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.

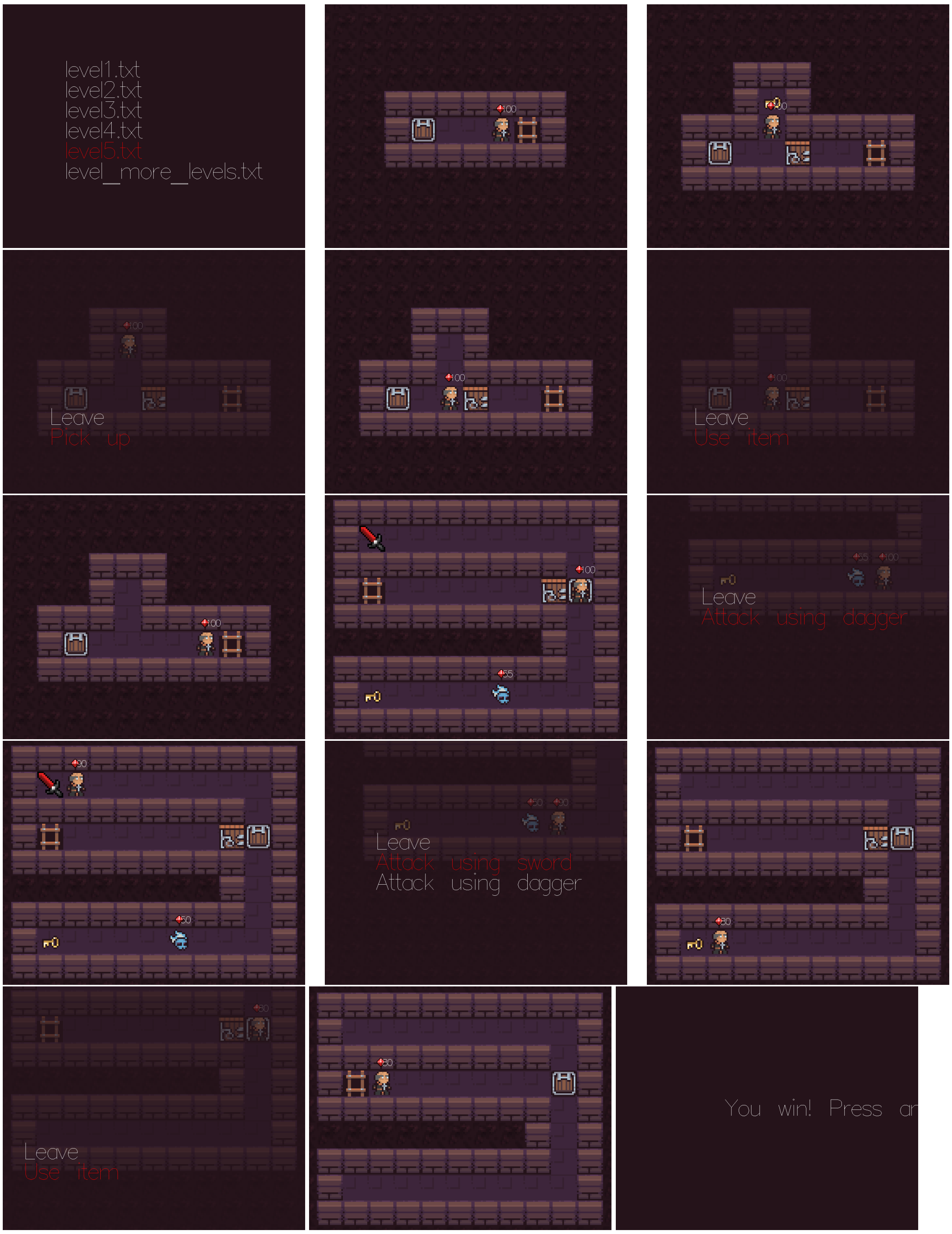
This README serves as both documentation and project report, so excuse the details that might not be important for the average reader.

Playing the game

These are the keybinds while *in* game. All other keybinds in menus etc. 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`, `Enter`
Show inventory	`i`	`Tab`
Restart level	`r`	
Quit game	`Esc`	

Example playthrough



Development notes

Engine architecture

`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. 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.

