Advanced Programming

Nelson Heinzelmann
Gaëtan Gallandat
Elyès Ghali
Faculté des Hautes Études Commerciales
Université de Lausanne

Abstract. This document presents a trading game we developed in the context of the advanced programming course at the University of Lausanne. In our game, the player makes investment decisions based on four variables: Fed rate, inflation, bond prices and stock prices. In this interactive scenario, the player tactically allocates their resources between bonds, equities, and cash within their investment portfolio. Each round sees fluctuations in the four graphical representations, demanding the player's constant portfolio adjustments. Victory is achieved by reaching a predetermined portfolio return by the game's conclusion. Our aim is to manifest our enthusiasm for finance and gaming through this engaging project.

1. Introduction.

Programming is an exciting and rapidly evolving domain that revolves around creating, designing, and implementing software applications. It is the process of instructing computers to perform specific tasks by writing sets of logical instructions. A fundamental skill that powers the digital era we live in, enabling the development of computer programs, websites, mobile applications, artificial intelligence systems, and much more. The programming field offers a world of possibilities and endless opportunities.

The field of finance is a cornerstone of the global economy, with its tentacles reaching into every sector and influencing every business transaction. From determining investment strategies to guiding fiscal policies, finance plays a pivotal role in shaping the economic landscape. Parallelly, the advent of programming in finance has radically reshaped the industry, paving the way for sophisticated quantitative analysis, risk management, and portfolio optimization.

As finance students, we have always been captivated by the intricacies of financial markets and the profound role they play in the global economy. Simultaneously, we also share a deep passion for video games, an arena that constantly challenges us and keeps us engaged. Driven by these dual interests, we decided to embark on an innovative project that merges our academic pursuit in finance with our love for gaming. There are different types of video games but there are three key elements we found in every project: A concept, a gameplay and animations. The concept is the universe, the story and the goal of the game. The gameplay is the set of possibilities the player can use to interact with the game. Animations are the music, images and texts of the game.

A famous example of a video game is Pong. Pong was the first video game with commercial success developed by the american Atari in 1972. Pong is a two-dimensional sports game that simulates table tennis. The player controls an in-game paddle by moving it vertically across the left or the right side of the screen. Another player controls a second paddle on the opposing

side. Players use the paddle to hit a ball back and forth. The goal is for each player to reach eleven points before the opponent: the points are earned when one fails to return the ball to the other. This game is based on a rather simple concept and gameplay. Also, the amount of animation is very low as they consist of only the movements of the ball and the paddle. Nevertheless, we found this simplistic approach to game-design very interesting and based our own game on the same concepts.

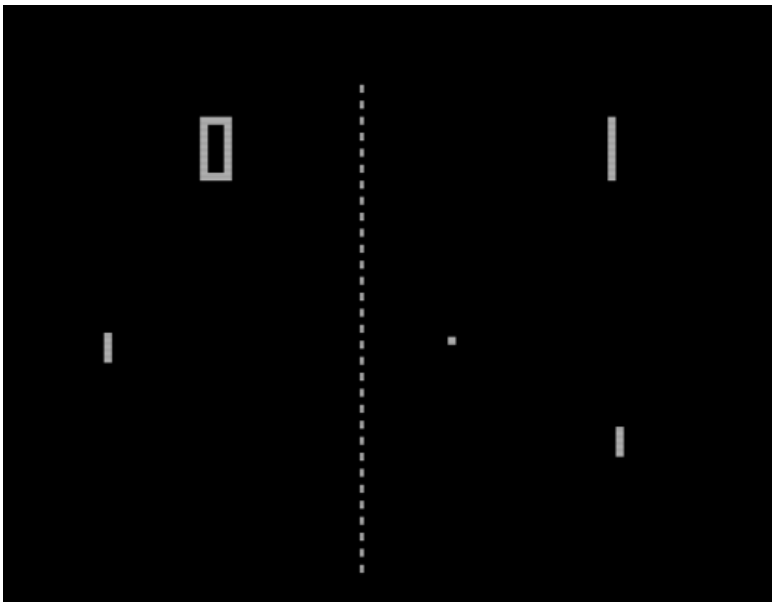

Figure 1.1. *A Gameplay of Pong*

2. Research question.

2.1. Problem. We acknowledge two significant challenges in the development of our trading game. First, the financial markets are known to be complex. Second, a good video game has to be simple and fun. A trading game should mix complexity and simplicity. It is not an easy task. Our research question is the following: What are the key components of creating a trading game in Python? By developing the application, we answer this exact question.

