If we look at the example, all the objects are

```
>Block<
    Entry = Key ('player') + >Block<
        Entry = Key ('hp') + Value (50)
        Entry = Key ('inventory') + >BlockList<
            length = 1
            Block
                Entry = Key ('id') + Value ("dagger")
                ... <several traditional entries like this>
                Entry = Key ('actions') + empty Block
    Entry = Key ('levels') + >BlockList<
        length = 2
        >Block<
            Entry = Key ('layout') + Layout
                <multiple lines that describe the layout>
            Entry = Key ('items') + empty BlockList
            Entry = Key ('entities') + empty BlockList
        >Block<
            Entry = Key ('layout') + Layout
                <multiple lines that describe the layout>
            Entry = Key ('items') + >BlockList<
                length = 1
                >Block<
                    Entry = Key ('id') + Value ("key")
                    ... <several traditional entries like this>
                    Entry = Key ('actions') + >Block<
                        Entry = >ConditionList< + Action ('retrieveItem(key)')
                            length = 1
                            Condition ('not(inventoryFull())'))
                        Entry = empty ConditionList + Action ('leave()')
            Entry = Key ('entities') + >BlockList<
                length = 1
                >Block<
                    Entry = Key ('id') + Value ("door")
                    ... <several traditional entries like this>
                    Entry = Key ('actions') + >Block<
                        Entry = >ConditionList< + Action ('useItem(key)')
                            length = 1
                            Condition ('inventoryContains(key)')
                        Entry = empty ConditionList + Action ('leave()')
```

Development notes

Engine architecture

TODO

`RPGEngine` is the main module. It contains the `playRPGEngine` function which bootstraps the whole game. It is also the game loop. From here, `RPGEngine` talks to its submodules.

These submodules are `Config`, `Data`, `Input`, `Parse` & `Render`. They are all responsible for their own part, either containing the program configuration, data containers, everything needed to handle input, everything needed to parse a source file & everything needed to render the game. However, each of these submodules has their own submodules to divide the work. They are conveniently named after the state of the game that they work with, e.g. the main menu has a module & when the game is playing is a different module. A special one is `Core`, which is kind of like a library for every piece. It contains functions that are regularly used by the other modules.

Monads/Monad stack TODO

Tests

TODO

Assets & dependencies

The following assets were used (and modified if specified):

- Kyrise's Free 16x16 RPG Icon Pack[1]
- 2D Pixel Dungeon Asset Pack by Pixel_Poem[2]

  Every needed asset was taken and put into its own `.png`, instead of in the overview.

RPG-Engine makes use of the following libraries:

- directory for listing levels in a directory
- gloss for game rendering
- gloss-juicy for rendering images
- hspec for testing
- hspec-discover for allowing to split test files in multiple files
- parsec for parsing configuration files

Future development ideas

The following ideas could (or should) be implemented in the future of this project.

- [ ] **Entity system:** With en ES, you can implement moving entities and repeated input. It also resembles the typical game loop more closely which can make it easier to implement other ideas in the future.
- [ ] **Game music:** Ambient game music and sound effects can improve the gaming experience I think.
- [ ] **Expand configuration file:** Implement the same methods for parsing stage description files to a configuration file, containing keybinds, dimension sizes, even window titles, making this a truly customizable engine.
- [ ] **Camera follows player:** The camera should follow the player, making it always center. This allows for larger levels increases the immersion of the game.

Conclusion

Parsing was way harder than I initially expected. About half of my time on this project was spent writing the parser.

TODO

References

[1] Kyrise's Free 16x16 RPG Icon Pack © 2018 by Kyrise is licensed under CC BY 4.0

[2] 2D Pixel Dungeon Asset Pack by Pixel_Poem is not licensed