
Taz Documentation

Release 1.0.0

Max Klingmann <KlingmannM@gmail.com>

May 26, 2015

1	Content:	3
1.1	Introduction	3
1.2	Using the “Game” class	5
1.3	Inheriting the “Scene” class	7
1.4	Examples	9
2	Contact:	11
2.1	Contact	11
3	Taz API Documentation	13
3.1	Taz API Documentation	13
4	Indices and tables	15
	Python Module Index	17

This project is to aid people in getting started with their own Python based game. Taz is a small library handling the switching of scenes and making sure your scenes get updated and rendered on every tick.

The scenes will be organised in a game stack, which will automatically be updated for all scenes whenever a new scene is registered with the Taz library. The user has the opportunity to force scene changes in pushing or popping from the game's stack. Whenever the user pops the last stacked scene the game will come to an end.

Taz works independent of any python based game library.

1.1 Introduction

This section is to inform about the purpose of Taz and some basic concepts of game development.

1.1.1 What is a game loop?

Let's assume you are playing on of the many shooters out there. On each frame the game listens to your commands, like "move forward", "shoot", "jump". The game needs to capture these inputs and respond accordingly in letting your character do the respective thing and draw this to your screen. But even if you stand still the game keeps drawing the surroundings. There might be still movements in your screen, like trees bending in the wind or rain falling from the sky. So the game asks for information over and over again. This "asking for information" is called the game loop. Somewhere in your code there will be an pseudo-infinite loop which asks for these **updates** and **renders** them to your screen. Of course the game doesn't run for an infinite time, so somehow you would have to manage to tell the game-loop to stop at a certain point and exit the game, which why I called it **pseudo**-infinite. Following an example of such a game loop.

```
def mainloop():
    delta_time = 0
    while True:
        start_time = current_time()

        level1.update(delta_time)
        level1.render()

        end_time = current_time()
        delta_time = end_time - start_time
```

This is a very common way to define such a main-loop. In this example the time between each iteration is calculated and passed to the update-function. Why you should do that can be read here: <https://www.scirra.com/tutorials/67/delta-time-and-framerate-independenc> What exactly you put in the **update**- and **render**-function will be explained in the following pages. For now it is enough, that you understand the basic functionality about it.

1.1.2 What is a scene stack?

In the above code-block the game-loop always calls the **update**- and **render**-function of an object called level1, which most likely represents the first map of a game. But what if you want to have multiple maps or a title scene, with an awesome intro. You would only be able to show these when they are called from within your level1. But then level1

would not be responsible for just the first map, but also a lot of other things. To make sure you can show as much different levels and menus you want, we would have to alter the code-block to something like:

```
def mainloop():
    delta_time = 0
    while True:
        start_time = current_time()

        current_scene.update(delta_time)
        current_scene.render()

        end_time = current_time()
        delta_time = end_time - start_time
```

Basically, what we did is getting rid off the definite call of the “level1”’s functions and rather tell the game to update and draw whatever scene you want the game to currently draw. As for now we introduced the term “scene”. But what is a scene exactly? Well, a scene is pretty much everything you can think of. It can be a title scene, where the mentioned intro is playing. It could also be the main menu, any level of your game or even the options menu. In each and every scene there are different functionalities, which must be coded and different inputs to be captured. In the title scene you might only allow “Press Esc to skip”, whereas in the menu you probably want some buttons capturing clicks or you want your character to interact with objects in your level.

The problem however is to tell the game, which scene specifically to render. You would have to change the *current_scene*, when you want to switch the scenes and that is exactly what is done, when using a scene stack. The scene stack represents the currently loaded scenes such as level and an options menu. To decide which scene to **update** and **render** the game looks at the stack’s top scene and calls its **update** and **render** function. This way the top pushed scene is always the active scene. When a scene is no longer needed, let’s say for example when you close the options menu and want to get back to the level it gets popped from the stack and new top scene is now active again.

	Options Scene						
	Level 1 Scene			Level 1 Scene	Level 1 Scene		
Stack:	<Stack is Empty>	Main Menu Scene	Main Menu Scene	Main Menu Scene	Main Menu Scene	Main Menu Scene	<Stack is Empty>
Function:	Start Game	push Menu	push Level	push Options	pop Options	pop Level	pop Menu
Use Case:	Start Game	Show Menu	Start Playing Level	Change Name	Play Level	Finish Level	Quit Game

With that in mind the code-block can be altered once more to the following:

```
def mainloop(self):
    delta_time = 0
    while True:
        start_time = current_time()

        self.get_top_scene_of_stack().update(delta_time)
        self.get_top_scene_of_stack().render()

        end_time = current_time()
        delta_time = end_time - start_time
```

1.1.3 What is Taz?

Taz is a small library handling the switching of scenes and making sure your scenes get updated and rendered on every tick.