2.2. Solution. The field of finance provides numerous insights that can help us create a Python-based trading game, with Harry Markowitz's Modern Portfolio Theory being instrumental in shaping our gameplay mechanics. Theoretically, one could plot the standard deviation of any possible portfolio, comprising these stocks, against their expected return given a time series of stock prices. We plan to utilize this concept by generating four-time series curves, composing a portfolio of three assets, and creating a meaningful link between the two. This allows for optimal investment in our portfolio based on the time series curves, while still maintaining a level of simplicity.

What we are aiming for is to create sensible relationships in terms of direction and more

between the four variables while maintaining simplicity in these relationships. The goal is not to capture with high precision the effect of a variable delta, but to use Python as a robust platform for this type of project due to its simplicity of language, its existing libraries for data manipulation, and its object-oriented programming capabilities. Additionally, the Pygame library provides easy solutions for game development, which is particularly beneficial for our project that doesn't require high rendering capabilities like Unreal Engine 5 or Unity would provide. Also note, these latter are based on C++ and not Python.

3. Methodology. The first challenge of this project was Data Generation. Indeed, for the advanced programming project, we wanted to be able to develop an AI for this game, so we had to implement relations that were not too complex but have some randomness still to avoid everything being completely deterministic. Also, randomness is a way to make the game more engaging and have a longer lifespan. Indeed, if the game could be played like a decision three, there would be a definite amount of games before you get to the optimal solution. This wouldn't be a good way to approach a game that should develop some intuition about the economy.

For all these reasons we decided to use ARMA processes to generate our Data. Also, note that the initial values are randomly generated based on the following laws to ensure that each game has a completely different setup.

$$\text{Inflation Rate: } r_0^{inf} \sim \mathcal{U}(0.02, 0.05)$$

$$\text{Fed Rate: } r_0^{fed} \sim \mathcal{U}(0, 0.1)$$

$$\text{Stock Price: } p_0^s \sim \mathcal{LN}\left(\log\left(\frac{\mu_\text{s}^2}{\sqrt{\sigma_\text{s}^2+\mu_\text{s}^2}}\right), \sqrt{\log\left(\frac{\sigma_\text{s}^2}{\mu_\text{s}^2}+1\right)}\right)$$

$$\text{Bond Price: } p_0^b \sim \mathcal{LN}\left(\log\left(\frac{\mu_\text{b}^2}{\sqrt{\sigma_\text{b}^2+\mu_\text{b}^2}}\right), \sqrt{\log\left(\frac{\sigma_\text{b}^2}{\mu_\text{b}^2}+1\right)}\right)$$

With this being set, we could work on the gameplay part of our project. At first, we had to set up a global structure for our project. We decided to split our code into 4 files that would all serve a specific purpose such as running the game, providing all utilities for the game, setting up the different windows of the game and defining the game loop. Our code's overall structure is the following:

```
Programming_Project
├── main.py
├── utils.py
├── init.py
├── scenes.py
└── assets/
    ├── sounds/
    │   ├── background.mp3
    │   ├── mouse_hover.mp3
    │   ├── page_turn.mp3
    │   └── music.mp3
    ├── sprites/
    │   └── Background.png
    └── text/
        └── Scenario.txt
```

We will explain more precisely how all these files work in the next section. The next step in our project was to define how we wanted our game to look. We decided to go for a very minimalistic and functional design. It would have 4 Scenes: the Title Scene to start the game, the Main menu to be able to access the game settings the players need to know to be able to play, and the scenario scene which explains the concept of the game and the actual game scene.

After this, the work was a lot of back and forth between the different Python files to ensure everything was working correctly. The main parts were to set up a game clock to ensure the game was being played one quarter at a time. Create separate classes for each scene, and add all methods related to the Portfolio, such as calculating the weights, being able to buy stock, or avoiding negative weights. The non-negative weight condition was to make it more realistic and to avoid scenarios where the AI is having unrealistically high shorting amounts. The final step of our project was cleaning up the code to avoid having too many code duplications, making the code reusable and modulable in case we want to complexify it and also adding a testing class to verify that the Portfolio class is correctly defined. We also defined a threshold for the minimal return to win the game.

