**Obligatory Assignment 3**

Fall ‘23

Odin Bjerke

[obj021@uit.no](mailto:obj021@uit.no)

# **Introduction**

This report will cover central design choices, implementations and findings made during the development of an Amiga-style game, built in Python using the open source Pygame library.

As a precursor, this document is written under the assumption the reader has a fundamental understand of both Python and the Pygame library.

# **Technical Background**

Prior to diving into specific topics related to the application at hand, this section will cover some relevant technical topics on an abstract basis.

**Simple DirectMedia Layer**, better known as SDL, is a development library designed to provide convenient, low-level access to a computer’s analogue components. Examples of the libraries core features include registering key input, playing sound and applying pixels to a canvas displayed by the screen. The open-sourced library is written in C, and thus provides next to native cross-platform compatibility.

Accessing SDL directly through high level languages such as Python, is not remotely convenient. Python and SDL do share a common attribute, however, as Python is also written almost exclusively in C.

**Pygame** solves this issue of convenience by providing easily accessed SDL features through Python, as well as providing a considerable number of built-in classes and methods in the process. Directly written on top of SDL2, Pygame has tolerable performance, and more importantly provides convenience to the degree that it is a typical choice for teaching programming to children. The Pygame documentation is also a rarely found combination of extensive and comprehensible.

# **Design and Implementation**

The main philosophy throughout implementation was simple enough, and can be described by the following ruleset:

* Isolate responsibility of features across classes to the degree it is possible.
* Rather than modifying a class to add more features, extend it.
* If feasible, write classes and methods that can be repurposed, or reused through inheritance.

The following table is meant as a general overview of key classes, as well as their purpose and associated areas of responsibility. This table mainly covers classes who have responsibilities over other objects. In the table, I define ‘children’ as other key objects that are directly controlled and/or created by the class, not including any potential ‘further’ successors of those children. I believe this table provides a better overview of the program logic than an enormous class diagram.

|  |  |  |
| --- | --- | --- |
| Class | Category of class | **Core Responsibilities** | **Children** |
| *App* | \* General Initialization  \* Root Game Loop  \* Menu  \* Cleaning up memory | \* Window  \* Timer  \* Map  \* Menu Containers |
| *Container(s)* | \* Positioning, updating  and drawing UI objects | \* Buttons, text, bars and/or other containers |
| *Window* | \* Setting up the core surface  \* Distribution of subsurface  area | \* Map Subsurface |
| *Timer* | \* Tracking of time  \* Ticking Pygame clock  \* Custom / timed events  \* Global, Timer-related UI | \* Time Segments  \* Time UI Container |
| *Map* | \* Creation of sprites and UI objects directly related to the map surface  \* Collision checks  \* ‘Inner’ loop | \* Player  \* Blocks  \* Turrets  \* Status Bar Container |
| *Turret* | \* Creating and updating projectile spawners | \* Projectile Spawner |
| *Projectile Spawner* | \* When and where to fire a projectile | \* Projectiles |

Applying these principles, I found that applying this principle of separated responsibility provided quite a few advantages, and some drawbacks.

The absolute greatest advantage came to light when I inevitably needed to adjust existing code or method functionality. As an example, outside of selecting the config file through the menu, the App class never directly interacts with the player. In fact, no other objects interact directly with the player – apart from the map. This means that if the player class is modified, the only other class that might have to be changed, will be its parent – the map. The config files are carried by the App down to where they belong, and adding new config settings through this does not require any change outside the player class itself.

In an application such as a game, modifying existing code is inevitable to a degree, and I found my approach to allow for quick and easy modification when needed.

The other major advantage is knowing where to look for any potential bugs, instead of having to filter through an enormous game loop containing every piece of logic in one place.

As far as drawbacks go – I found my design approach did result in an extensive amount of code, especially when considering the config files.

# **Experiments**

Throughout the development process, I occasionally ran cProfile to check for bottlenecks, if any.

The program runs with the Pygame tick.busy\_loop() as default, which uses a lot more resources, but I found it stabilized the framerate significantly, and before adding projectiles and the background image, the app ran at 500-600fps without issues.

After adding projectiles, I found the pixel perfect collision detection of bullets through mask\_collide consumes a rather hefty amount of resources, but I’m nonetheless confident the method used is fairly optimal. Because bullets collide with terrain blocks, which there are an abundance of, there are simply a staggering number of masks to check for collision – every frame.

# **cProfile Results**

This test is done using an fps limit of 250, using a busy\_loop.

To summarize the results, blitting the background image and projectile collision use more resources than everything else – combined, and by a significant factor.

(Continued on the next page)

Text

Description automatically generated with low confidence

Text

Description automatically generated with medium confidence

This report and an additional one can also be found in cprofile.txt

# **Discussion & Summary**

In my assignment, I have deviated somewhat from the assignment requirements. Namely, the PG\_Common.py file contains directly called functions, rather than a class with methods. On top of this, all classes do not contain docstrings. I have no real excuse for this other than the sheer number of classes implemented in my assignment, and this is also the reason why I felt a table was more appropriate than a class diagram. During the last remaining time, my priority was getting the menu to a functional state.

In an effort to create readable code, class, constants and variables names are extremely verbose.

For a summary of the configuration files, I would recommend starting at config.cf\_map.py, which essentially bundles of other configs used by the map. The program, as well as buttons, dynamically adjust themselves to added players or maps. To try this out, I would recommend creating an entry in the cf\_map dict, following the exact format of the two existing ones, and tweaking values, colors or the background to your liking.

If I had the opportunity to, I would have spent more time both making a more detailed report – as well as adding the proper docstrings, but I will admit I got a bit carried away and simply ran out of time.

There could be some bugs, specifically regarding the post-game menu, but I have tested and fixed what I could to provide what I would consider an alpha-state of a functional app. For the docstrings that exist, I believe I have updated them when modifying code, but as a disclaimer – there could be remnants of prior implementations. A few classes were completely refactored, up to several times.

To summarize in short, I’m convinced I have adequately fulfilled the task at hand despite a few shortcomings. To quote one assignment requirement, “… we want to make sure that everyone has seen and used inheritance in a natural setting.” – I feel certain my code and this report leave no doubt I do understand these concepts.

See README.md for version requirements.