The scenes will be organised in a game stack, which will automatically be updated for all scenes whenever a new scene is registered with the Taz library. The user has the opportunity to force scene changes in pushing or popping from the game’s stack. Whenever you pop the last stacked scene the game will come to an end.

1.1.4 How can Taz help me organizing my game code?

Usually when starting from scratch you have to setup the backend of your game first, before you can go and implement whatever features your game should have, which means you would have to setup the game's main-loop and the scene-stack to get going. With Taz you have the opportunity to import this backend, so you won't have to think about it. All you can do is setting up your scenes and get going with the awesome features you have planned. Taz provides you with the main-loop and makes sure your scenes are updated on each tick. With the library there is a set of just a few functions you will have to worry about. These functions include pushing to and popping from the stack as well as registering your scene with the game and of course everything to get your scene going. You will be provided with a scene base class and all you have to do is implement some abstract methods, such as "initialize", "tear_down", "update", "render". What these methods are for is shown on the next few pages to get you going with your very own "Taz"-based game. Need another reason: Taz is working independent of any python based library, so whatever you are using, be it PyGame, PyOgre, PyOpenGL, Taz will be ready for use after setting up a few things.

1.2 Using the "Game" class

This section is to present the Game class and its functionalities

1.2.1 What is the Game class

The Game class is responsible for managing the scene stack and running the game's main-loop. It is responsible for calling the update- and render-functions of a scene and shuts down the game if the last scene has been popped from the scene stack.

1.2.2 Instancing the Game class

To getting started with Taz you first have to create an object of this class. The constructor takes two parameters, which are the update_context and the render_context. The game passes these contexts to the respective update- and render-function on each tick. The contexts are represented as dictionaries holding specific items you want every update- or render-function of your scenes to get on each tick. You might be using the update_context to pass in the delta time or input events for example. To see an example let's look at the following code block:

```
from taz import Game

if __name__ == "__main__":
    update_context = {
        "input_fob" : sys.stdin
    }

    render_context = {
        "output_fob" : sys.stdout
    }

    game = Game(update_context, render_context)
    example_scene = ExampleScene("ExampleScene")
    game.register_new_scene(example_scene)
    game.push_scene_on_stack("ExampleScene")
    game.enter_mainloop()
```

In this case the update and render context just hold the information which input and output to use. This method is used in the integrationtest, which is represented as text-based adventure. So in this particular example the render-function always writes to sys.stdout, while the update-function reads from sys.stdin. You can setup these contexts to whatever needs you have in your application and access them directly through the Game instance.

1.2.3 Registering scenes with the game

Whenever a scene is created the game needs to know about it, thus it needs to be registered with the game. The

`Game.register_new_scene(scene)`

Parameters `scene` – This is the scene, which should get registered with the game

Raises `Game.SceneAlreadyRegisteredError` If a scene is registered multiple times an error is raised.

Before a scene can be pushed on the game's stack it must be registered with the game. This method should be called right after a scene has been created.

function gives you the possibility to register your scene object. The registration of scenes needs to be done before the scene gets pushed to the stack, otherwise a **Game.NoRegisteredSceneWithThisIDError** is raised.

1.2.4 Using the scene stack

Using the scene stack is fairly simple. To push a scene on the you use the

`Game.push_scene_on_stack(ident)`

Parameters `ident` – This is the string-identifier of the scene.

Raises `Game.NoRegisteredSceneWithThisIDError` If the scene given hasn't been registered with the game this function raises an error.

This function pushes a given scene to the stack. The scene is given by its identifier and must have been registered with the game before it can be pushed on top of the state.

with the scene's identifier. Whenever a scene is pushed on the stack its **initialize**-function is called. When you push a scene on top another this scene gets paused. To pop a scene you use

`Game.pop_scene_from_stack()`

Raises

- **Game.StackEmptyError** – If the stack is empty and this function is called an error is raised.
- **Game.GameExitException** – If the last item of the active stack is popped an `GameExitException` is raised and the game is shut down.

This scene is used pop the current scene from the game's stack. When this is called, the scene's `tear_down` function will get called in order to destroy the scene. If the popped scene was on top of another scene, the other scene's `resume`-function is called, to re-initiate the state of this scene.

Whenever you pop a scene, which is on top of another scene that scene's **resume**-function is called in order to re-initiate this scene.

When last scene of the stack is popped the game is closed.

1.2.5 Sharing data between all scenes

TODO

1.3 Inheriting the “Scene” class

This section is describe how to inherit from the Scene class and how to use its methods. For an example please scroll down to the end of this section.

