Описание архитектуры Roguelike игры

Видение

Roguelike — обширное понятие, под которое попадают игры с разными стилистиками и игровыми механиками. Ниже представлены те аспекты геймплея, которые были зафиксированы для дальнейшей реализации.

Короткое описание
Суть игры заключается в том, чтобы проходить последовательность случайно генерируемых уровней (или определенных заранее), прокачивать своего Героя, улучшать снаряжение и сражаться с монстрами. Для победы игроку надо дойти до последнего уровня. В случае победы или смерти игра заканчивается и игроку присуждаются некоторое количество очков.

Мир игры
Игра состоит из 5 уровней, каждый из которых представлен некоторым биомом:

- Поляна
- Лес
- Шахта
- Склеп
- Преисподняя

Биом определяет как уровень выглядит, какие монстры в нем появляются и какой лут может выпасть. Сам уровень представлен сеткой небольшого фиксированного размера (достаточного для того, чтобы полностью загрузить и отобразить на пользовательском экране). На каждом тайле может находится либо враг, либо предмет окружения, либо сам герой. Чтобы перейти на следующий уровень надо наступить на определенный тайл. Вернуться на предыдущий уровень нельзя.

Характеристики существ и Героя
У всех существ есть следующие характеристики из которых затем складывается максимальный уровень здоровья, сила физической атаки, сила магической атаки, точность попадания и прочее:

- Сила
- Ловкость
- Телосложение
- Интеллект
- Показатель брони

У героя также есть показатель набранного опыта. При повышении уровня Герой может повысить одну из своих характеристик. Герой получает опыт за следующие действия:

- Убийство моба
- Прохождении очередного уровня
- Применении специальных предметов (см. применение предметов ниже)

Ход: возможные действия и порядок хода
Игра является пошаговой. За свой ход Герой и каждый моб могут:

- Переместиться по карте
- Атаковать другое существо в зоне досягаемости

Существа ходят в порядке убывания ловкости (если показатель ловкости один, то ход определяется случайно)

Бой
Если существо встает на клетку, где уже находилось другое существо или использует действие атаки, то происходит боевой раунд:

- Атакующий и защищающийся бросают кубики и добавляют свои значения ловкости. Защищающийся также добавляет показатель брони.
- Если защищающийся кинул больше, чем атакующий, то атака считается проваленной и сразу завершается
- Иначе атакующий кидает кубик, добавляет свое значение силы и наносит столько урона защищающемуся.

Предметы
Во время исследования карты, игроку попадаются предметы, которые он может положить к себе в инвентарь
Предметы могут оказаться на клетке карты при её генерации либо они могут выпасть из поверженного моба, занимая ту же клетку, в которой ранее находился моб
Пользователь может надеть/снять или применить вещь из инвентаря.
Вещи изменяют характеристики (не обязательно в плюс. Так например латный доспех может дать +4 к броне, но -2 к ловкости)
Расходные предметы меняют характеристики перманентно

Общее описание архитектуры

Архитектура игры состоит из двух частей:

- Библиотечная часть, реализующая всю логику игры, но она
    - не содержит в себе реализацию пользовательского интерфейса,
    - не содержит в себе реализацию обработки пользовательского ввода: будь то нажатие мышки, клавиши клавиатуры или геймпада;
- Исполняемая часть (`Application`), которая реализует недостающую функциональность выше.

Предполагается, что реализация пользовательской части может быть разной в зависимости от платформы.
Такой уровень косвенности позволяет ограничить доступ пользовательского интерфейса к логике посредством публичного API, реализуемого паттерном **Команда**.

Публичное API библиотечной части

Главный класс Game

Класс **`Game`** — это главный класс, его объекты надо рассматривать как конкретные игровые партии. Этот класс:

- В самом начале и при переходе на каждый следующий уровень инициализирует карту через **`MapLoader`**
- Отвечает за логику и порядок хода
- Предоставляет API для создания команд (**`move(Direction)`**, **`useItem()`** и т.д.),
- Предоставляет единственный корректный для **`Application`** способ исполнения комманд **`execute(Command)`**, возвращая список произошедших (в течение хода) **`ActionResult`**.

Создание и запуск команд

Пользовательский интерфейс изменяет состояние игры через объекты класса **`UserCommand`**, которые ему предоставляет класс **`Game`**. После создания команды ее надо запустить с помощью метода **`Game.execute(Command)`**.

Список доступных команд:

- **`MoveCommand`**
    - Перемещают игрока по некоторому направлению. Если игрок не может физически встать на эту клетку команда проваливается.
- **`AttackCommand`**
    - Попытка произвести атаку по какой то точке на уровне. В зависимости от снаряжения атака может провалиться автоматически (попытка совершить дальнюю атаку с мечом в руках).
- **`UseItemCommand`**
    - Применение расходников в инвентаре. Также позволяет надевать и снимать снаряжение
- **`PickUpItemCommand`**
    - Поднять предмет с точки на уровне
- **`DropItemCommand`**
    - Применяется для удаления вещи из инвентаря. Вернуть удаленную вещь нельзя
- **`UpgradeCommand`**
    - Применяется при повышении уровня и выбором пользователя желаемых для изменения параметров

После выполнения некоторых команд ход от героя передается мобам, а другие команды (например, **`UseItemCommand`** – взаимодействие с инвентарем) могут быть выполнены в один ход любое количество раз. Метод **`UserCommand.advancesTurn()`** позволяет различать эти два класса команд.