4. Implementation. As mentioned, our game runs on python, more specifically through the pygame library. It also requires the installation of several other libraries: sys, pathlib, os, numpy, random, unittest. In order to play the game you only have to run the __init__.py file.

4.1. Front-end. The front end of the game is running through a pygame interface. It starts with the TitleScene which just asks the player to press enter to get to start the game. After that, he gets the Main menu scene which displays 3 clickable options: Play, Option, and Exit. This scene has been added to ensure the player has a way to access the commands used to play. By clicking on Option he will be taken to a scene that shows the predefined keyboard inputs to play. The third scene is showing the game's scenario, this is just a transition scene to add some overall context. Then there is an option to navigate to the main game scene where

4 graphics are rendered, one for each variable. The data projected consist of 12 quarters preceding the actual game state to ensure the player has enough information to make his portfolio management choices. All information such as portfolio value, and current weights are also nicely shown on the screen. At any moment in time, the player can press ESC to pause the game where he can then either go to the options, quit the game, reset the game or just resume it. Finally, after the 18 quarters are played an end scene is rendered to show if the player won or lost the game.

4.2. init.py. This file just serves as a way to run the game. All it does is create a game instance using main.py and then run this game instance. The main block is just to ensure that the game instance is only created and run when `init.py` is executed.

4.3. main.py. The code in `main.py` has the following architecture:

Game Configuration: The `Config` class defines the game configuration settings such as screen width, height, frames per second, and title. It also includes attributes for the game screen size.

Scene Management: The `SceneManager` class is responsible for managing scene transitions within the game. It maintains a reference to the current scene and provides methods for transitioning between scenes, restarting the game, and retrieving the stored scene. The stored part is important in case you decide to pause the game. It ensures that when you resume you are indeed taken back to the game state you were in and do not create a new game instance.

Main Game Class: The `Game` class is the main entry point of the game. It initializes the game configuration and screen using the `Config` class. It also creates an instance of the `SceneManager` and manages the game loop.

Game Loop: The game loop is the core structure of the game. It keeps the game running by repeatedly executing certain tasks. Within the game loop, the current scene's event handling, update, and rendering functions are executed. This ensures the execution of all inputs made by the player, and that everything is always rendered on the screen as it should.

Scene Execution: The `scene()` method in the `Game` class executes the event handling, update, and rendering functions of the current scene. It delegates these tasks to the `SceneManager` and the respective scene objects.

Overall, this is a key part of any game development using pygame. Setting up a loop is a fundamental step, and the `SceneManager` class ensures that there is a good transition between all the scenes.

4.4. utils.py. The **utils.py** file contains utility functions and a class related to game functionalities. Let's examine the design and structure of the code in terms of software architecture:

Game Screen Initialization: The `InitGame()` function initializes the game screen with the specified size and title defined in `main.py` using the `pygame` library. It sets up the screen

and returns it as a `pygame Surface` object.

Pygame Quitting: The `QuitGame()` function quits the game, optionally quitting the `pygame` library as well. It calls `pygame.quit()` and `sys.exit()` to correctly exit the game. This is mainly used in the several `handle_events` functions that will be defined in the several classes of the `scenes.py` file.

Sprite Loading: The `LoadSprite()` function is a way to simplify the loading of images. This allows loading all files in the `sprites` folder and returning them as a `pygame surface` object.

Text Scrolling: The `TextScroller()` function splits text into lines that fit within the specified rectangle using the specified font. It splits the text into lines and words, and then dynamically adjusts the lines to fit within the given dimensions. The final lines are returned as a list. This function has been solely added to have a clean rendering of the scenario in the `ScenarioGeneration` class of `scenes.py`.

Portfolio Class: The `Portfolio` defines all necessary tools to make investing possible. It provides methods for initializing the portfolio, calculating its total value, weights, stock and bond values, buying and selling stocks/bonds, and adjusting cash distribution based on asset values. Also, as we mentioned earlier, we wanted to ensure that the weights are always positive and sum to 1. To ensure this, all buying and selling functions have been set up in a way that they will always check if the player owns the required amount before selling or has enough of the counterpart when buying. The standard counterpart is always cash, but when it is no longer available, the other index is used.