1.3.1 Instancing the Scene class

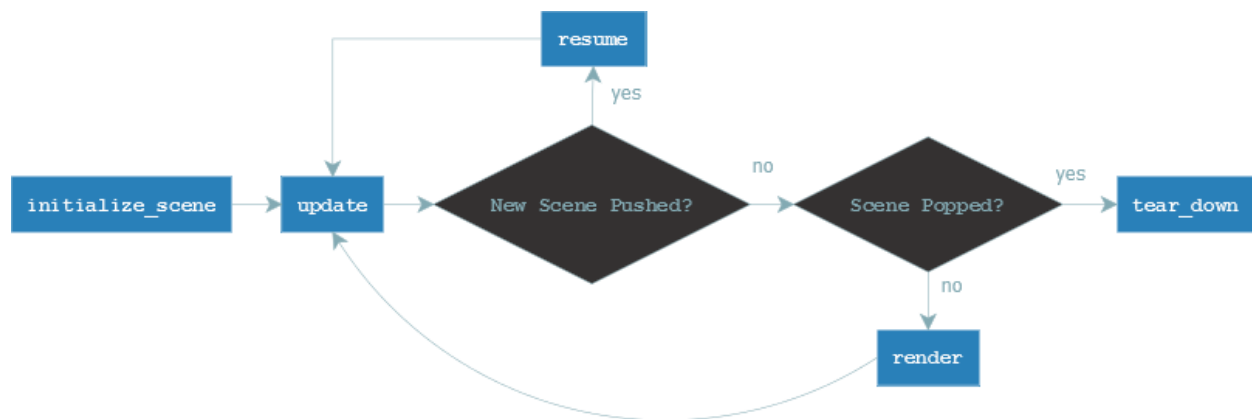
After you’ve setup a game instance, you need some scenes to start your game. The scene class gives you basic functionality to start with. First of all you need to create a sub-class of Taz’ Scene class and implement all its abstract methods. The method names of these abstract methods should already give you the idea of their purpose. Even if not, the explanation ffor each and every one of these follows in this section. However, when you create an object of your sub-scene you need to provide a unique identifier for each scene you create. This identifier is used to find this particular scene in the game’s registered scenes as well as to push it on the scene’s stack. In order to not break the code’s functionality you need to call the super constructor, if you intend to override the scene’s `__init__` method. Too see how this is done see the following code-block of an example sub-class:

```
from taz import Game, Scene

class MyScene(Scene):
    def __init__(self, identifier):
        super(MyScene, self).__init__(identifier)

    def initialize_scene(self):
        self.player = 0
        self.screen = 0
```

1.3.2 A Scene’s life cycle



What to put in “initialize”

This function is called whenever a scene is pushed on top of the game’s scene stack and should build up a certain scene. In here you might build up a level or a main menu and its buttons. Everything that needs to be done when calling the scene should be put in here.

```
from taz import Game, Scene

class MyScene(Scene):
```

```
def __init__(self, identifier)
    super(MyScene, self).__init__(identifier)
    self.player = 0
    self.screen = 0

def initialize_scene(self):
    self.player = Player()
    self.screen = Screen()
    self.create_nice_map()
```

The scenes event cycle

In the game's main-loop the scene, which is on top of the scene stack is always the active one and its **update**- and **render**-methods are called on each and every tick. Why are there two functions when you call them right after another you ask? You will get an answer in the following sub-sections.

Using the Update method

The update method is responsible for updating (who would've thought of it) the game's or scene's status. This function should listen and respond to input events, calculate new positions of players, basically doing most of the things that happen within a certain scene.

```
from taz import Game, Scene

class MyScene(Scene):
    def __init__(self, identifier)
        super(MyScene, self).__init__(identifier)
        self.player = 0
        self.screen = 0

    def initialize_scene(self):
        self.player = Player()
        self.screen = Screen()
        self.create_nice_map()

    def update(self):
        key = self.capture_input()
        self.player.move_player(self.game.update_context["deltatime"], key)
```

Using the Render method

The render method is the function to get your game to life. It is responsible for and just for drawing/printing everything to your screen. It is NOT responsible for any calculations, as this is update's job.

```
from taz import Game, Scene

class MyScene(Scene):
    def __init__(self, identifier)
        super(MyScene, self).__init__(identifier)
        self.player = 0
        self.screen = 0

    def initialize_scene(self):
        self.player = Player()
```

```

self.screen = Screen()
self.create_nice_map()

def update(self):
    key = self.capture_input()
    self.player.move_player(self.game.update_context["deltatime"], key)

def render(self):
    self.screen.draw(self.player)

```

What to put in `tear_down` and `resume`

When leaving a scene there might still be some things to clean up before you can destroy the scene. Everything you need to be cleaned up before popping a scene should be put inside the `tear_down` function.

