Endless Runner Test

Ссылка на GitHub: https://github.com/Darkwing-Duck/EndlessRunner.git

Игра

Как описывалось в задании - это бесконечный раннер. В игре есть только одно взаимодействие игрока с героем - это прыжок. В задании не было, но я решил добавить чтобы продемонстрировать как можно легко добавлять игроков в игру. На данный момент в игре 2 игрока:

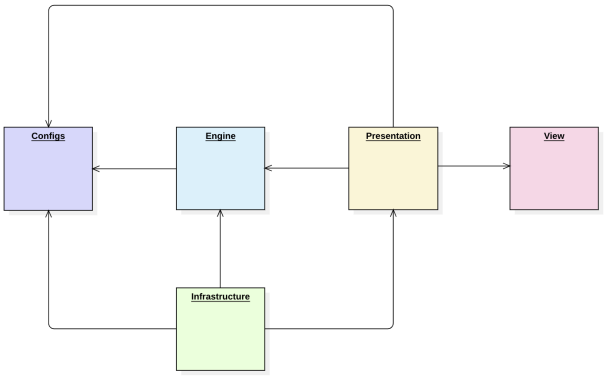
1. **Локальный игрок**, который может управлять прыжком своего героя по тапу на экране,
2. **Бот**, который просто прыгает через определенный интервал.

Герои могут взаимодействовать с элементами в игре:

- **Зеленая монета** - ускоряет героя на 10 сек.
- **Красная монета** - замедляет героя на 10 сек
- **Звездочка** - ускоряет героя и придает ему особый статус полета на 10 сек.

Слои

Весь код разбит на несколько слоев - **asmdef** в Юнити.
На следующей схеме показаны их зависимости:



Configs - содержит только код конфигов игры и не имеет никаких зависимостей.

View - этот слой не имеет зависимостей и содержит только классы *MonoBehaviours*.

Engine - описывает состояние игры и логику изменения этого состояния. Это центральный слой, независимый от **Unity**, поэтому для него можно легко писать тесты. И даже его может писать отдельная команда. Он не хранит позицию элементов и не отвечает за физику, поэтому он не является полной симуляцией игрового процесса. (ps. Если добавить в него управление позициями и столкновениями, то можно полностью симулировать его без модуля презентации и восстанавливать состояние на любом тике, а также легко воспроизводить реплеи имея только Random.Seed на входе и на каком тике какой инпут был от игрока)
Имеет только одну зависимость - **Configs**. На самом деле, в реальном проекте я бы так не делал, потому что енджин вообще не должен ничего знать об игровых конфигах. Правильно было бы, чтобы енджин имел свои собственные конфиги. Поэтому на этапе создания енджина нужно было бы зарегистрировать конфиги енджина только с нужными полями из игровых конфигов.

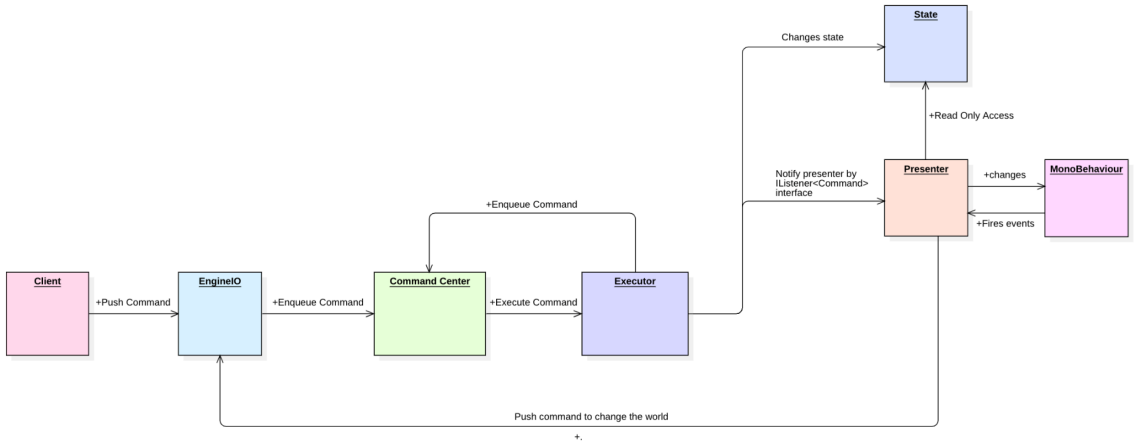
```
engine.RegisterHeroConfig(gameHeroConfig.Speed, gameHeroConfig.Health, gameHeroConfig.Damage);
```

Таким образом мы убираем зависимость от ненужных для енджина данных, которые могут быть нужны только для слоя презентации.
Енджин можно представить как черный ящик, мы отправляем что-то на вход и получаем результат на выходе, не вдаваясь в подробности реализации.

Presentation - этот слой содержит в себе логику игрового процесса. Основной элемент презентации - это Presenter (*HeroPresenter*, *LevelPresenter*). *Presenter* отвечает за визуальную логику и также имеет ссылку на соответствующий *MonoBehaviour*. Presenter отвечает за изменение соответствующего MonoBehaviour из слоя **View**, поэтому только этот модуль имеет зависимость на **View** модуль. А также имеет зависимости на **Engine** и **Configs**. Слой презентации имеет только *ReadOnly* доступ состоянию енджина. И чтобы как-то повлиять на это состояние нужно отправить команду в енджин.