4.5. scenes.py. The `scenes.py` is the main file as it will contain everything related to the actual Trading simulation. It consists of the four scenes mentioned earlier: `TitleScene`, `MenuScene`, `ScenarioGeneration`, `InvestmentScene`, as well as three scenes that are rendered depending on the player's inputs: `EndScene`, `PauseMenuScene`, `OptionScene`.

Generally speaking, because of the way we defined the Game class inside of `main.py`, every one of these classes has to have a `handle_events`, `update` and `render` function. `Update` is used when the screen needs to get the mouse position and recognize that the player has clicked on a specific location. The `handle_event` is used to recognize all the player's inputs and load an action based on these. Finally, the `render` function permits us to draw everything we want to display on the screen. Also, for each new class, the different assets are loaded in the `__init__`. To facilitate the placement of objects on the screen, the width and height of the screen using the screen object we created and passed previously.

TitleScene: This class represents the title scene of the game. This is the first scene the player gets when starting the game. It will just ask the player to press space to get to the next scene. It also initializes all standard attributes.

MainMenuScene: This scene renders three Buttons on the screen, they are clickable and their colour will change slightly when hovering on them with the mouse. A sound will also be played when doing this. This is done through the `update()` and `handle_event()` function which will retrieve the mouse position and if it coincides with the position of the

buttons the `handle_event` will trigger the sound and colour changing. It also initializes all standard attributes.

ScenarioGeneration: This class represents the scenario generation scene of the game. It initializes all standard attributes as well as the scenario text, text and button rectangles, text scroller, line and character indices. The text will then be smoothly rendered character by character using the `TextScroller` function defined in `utils.py`. Also, it ensures that the whole text is visible to the player. Again, most of it is done with the `update` and `render` functions. There is also a rectangle in the bottom left of the screen to go to the `InvestmentScene` which also has sound and colour effects.

Before talking about the `InvestmentScene`, we will cover the three optional scenes.

EndScene: This class represents the end scene of the game. It initializes all standard attributes plus the final portfolio value, and message. It has two more parameters than the other scenes as it will also take a boolean that indicates if the player won the game as well as the final portfolio value. Both of these are retrieved from the `Investment Scene`.

PauseMenuScene: This class represents the pause menu scene of the game. It allows the player to pause the game while playing. It can only be called in the `InvestmentScene` by pressing ESC. It also adds a method to restart the game by using calling the scene manager without the previously stored game state. This way a new `InvestmentScene` is initialized and we immediately navigate to the `InvestmentScene`. The player can also go to the options menu and in this case, the game state will be correctly stored when resuming the game later on.

OptionsScene: This class represents the options scene of the game. It only renders the game commands that have been predefined by us. There is also a Return button so that the player can get back to the last scene, meaning either the Main menu or the Pause Menu.

InvestmentScene: This class represents the investment scene of the game. This is the core scene for our gameplay. It initializes various attributes related to the investment simulation, such as portfolio instance using the `Portfolio` class of `utils.py`, graph parameters, initial data values and all parameters to generate the data. It also creates instances of the `LineGraph` class to display the graphs for inflation, fed rate, stock prices, and bond prices.

The `LineGraph` class is defined as an inner class within the `InvestmentScene` class. It encapsulates the functionality for rendering line graphs on the screen. It includes methods to normalize data, render the graph on a surface, and handle the rendering of tick marks and labels.

The `InvestmentScene` class includes a `generate_initial_data` method that generates initial data using a log-normal distribution and stores it in different history lists. This means that values for the period 0 are randomly generated. The `generate_next_data` method generates subsequent data using the ARMA processes we defined earlier on. Note that the `generate_initial_data` also calls the `generate_next_data` 11 times to have 12 quarters of Data at the beginning of the game. The `render` method will draw all the graph and portfolio values. The `handle_events` method reads all the players input such as increasing stock

weight. It also defines the keyboard settings for portfolio management to be very easy to use. Pausing the game is also handled in this method. When the player has reallocated his weights 18 times, he is transported to the End scene showing his result and final portfolio value.

5. Maintenance and update. The code structure and design in the financial simulator game facilitate future maintenance and potential extensions. For example, if we wanted to make the game more realistic by using more complex functions to generate data, we would only have to replace the existing function with the new methods. The rest of the code could stay unchanged.

