Simple Python “Games” pt. 3

Contents

1 Skills/Concepts

This module is a continuation of the Python Games module.

This module will continue to build on the previous concepts of:

- SDL
- blitting
- Drawing APIs
- Classes

In addition, this module will introduce input handling, finite state machines, and callbacks.

2 Preparation

Continue with the same environment from the previous (Python Games) module.

We will be using an existing git repository that contains a fair amount of code instead of producing a new program from scratch. This is to avoid copy and paste errors and allow us to cover material faster.

To get the code, execute the following command:

```
git clone https://www.github.com/cyberpost500/modules/
```

Then extract the project repository to your desktop:

```
cd modules/python-games/
tar -C ~/Desktop/ -xf project.tar
```

The code will be in the Desktop/project directory.

3 Menus

3.1 Background

The "game" already features drawing graphics and text, but it doesn't use them in a meaningful way. This module introduces a menuing system.

The code to be modified is in the `menus` branch. This is something of a departure from previous branches, there are quite a number of changes between this branch and the previous branch.

```
git checkout menus
```

3.2 Review/Run/Modify the Code

0. View the Documentation for the program:

```
pydoc game
```

Press *Q* to exit when you're done.

1. Run the Program:

```
python game.py
```

Press the escape key and then use the arrow keys to navigate the menu.

Eventually choose the quit option from the menu.

2. Take a moment to review the content of the file, just skimming through it is fine.
3. Change the colors used in the menu, see the `create_menu` function (line 322).

```
    return Menu([MenuItem("MENU", 40, WHITE, None),
                 MenuItem("play", 20, WHITE, BLACK, play),
                 MenuItem("reset", 20, WHITE, BLACK, reset),
                 MenuItem("quit", 20, WHITE, BLACK, doquit)])
```

You may notice there are variables for some colors, `WHITE`, `BLACK`, `GREEN`, `RED`, `YELLOW`, and `BLUE` are all (currently) valid color options. Use one of those, or define a new color.

5. Save and run your program.
6. Change the sizes used in the menu, again, see the `create_menu` function.
7. Save and run your program.
8. Modify the reset option so that it resets the game to the initial state with the shapes in the upper left hand corner again.

This modification will require creating a new state, modifying the `reset` function to assign the global STATE to this new state, and modifying the `main` function to take an action when the STATE has been set to this new state.

Hints:

- Create a new state variable (around line 409). You could call it `RESET_STATE`.
- Modify the `reset` function to update `STATE` and assign the new value.
- Modify the `main` function to reassign the shapes variable when `STATE` is in this new `STATE`.
- Make sure the `STATE` is changed back to the `PLAY` or `MENU` state once the shapes have been reassigned.

9. Save and run your program.
10. Revert the file changes back.

Undo your edits to the file using the following git command:

```
git checkout game.py
```

Note: There is a `menus_with_reset_example` branch for one example of how the reset implementation might work.

3.3 Conclusion

These were small modifications, but they served to help make you familiar with the new program flow and introduce global variables and state machines.

The global variables used by this version of the program were in all capital letters. This is just a convention to make it more obvious when a variable somewhere is actually a global variable. The reason for this is that it can be difficult to tell when a variable being used is actually a global variable.

Global variables can be helpful, but there's a trade off in using them. Global variables are useful because they can be accessed globally. But they can impact maintaining a program into the future because it's not always obvious where the variables are being used, hence the all capitals convention.

The STATE variable was used to track the state of the game. It was made global because it needed to be affected at a very wide scope. The callback functions `play`, `reset`, and `quit`, needed to be able to modify STATE and there was no direct path between the functions to convey this change through any other means. Then, once the STATE had been changed, the `main` function needed to be able to choose different actions to take when in the new STATE after a callback function executed.

Finite state machines are usually recognizable in code by a state being tracked (the STATE variable) and a main loop that examines the state and then takes action based on the value of the state. This part is generally a series of if statements (or a switch or case statements in other languages). The loop that examines the state will also progress that state through several finite states. In the game these states were MENU, PLAY, RESET, and QUIT. The program began in the menu state and then based on user input it decided what states it should move into. Some states could move into new states without additional input, for example, the RESET state you added, but others, like PLAY and MENU required additional user input to transition to a new state.

4 Gamier

4.1 Background

The "game" is a little more active, but not really interactive. This module introduces more meaningful input handling.

The code to be modified is in the `gamier` branch. This again adds a number of changes between this branch and the previous branch.

```
git checkout gamier
```

4.2 Review/Run/Modify the Code

0. View the Documentation for the program:

```
pydoc game
```

Press *Q* to exit when you're done.

1. Run the Program:

```
python game.py
```

2. Make the food items always use the same color.

Line 533 defines a `create_food` function, find the element responsible for the color and set it to a constant color instead of using random.

3. Save and run your program.
4. Change the value of the limit parameter as used by create food, try making it smaller, and then try making it larger, but change it in very small increments.

Save and run your program between modifications.

5. Modify the total number of npcs created by altering the values passed in as the count parameter to `create_npc_shapes`. See the invocation in `initialize`.
6. Save and run your program.
7. Modify the `create_npc_shapes` function so that npcs are only ever created in the top half of the screen.
8. Save and run your program.
9. Make the speed of the moving shapes depend on the size of the shape that's moving. Making larger objects move slower will give smaller objects a chance to grow before they can be consumed, but if you make the larger objects faster, the scenario can more quickly eliminate smaller objects.

Hint: You'll need to find a function common to all of the moving shapes that gets called after the size has been modified but before the shape is drawn again, in order to change the speed.

10. Save and run your program.
11. Add a shape for the player of the game by extending one of the existing moving shape classes and render a letter or text in the middle of the shape so that it's obvious it's the human/player. Consider that you may also add features to the shape, like eyes/nose/mouth/etc.

Hints:

- Some shapes cannot resize, so pick one that can be resized.
- Consider modeling the new shape after the bouncing rectangle class
  - This would make the player bounce off a wall
  - Even if you don't make the new class from the bouncing rectangle, pay attention to the way the BouncingRectangle only needs to over-ride a single method to accomplish what it does.
- Look at the initialization of font and text in main for an idea of how to create the text and image for the label.
- Put the actual label blitting into the draw function of the new shape.

12. We will add input handling in a future module after we discuss the types of input, game objectives, and scoring that should be added to the game. For now, continue to experiment by tweaking and customizing the code.
13. As you make changes that you'd like to keep, consider committing those changes, for example:

```
git commit -a -m 'added a player class'
```

And after adding more: ~˜ git commit -a -m 'added a label to the player class' ˜ ˜ ~˜

Even if it's not quite right you can still commit your changes: ~˜ git commit -a -m 'added a player class, but it's not quite right...' ˜ ˜ ~˜

4.3 Conclusion

The "game" still isn't an actual game, it's more of a limited simulation in its use of randomly generating content, but the components are getting closer to a full-on game.

You can start to see how the code can be adapted to turn the "game" into a more proper game with elements such as a score count, actual game objectives (eat the food to get larger, get larger to take out other enemies, etc).

To take it further, we need to discuss the sorts of objectives you'd like incorporated into the game and what other features you might want to add.