Infrastructure - слой, отвечающий за сбор всех частей вместе - *Application Layer*. Поэтому он имеет зависимости на **Engine**, **Presentation** и **Configs** модули. Тут создается игра, создаются игроки, а также может быть любой код, не относящийся напрямую к игровому процессу (аналитика, …).

Архитектура



Для архитектуры проекта за основу взяты шаблоны ***Redux*** и ***MVU***. Где есть один глобальный стейт приложения и изменить его можно только через одну точку - отправить соответствующую команду на вход. Основное отличие в том, что в этих шаблонах стейт не изменяем (*immutable*) и при отправке на вход действия на изменения стейта, на выходе получаем копию стейта с внесенными изменениями. В моей же реализации стейт изменяемый, но только в рамках слоя **Engine**. Остальные слои, имеющие доступ к енджину, могут только читать стейт (реализовано за счет модификатор доступа *internal*).

Как это все работает:

1. Внешний код посылает команду в енджин
   ```
   _engine.Push(new CreateHeroCommand(heroConfigId, heroConfig.Speed, forPlayerId));
   ```
2. Енджин добавляет эту команду в очередь в *CommandCenter*
3. *CommandCenter* для каждой команды в очереди ищет соответствующий Executor и вызывает метод *Execute* на нем передавая команду на вход.
   ```
   executor.Execute(command);
   ```
4. *CreateHeroCommand.Executor* выполняет свою логику, меняя стейт и возвращает результат *CreateHeroCommand.Result*.
5. *CommandCenter* ищет реакцию для *CreateHeroCommand.Result* и отправляет в нее этот результат.
6. В слое **Presentation**, *LevelPresenter* реализует интерфейс *IListener<CreateHeroCommand.Result>*, это значит, что он словит нотификацию с результатом выполнения команды и создаст *HeroPresenter* и добавит его в дерево отображения - *PresentationRoot*.
7. *HeroPresenter* загружает и создает вью героя из аддрессеблов.

Терминология кода

Element - состояние любой динамической игровой сущности, которая может встречается в игровом мире (Герой, монетка, препятствие и тд.).

Stat - атрибут элемента, который может меняться посредством наложения модификаторов. Элемент хранит набор статов. В данной игре только *Speed*.

StatModifier - модификатор стата, который можно накладывать на стат. Описывает одну операцию над статом. В игре поддержано 2 типа:

1. **Add** - сложение
2. **Sub** - вычитание

Hero - элемент в мире, представляющий главного персонажа, который бежит и взаимодействует с другими, второстепенными предметами в игре.

- В игру добавлено 2 героя: Бабушка(*Granny*) и Рестлер в купальнике(*Ortiz*).

World - состояние игрового мира. Хранит список элементов и предоставляет доступ к ним.

Player - игрок/юзер, каждому Player'у соответствует один Hero.
В данном проекте поддержано 2 игрока:

1. **Local** - позволяет управлять героем вручную.
2. **AI(бот)** - управляет героем как бот (очень простая реализация)
3. Тут также мог бы быть **NetworkPlayer** - который бы управлял героем из сети.

Effect - это одно действие. Все модификации накладываемые на игрока - это эффект. Каждый эффект отвечает только за одно конкретное изменение и не должен влиять на несколько параметров. (Пример: нанести 5 урона, добавить модификатор скорости на 5).

В данном проекте есть 4 эффекта:

1. **ModifySpeed** - модифицирует скорость у элемента на указанную величину (5 / -5)
2. **AddStatus** - накладывает статус на элемент
3. **RemoveSrtatus** - снимает статус с элемента и все примененные модификаторы, примененные этим статусом
4. **SetState** - изменяет состояние героя

Status - статус позволяет накладывать на игрока любое количество эффектов. Т.е. статус описывает набор эффектов, которые применятся на цель при наложении. Статус можно наложить на любой элемент (В данном проекте накладывается только на героя).
Есть 2 типа статуса:

- **Permanent** - висит на цели, пока мы его не снимем (Duration = -1)
- **Timed** - висит на цели указанное количество времени

В данном проекте поддержаны 3 временных статуса:

1. **SpeedUp** - повышает скорость героя на 2 на 10 сек (эффект *ModifySpeed[2]*)
2. **SpeedDown** - снижает скорость героя на 2 на 10 сек (эффект *ModifySpeed[-2]*)
3. **SuperFly** - увеличивает скорость героя на 10 и позволяет герою взлететь и лететь в течении 10 секунд. Накладывает 2 эффекта (*ModifySpeed[10]* и *SetState["SuperFly"]*)

Command - описывает команду, которую можно отправить в енджин на выполнение. Это просто данные, никакой логики.

Command.Executor - содержит логику для выполнения определенной команды. Только *Executor* может изменить состояние мира. Он же и создает результат команды и возвращает его.

Command.Result - результат выполнения определенной команды. *Executor* должен создать результат и вернуть его из метода *Executor.Execute*