Modularity: The code is modularized into separate files and classes, each handling a distinct responsibility. This makes the code easier to understand, modify, and test to understand, modify, and test.

Documentation: Inline comments and docstrings have been added throughout the code, explaining the purpose and functionality of the code blocks, functions, and classes. This increases the readability of the code, making it easier for future developers to understand the codebase and continue the project.

Unit Testing: The code is written to be testable, with unit tests covering critical game logic like the handling of transactions in the portfolio. Python's built-in unittest module is used for these tests. To run these tests you only need to execute the scenes.py file as the testing function is called in a main block inside of scenes.py. By having a comprehensive suite of unit tests, future developers can confidently make updates or modifications to the codebase without introducing unintended errors.

6. Results. The player has to run the init.py file to launch the game. Upon pressing the touch "Space", he enters the Title scene. Within this scene, the player can make their selection among the options of play, options, or exit by utilizing a mouse click. It is important to note that we did not include methods to update the relative positions of the object drawn on the screen. Therefore, the player should only resize his window at the beginning of the game if he wishes to do so to avoid any graphical bugs.

Figure 6.1. *Main Menu*

If he chooses the section Play, a scenario will be written down on his screen giving the player all the keys to understand the purpose of the game. He will start the trading game by clicking on "Invest now! Note that we observed that users on Mac Os encountered overlapping in the text in this scene. This problem did occur for Windows users, but we could not identify the source of this problem.

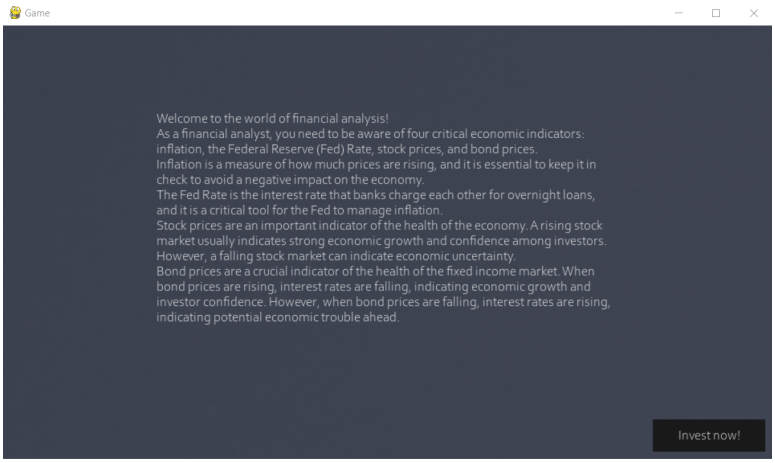

Figure 6.2. *The Games Scenario*

If he chooses the section Options, the player has access to the directional keys of the trading game. The directional keys are the following: 'C' to increase cash assets, 'S' to increase stock assets, 'B' to increase bond assets, 'Enter' to update periods, and 'Esc' to pause the game. If he chooses the section Exit, the player stops the run. He can get back to his previous screen at any time by clicking on return.

When he does finally start to play the game he will be on the main screen. There all the necessary information to be able to make his choices and rebalance his weight are presented. The core components are the graphics, they will always plot all the available Data up to the actual quarter starting from period 0. Note that the actual gameplay starts at period 12. Also,he will see his current wealth, how it is distributed among the 3 assets he owns and what they respectively represent as a part of the total portfolio, meaning their weights. He will get a starting amount of 500'000. If the player considers that he has correctly reallocated his weights, he can press "Enter" to update the period. The value for "Quarter" will be indented by 1, and all information will take into account the newly generated prices, inflation and Fed rate values.

If the player decides to pause the game, he can do so by pressing the "Esc" key. This will render the Pause Scene with its 4 clickable buttons as explained previously.

When the maximum amount of quarters is reached, he will automatically get to the Endscene showing his performance. The player has to attain a 505'000.- or more of portfolio value at the last period to win the game. If he succeeds to reach such an amount of portfolio return, a new scene appears and tells the player he wins the game. If he doesn't succeed to

reach such an amount of portfolio return, a new scene appears and tells the player he loses the game.

