|  |  |  |
| --- | --- | --- |
|  |  | HR Minigames  Documentation |



Written by Dennis Kievits (0946572)

# Table of contents

* **Introduction**
* **Project Structure**
* **Menu Layer**
* **GamePlayer**
* **Minigame**

# Introduction

I have chosen to write this document in order to help my other team members with developing their minigame. The reason I am the one writing the documentation is because I structured the code in this way myself, and that makes me the one that can thoroughly explain my thought process that went into it.

# Project Structure

*Note: a more detailed documentation (for each file) can be found on a different page.*

The project is structured in the following way, there are 2 folders beneath the project’s root directory.

The folder ‘Minigames’ contains each team member’s own minigame, this is where each team member can split up their code into smaller parts, allowing them to be more focused on developing their minigame instead of focusing on the entire project at once.

The folder ‘Shared’ contains all shared game logic and assets. Use this folder to store media and/or logic that should be shared between different minigames.

There are also a few other python files that are core to the game’s logic.

* GamePlayer.py

This file contains the logic for when the player enters one of the minigames. Both the preview state and playing state are handled in this file.

* Main.py

This file is the main execution file, and should be executed by the python runtime. Main.py loads all other files and runs the game.

* Menu.py

This file contains the logic for when the player is on the menu screen, it is drawing all the arcade machine and showing each minigame’s mini preview. The menu also opens a GamePlayer for when a minigame is selected.

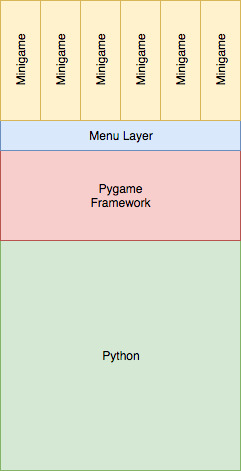
* Minigame.py

This file contains all of the logic and methods a minigame is supposed to handle. These methods are being executed by the GamePlayer. As a developer you will be modifying your own methods in order to add logic and a visual representation of your minigame.

# menu layer

The menu layer is used to abstract the minigames from the pygame itself. Pygame is a framework build for python that makes it easy to write games in a python environment. However the framework gives full control over the entire window space and how it’s going to be used.

The menu layer is placed on top of pygame in order to sandbox each minigame in its own space.



*This figure shows how the menulayer interacts with pygame and each minigame.*

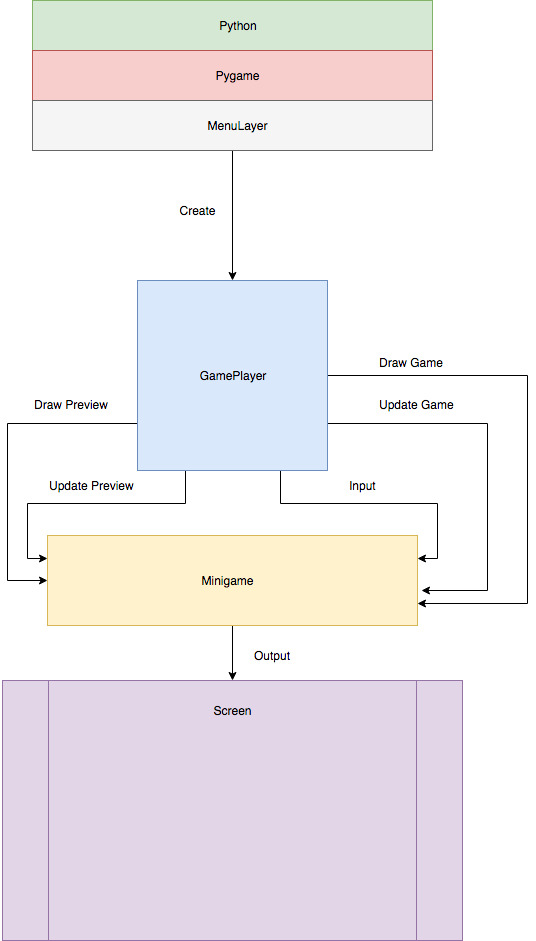
The menu layer holds each minigame in its memory and shows each mini preview on it’s menu scene.

When the user selects a minigame from the menu, the menu instantiates (creates) a ‘GamePlayer’ object.

# Gameplayer

The GamePlayer is responsible for showing the big preview of each minigame, as well as playing the game itself.

The GamePlayer has 2 states, the preview state, which draws and updates the minigame’s preview, and the play state, which provides user input, draws the game and updates the game.



# minigame

In total there will be up to 6 minigames. Each different minigame uses the Minigame class to inherit (pass down) behavior. Each minigame consists of a few things.

* Name
  + What is the game called?
* Author
  + Who made the game?
* Identifier
  + This value is used to determine what arcade machine color to apply to the game.

Each minigame will be given a few methods that you can override to change the behavior. These methods will be called by the MenuLayer/GamePlayer.

# When a player starts this minigame

    def enter(self):

        raise NotImplementedError("You need to override the enter method on your minigame.")

*This method gets called when the player pressed the return key when the GamePlayer is in its preview state. Use this method to initialize/reset your game state.*

# When a player leaves this minigame

    def leave(self):

        raise NotImplementedError("You need to override the leave method on your minigame.")

*This method gets called when the player pressed the escape key when the GamePlayer is in its playing state. Use this method to save any data before exiting the game.*

def handleEvents(self, events):

        raise NotImplementedError("You need to override the handleEvents method on your minigame.")

*This method gets called when the GamePlayer is in its playing state. Almost all events from pygame will be passed down to this method, the ‘events’ parameter contains an array of events.*

# Gets called on every frame

    def update(self, dt):

        raise NotImplementedError("You need to override the update method on your minigame.")

*This method gets called when the GamePlayer is in its playing state for each frame that is going to be rendered. Use this method to update your game’s state, the parameter ‘dt’ contains the deltatime. You can use the deltatime to consistently update values regardless of the amount of frames the game is rendering.*

# Draw the current game state

    def draw(self, surface):

        raise NotImplementedError("You need to override the draw method on your minigame.")

*This method gets called when the GamePalyer is in its playing state for each frame after the update method. Use this method to draw your current state to the provided surface, the surface will render on the screen.*

The Minigame class might have some more information/methods, but most of them should be self-explanatory.