Обработка полученных ActionResult

Получив список **`ActionResult`**, исполняемой части надо его обработать. Чтобы избежать **`switch,`** для этого можно создать реализацию визитора **`ActionResultVisitor`**

Состояние игры и игровые объекты

Исполняемая часть получает доступ к игровым сущностям **`Map`**, **`Item`**, **`Entity`**, про которые будет сказано ниже.

Игровые сущности

Состояние игры выражено набором игровых сущностей, главной из которых является карта – **`Map`**. Можно сказать, что состояние игры имеет структуру дерева с корнем в объекте-карте.

Map

`Map` представляет карту, которую на экране видит пользователь. Карта состоит из не изменяемого набора клеток, которые можно сравнить с фоном, и из объектов **`Entity.`**

Интерфейс Entity

Это любая сущность располагающийся на карте. Она имеет свое местоположение (**`Location`**)и может передвигаться. Также у нее есть ссылка на карту. В текущий момент существует 3 вида этих сущностей: герой, мобы и лежащие вещи.

Герой и мобы

Все живые сущности на карте должны реализовывать интерфейс **`Creature,`** расширяющий **`Entity`**. Он хранит в себе характеристики, которые выделены в отдельный класс **`Stats`**:

- **`Health`**
- **`MaxHealth`**
- **`Strength`**
- **`Dexterity`**
- **`…`**

Все классы мобов должны реализовывать интерфейс **`Mob`**, расширяющий предыдущий интерфейс. Мобы умеют передвигаться и атаковать игрока, и у них есть поведение, представленное **`MobState и MobStrategy:`**

- **`MobState`**. Переходы между состояниями осуществляются в вызове единственного метода **`updateState()`**.Существуют три состояния:
  - **`AngryState`**, в которое:

        - Пассивный моб переходит, будучи атакованным (если `MobState.update()` замечает, что здоровье было уменьшено)
        - Агрессивный моб переходит, когда замечает героя
    - `DefaultState (PassiveState)`, в котором изначально находится любой моб
    - `CowardState`, в которое переходит любой моб, в случае, если уровень его здоровья становится меньше допустимого для боя значения
- `MobStrategy`. Изменяется в методе `MobState.update()` в случае, если необходимо изменить состояние. Каждое состояние реализует единственный метод **makeTurn**(), реализующий основную логику поведения моба. Существуют три стратегии, соответствующие каждому состоянию:
    - `AggressiveStrategy`, при которой моб старается атаковать героя
    - `PassiveStrategy`, при которой моб перемещается по карте в случайном направлении, не жалая атаковать героя
    - `CowardStrategy`, при которой моб перемещается по карте в направлении ОТ героя
- Высокоуровнево ход моба состоит из вызова двух функций:
    - `state.updateState()`
    - `strategy.move()`

Класс `Hero`

- Хранит в себе дополнительные характеристики:
    - `XP: Int` (Experience)
    - `upgradesUsed: Int`
- Имеет поле `inventory`, представляющее собой список из `Item`

Лежащие вещи

Лежащий на карте вещи представлены классом `DroppedItem` – оберткой вокруг `Item`.

Интерфейс Item

`Item` — это любая вещь, которая хранится у пользователя в инвентаре:

- Одежда, кольца
- Оружие
- Расходники (зелья, свитки)

Инициализация и изменение состояния игры

Игровые действия (Action)

Из любой точки библиотечной и исполняемой части запрещено изменять состояние игры напрямую, это могут делать только `Action`. Когда некоторое действие пользователя или моба должно изменить состояние игры, оно создает объект `Action`

и запускает его. Например, перемещение игрока (**MoveCommand**) может создать действия **Move** и **Attack**, в случае если начался бой.

Набор объектов, реализующих интерфейс **Action**, предоставляет набор примитивов позволяющих совершать игровые действия. Каждый такой объект в результате выполнения создает результат действия (**ActionResult**)

Результаты игровых действий (ActionResult)

Для активного поддержания корректного состояния на экране каждый **Action** порождает набор **ActionResult**, все созданные за ход результаты действий накапливаются в **ActionResultAccumulator** и возвращаются исполняемой части.

Создание карты

Логика игры предполагает два сценария создания карты: путём чтения файла конфигурации с диска или путём её генерации.
Для создания карты классом **Game** вызывается метод **getMap** класса **MapLoader**, которому передаётся номер уровня. Номер уровня определяет его биом и то, должен ли он быть загружен с диска.
В случае, если номер уровня попадет в множество уровней, которые нужно считать с диска, класс вызывает метод **loadConfig** у класса **MapConfigSerializer**, в противном случае он вызывает метод **generateConfig** у класса **ProceduralGenerator.** Оба этих метода возвращают объект класса **MapConfig,** который хранит в себе информацию о расположении клеток и мест появления мобов, героя и вещей на карте с информацией об их типах.
Биом уровня определяет то, какие на нём будут предметы, клетки и вещи. С помощью информации о биоме класс создаёт соответствующую ему фабрику, экземпляр класса **MapFactory**. Например, локации леса будет соответствовать класс **ForestFactory**, а локации преисподней — **HellFactory**. У каждой фабрики есть доступ к конкретным мобам и вещам, относящемся к конкретному биому. С помощью вызова методов **getMob**, **getTile** и **getItem** фабрика порождает необходимые сущности.
Преимущество данного подхода заключается в том, что логика генерации объектов, соответствующих конкретному биому инкапсулируется в класс фабрик.