In some cases you might want to store your current status of a level, e.g. when switching to an options menu. When returning back to the game you most likely don't want to start over the level but instead keep on playing where you left of. In fact Taz implementation allows you to push a scene on top another. When this top scene is then popped again the scene following is re-activated and the new top scene. When this happens the `resume`-function of that scene below the popped one is called, so you can put everything in here, which should get re-initiated.

```

from taz import Game, Scene

class MyScene(Scene):
    def __init__(self, identifier):
        super(MyScene, self).__init__(identifier)
        self.player = 0
        self.screen = 0

    def initialize_scene(self):
        self.player = Player()
        self.screen = Screen()
        self.create_nice_map()

    def update(self):
        key = self.capture_input()
        self.player.move_player(self.game.update_context["deltatime"], key)

    def render(self):
        self.screen.draw(self.player)

```

1.3.3 Adding data to a single scene

TODO

1.4 Examples

1.4.1 Example Usage with PyGame

TODO

Contact:

2.1 Contact

If you have problems or suggestions please contact [<KlingmannM@gmail.com>](mailto:KlingmannM@gmail.com) or visit us at [irc.freenode.net#pygame2](irc://irc.freenode.net/pygame2)

Taz API Documentation

3.1 Taz API Documentation

class `taz.game.Game` (*update_context*, *render_context*)

This class is the main game class. It contains the stack with all registered scenes and is managing the game loop.

Parameters

- **update_context** – This update_context is passed to the update-function of all scenes
- **render_context** – This render_context is passed to the render-function of all scenes

To construct an instance of this class, a `update_context` and a `render_context`, depending on the used underlying engine is used. To ensure functionality call the base class' constructor when overriding the `__init__`-function.

pop_scene_from_stack ()

Raises

- **Game.StackEmptyError** – If the stack is empty and this function is called an error is raised.
- **Game.GameExitException** – If the last item of the active stack is popped an `GameExitException` is raised and the game is shut down.

This scene is used pop the current scene from the game's stack. When this is called, the scene's `tear_down` function will get called in order to destroy the scene. If the popped scene was on top of another scene, the other scene's `resume`-function is called, to re-initiate the state of this scene.

push_scene_on_stack (*ident*)

Parameters *ident* – This is the string-identifier of the scene.

Raises **Game.NoRegisteredSceneWithThisIDError** If the scene given hasn't been registered with the game this function raises an error.

This function pushes a given scene to the stack. The scene is given by its identifier and must have been registered with the game before it can be pushed on top of the state.

register_new_scene (*scene*)

Parameters *scene* – This is the scene, which should get registered with the game

Raises **Game.SceneAlreadyRegisteredError** If a scene is registered multiple times an error is raised.

Before a scene can be pushed on the game's stack it must be registered with the game. This method should be called right after a scene has been created.

class `taz.game.Scene` (*ident*)

This is the abstract base class of all scenes. All Scenes must be derived from this and override the abstract methods. On each tick the render and update functions are called. They get passed the `update_context` and `render_context` passed to the Game object on creation.

When creating an instance of this class a string-identifier must be passed to the constructor in order to be able to call the function through the game stack and identify it in the game's registered scenes. :param ident: This is the string-identifier of the Scene :return:

get_identifier ()

Returns This returns the string-identifier associated with this object

initialize ()

This method builds up the scene

pause ()

This function is called whenever a scene is pushed on top of the active one

render ()

This method draws the game's screen

resume ()

This re-builds the state before pausing it (e.g. after returning from an options menu)

tear_down ()

This method cleans up the scene before destroying it

update ()

This method updates the game's logic

Indices and tables

- `genindex`
- `modindex`
- `search`

t

`taz.game`, [13](#)

G

Game (class in taz.game), [13](#)

get_identifier() (taz.game.Scene method), [14](#)

I

initialize() (taz.game.Scene method), [14](#)

P

pause() (taz.game.Scene method), [14](#)

pop_scene_from_stack() (taz.game.Game method), [6](#), [13](#)

push_scene_on_stack() (taz.game.Game method), [6](#), [13](#)

R

register_new_scene() (taz.game.Game method), [6](#), [13](#)

render() (taz.game.Scene method), [14](#)

resume() (taz.game.Scene method), [14](#)

S

Scene (class in taz.game), [13](#)

T

taz.game (module), [13](#)

tear_down() (taz.game.Scene method), [14](#)

U

update() (taz.game.Scene method), [14](#)