- `Config`: Configuration values, should ultimately be moved into parsing from a file.
- `Data`: Data containers and accessors of information.
- `Input`: Anything that handles input or changes the state of the game.
- `Parse`: Parsing
- `Render`: Rendering of the game and everything below that.

Monads/Monad stack Monads:

- Extensive use of `Maybe` for integers or infinity and do in parser implementation.
- `IO` to handle external information
- …

Monad transformer: ??

I am afraid I did not write any monad transformers in this project. I think I could (and should) have focused more on writing monads and monad transformers. In hindsight, I can see where I could and should have used them. I can think of plenty of ways to make the current implementation simpler. This is unfortunate. However, I want to believe that my next time writing a more complex Haskell program, I will remember using monad transformers. Sadly, I forgot this time.

An example of where I would use a monad transformer - in hindsight:

1. Interactions in game: something along the lines of …

```haskell
newtype StateT m a = StateT { runStateT ::  m a }

instance Monad m => Monad (StateT m) where
    return = lift . return
    x >>= f = StateT $ do
        v <- runStateT x
        case v of
            Playing level    -> runStateT ( f level )
            Paused  continue -> runStateT ( continue >>= f )
            -- etc
```

```
class MonadTransformer r where
    lift      :: Monad m => m a -> (r m) a
instance MonadTransformer StateT where
    lift      = StateT
```

2. Interaction with the outside world should also be done with Monad(transformers) instead of using `unsafePerformIO`.

Tests

Overall, only parsing is tested using Hspec. However, parsing is tested *thoroughly* and I am quite sure that there aren't a lot of edge cases that I did not catch. This makes for a relaxing environment where you can quickly check if a change you made breaks anything.

`Spec` is the main module. It does not contain any tests, but functions as the 'discover' module to find the other tests in its folder.

`Parser.StructureSpec` tests functionality of `RPGEngine.Parse.TextToStructure`, `Parser.GameSpec` tests functionality of `RPGEngine.Parse.StructureToGame`.

Known issues:

- [ ] Rendering is still not centered, I am sorry for those with small screens.
- [ ] Config files cannot end with an empty line. I could not get that to work and I decided that it was more important to implement other functionality first. Unfortunately, I was not able to get back to it yet.
- [ ] The parser is unable to parse layouts with trailing whitespace.

Conclusion

Parsing was way harder than I initially expected. I believe over half my time on this project was spent trying to write the parser. I am still not absolutely sure that it will work with *everything*, but it gets the job done at the moment. I don't know if parsing into a structure before transforming the structure into a game was a good move. It might have saved me some time if I did it straight to Game. I want to say that I have a parser-to-structure module now, but even so, there are some links between `TextToStructure` and Game that make it almost useless to any other project (without changing anything).

Player-object interaction was easier than previous projects. I believe this is both because I am getting used to it by now and because I spent a lot of time beforehand structuring everything. I also like to think that structuring the project is what I did right. There is a clear hierarchy and you can find what you are looking for fairly easy, without having to search for a function in file contents or having to scavenge multiple different files before finding what you want. However, I absolutely wasted a lot of time restructuring the project multiple times, mostly because I was running into dependency cycles.

Overall, I believe the project was a success. I am proud of the end result. Though, please note my comments on monad transformers.

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

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

Appendix A: 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.

Changes in the backend:

- [ ] **Make inventory a state** At the moment, there is a boolean for inventory rendering. This should be turned into a state, so it makes more sense to call it from other places as well.
- [ ] **Direction of entities** Change the rendering based on the direction of an entity.
- [ ] **Inventory with more details** The inventory should show more details of items, e.g. name, value, remaining use times and description.

Appendix B: Writing your own worlds

A world description file, conventionally named `<world_name_or_level_x>.txt` is a file with a JSON-like format. It is used to describe everything inside a single world 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.

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

`player` describes a `Block` that represents the player of the game. Its `Entry`s are hp (a traditional value) and `inventory` (a `BlockList` of several other `Block`s). 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 `Entry`s like so:

- `Key` has type `ConditionList`.

  A `ConditionList` consists of several `Condition`s, 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()')
```