PyBomberman

Bombs, plot twists, torn off limbs, gruesome deaths. Fun for the whole family.

Requirements

Project is developed in Python 3.4 and Python 3.5 environments. File requirements.txt contains all the requirements. See Installing dependencies below.

Creating Virtual Environment

Using virtual Python environment is strongly recommended to run PyBomberman.

```
pyvenv-3.5 env
source env/bin/activate
```

Installing dependencies

Navigate to project root directory and run following line in a shell:

```
pip install -r requirements.txt
```

Hint: you may have to install PyGame in a non-standard way. Either try to compile it from sources, or let pip do it. Please note, you need `mercurial` installed for this to work.

```
pip3 install hg+https://bitbucket.org/pygame/pygame
```

Running project

PyBomberman may be started by running script from project root:

```
./main.py
```

Technical details

- **framework** module contains generic patterns for developing games in Python 3 with PyGame.
- **core** submodule encapsulates simple input-update-draw logic in its `Game` class. It is responsible for implementing game window and invoking `handle_input`, `handle_draw` and `handle_update` methods of provided `GameHandler` object in initializer.
- **state** submodule has `State` interface derived from `GameHandler`, which subclass instances are operated by `StateManager`. To use this feature, you need to pass `StateGameHandler` to `Game`'s initializer.
- **scene** includes `Node` and `NodeGroup` for managing scene graph. These should be `update`d and `draw`n in appropriate `GameHandler` events. It's a shame PyGame hasn't got something like this!
- **input** allows easy use of PyGame key input (although it can be easily adapted to handle other input as well). `Action` interface along with straightforward `NormalAction` and almost as trivial `InitialAction` managed by `InputManager` may be used as deadly simple, yet powerful and extensible input processor.
- **___init___** lets you `from framework import state_manager` to make some state pushing and `from framework import input_manager` if you want to make use of **input** submodule.
- **pybomberman**
- **shape** contains `Rectangle` implementation of `Shape` interface. You know what rectangle is, right?
- **physics** holds simple methods for `Shape`s collision checking and resolution.
- **objects** has `GameObject` extending `framework.scene.Node` with `Shape` and other variables such as `velocity` or `speed`. It is responsible for `update`ing and `draw`ing itself to PyGame's Surface.
- **config** contains configuration classes loaded from and saved to `config.json` file. Number of players, their key bindings and screen size can be found here.
- **menu** has got some implementation of `MenuState` to use with `framework.input.InputManager` and `Item` which should be named `MenuItem` instead. After serious refactoring (or even rewriting) it is intended to be moved to **framework** module.

- **controllers** and its `Controller` interface are responsible for **controlling** `GameObject`s. This abstraction layer enables easy integration of AI algorithms or player's input. Latter is provided with `HumanController` in the same submodule.
- **facade** provides an easy access to the game system.
- **utils** submodule. What to explain here? Well, if PyGame's Rectangle would use floats instead of ints, it wouldn't be neccessary.
- **superpower**? There is no such module, but it would be nice to have its implementation!

Assembling complete game from framework and pybomberman

Every computer game needs a window (technically we can assume that terminal is a window - and fullscreen window is still a window, right?) with graphics inside (ASCII characters are also graphics!) that is constantly changing dependently on user's input. All of this is provided with `Game` class.

We want to make a simple menu, where user will be able to start the game, configure it, or exit whenever he feels like it. Maybe because their graphic card is overheating or something. Yeah, I see no other reason. Menu and 'MainGame' will be implemented as `State`s, so we will use `StateGameHandler` and `StateManager`.

This is exactly what we're looking for:

```
state_manager.push(MenuState())
Game(handler=StateGameHandler()).start()
```

When user wants to quit, on exit option selected, `state_manager.pop()` method may be used. Framework will automagically close game window when there's nothing on state stack.

When user wants to play the game, on play option selected, `state_manager.push(GameState())` shall be invoked and Framework will present new state to the user.

`GameState` is more complicated.

On initialization, it creates `World` instance, where all the `GameObject`s, such as players, walls or even bombs, will be stored. Controllers are also instantiated here, but they are not stored in `World`.

Since we created `InputManager`, `handle_input` method is as simple as forwarding input events to its instance.

Every `handle_update`, all the `controller`s and `World` are being updated. Collisions between players with walls are resolved, but bombs with players and destructible walls are only checked. And if a player is ablaze by a bomb that has just exploded, they (and their controllers) are simply removed from the game. (Unless they're immortal. Which they can temporarily be by picking up a specific powerup. Well, in the future it will be implemented. Perhaps.) Same goes for destructible walls, but here, a powerup might (20%) be left in `World`.

Game is active as long as >= 2 players have all their limbs.