Simple Python “Games”

Contents

1 Skills/Concepts

This module will involve creating some simple, Python programs. These programs will build up the skills needed to write a simple python game.

This will involve the following concepts:

- SDL
- blitting
- Drawing APIs

1.1 Console Use

This module relies heavily on the command-line. You will be using a terminal emulator to execute the following commands.

Command	Description
git	A version control system
python	The Python interpreter
pydoc	Views Python documentation

2 Game Design

Working on actual software project or game is not carried out at all in the way we will be doing development in this module. This module serves as an example of different application programmer interfaces (APIs) that one would use in creating a game. We will discuss requirements and design in a future module.

3 Preparation

You should be fine booting up any Linux distribution as long as it has Python and the “Python Game” package (which is a Python wrapper for SDL - The Simple Direct Media Layer).

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/
```

The code will be in the `modules/pythong-games/project` directory.

4 Hello World

4.1 Background

In order to create a simple “Hello, World!” style program using PyGame, the graphics sub-system has to be initialzed, a primary surface for creating graphics is needed, and some basic event handling is needed.

These APIs are demonstrated on the `hello_world` branch of the project repository. Switch to that branch with the following command:

```
git checkout hello_world
```

4.2 Review/Run/Modify the Code

Review the structure of the code by opening the `game.py` file in a text editor. The bulk of the interesting code is in the `main` function.

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. Close the Program

Either press *Control+C* in the window where the program was executed, or close the window that is launched by the program using the mouse.

3. Modify the Window Size

```
screen = pygame.display.set_mode((800, 600))
```

Line 24 creates the primary display device with an initial "mode" of 800 by 600 pixels in size. Change the size of the window so it is either larger or smaller and run the program again.

4. Modify the Window Caption

```
pygame.display.set_caption("The Game")
```

Line 25 sets the window title to "The Game". Edit the value so that it says anything else and run the program again. (Don't forget to close the previous window before running it if it's still open). Can you guess what it might do if this line were removed from the code?

5. Modify the screen fill color

```
screen.fill((0, 0, 0))
```

The `fill` function takes a color value. This value is the amount of Red, Green, and Blue, respectively used to fill the screen.

Modify the value and run the program again.

6. Comment out the call to `flip`

```
pygame.display.flip()
```

Put an octothorpe (number sign or hash tag) in front of line on line 41 to comment it out and then run the program again. Based on the comment on line 39 to 40, do you understand why commenting out the line had this effect?

7. Revert the file changes back

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

```
git checkout game.py
```

4.3 Conclusion

The sample code contains the simplest set of instructions to get a graphical application to pop up with its own window using an explicit window size and giving it a title.

5 Full Screen Hello World

5.1 Background

The `hello_world` branch of the `game.py` file uses a hard-coded value for the window size. It would be nicer if it could display a full screen window instead.

Check out the `better_hello_world` branch to see a better way of creating the window.

```
git checkout better_hello_world
```

5.2 Review/Run/Modify the Code

Review the structure of the code by opening the `game.py` file in a text editor.

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. It's still not full screen, so make it full screen by default.

Start with the `main` function and locate the code responsible for creating the window. Perhaps you noticed in the documentation that it has been moved to a `create_window` function.

```
screen = create_window()
```

On line 37, modify the code so that it passes a value for fullscreen, the modified call will look like this:

```
screen = create_window(fullscreen=True)
```

Run the code again. What happened to the window title? Use *alt-tab* to switch windows to the terminal where the program was executed and close it with a *control+c*.

3. Revert the file changes back

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

```
git checkout game.py
```

5.3 Conclusion

The logic to make the program fullscreen all the time didn't need to be in a separate function, but by separating it out like this it keeps logic for doing minute details out of the main function. This makes the main function shorter and easier to read. Incorporating default parameters can also improve readability and if the default values are the most commonly used values, it makes it easier to use the function.

6 Lies! That's not Hello World!

6.1 Background

Perhaps you noticed the so-called "Hello, World!" program doesn't actually produce any output (aside from a filled window).

Check out the `really_hello_world` branch to see a version of the `game.py` file that actually displays "Hello, World!".

```
git checkout really_hello_world
```

6.2 Review/Run/Modify the Code

Review the structure of the code by opening the `game.py` file in a text editor.

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. Modify the text that gets displayed

```
font = pygame.font.Font(None, 100) #use default font, size 100
text = font.render("Hello, World!", 1, (0, 0, 0), (0xff, 0, 0))
```

Line 40 creates a font for rendering text. Line 41 renders that text using anti aliasing (the 1), the specified foreground color (black), and the specified background color (red). Modify these values and witness the effect by running the program again. (Don't forget to save your changes before running the code again.)

3. Comment out the call to blit

```
screen.blit(text, (0, 0, 800, 600))
```

Put an octothorpe (number sign or hash tag) in front of line on line 51 to comment it out and then run the program again. Based on the comment on line 49 to 50, do you understand why commenting out the line had this effect?

5. Revert the file changes back

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

```
git checkout game.py
```

6.3 Conclusion

Now the program is a basic, graphical "Hello, World!". It handles window events for proper closing and illustrates the use of fonts and simple blitting.

Blitting is the act of performing a blit. A blit gets its name from "bit block transfer". It takes one or more source bitmaps and copies them to a destination bitmap. In the case of our program, the source is the rendered font and the destination is the screen.