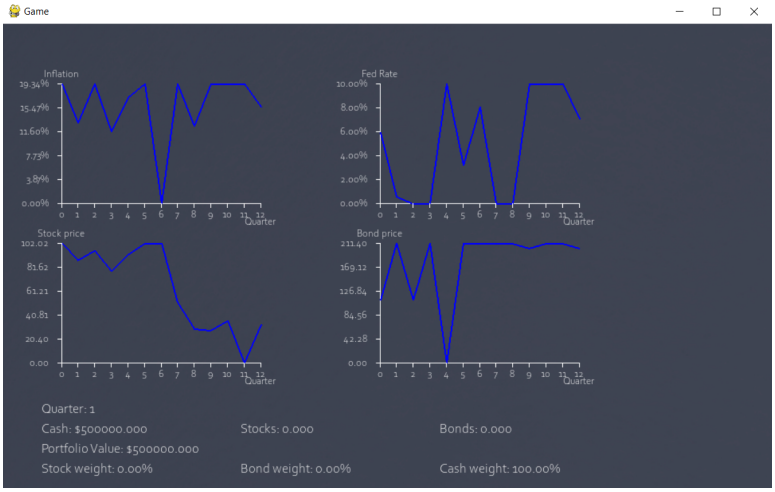

Figure 6.3. *The Investing Scene at Period = 0*

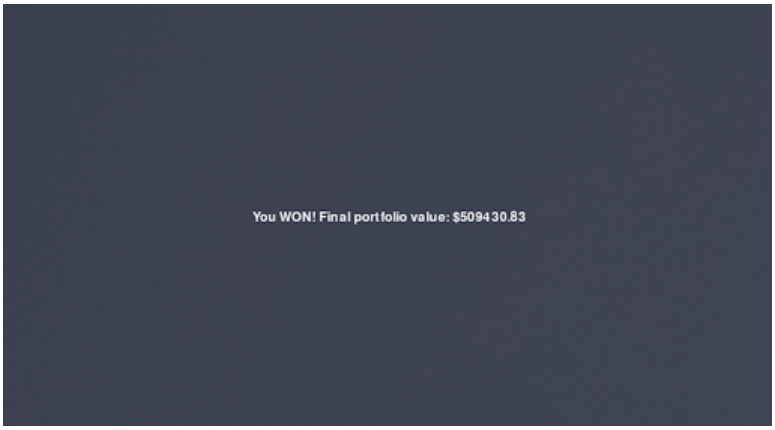

Figure 6.4. *The EndScene where the player has won*

7. Conclusion. Our Financial Trading Simulator Game represents a significant step forward in the realm of financial education. By integrating real-life economic principles into a hands-on interactive experience, we've provided players with an engaging and practical way to learn about financial markets and portfolio management. Through the game, players get insights into the correlations between various economic indicators and how these can impact the performance of different asset classes, all while trying to reach a portfolio target. That said, like any ambitious project, there are opportunities for improvement and future development.

Firstly, one area that could be enhanced is the generation of variables. While the ARMA models used in our game provide a solid foundation, incorporating more advanced or real-life

financial models could potentially make the game even more realistic. Models such as GARCH or VAR could be used to more accurately reflect the dynamic nature of financial markets.

Secondly, the portfolio weight adjustments could be made more intuitive and user-friendly. While keyboard inputs were used in the current iteration for simplicity, the introduction of input boxes or sliders could make this aspect of the game more visually engaging and have more precise control over the weights. Making short-selling a possibility would also be an interesting option to incorporate to have a more realistic experience.

Additionally, while the economic environment we have created within the game is focused on the Fed rate, inflation, bond prices, and stock prices, this could potentially be expanded. Our game environment could benefit from integrating additional variables to more accurately mimic real-world financial market conditions. Potential variables to consider might include macroeconomic indicators like GDP growth, unemployment rates, and consumer confidence indices. Other industry-specific metrics like company earnings and financial ratios could be factored into stock prices, while geopolitical events could introduce an additional layer of complexity and unpredictability into the game.

Other possible improvements could be incorporating news events which affect the markets, allowing for a multiplayer mode where players compete against each other, or introducing different difficulty levels and customizability options. Despite these potential enhancements, the simulator, as it currently stands, provides a robust framework for understanding the principles of investing. The game not only entertains but also educates, helping users to make better-informed financial decisions.

In conclusion, the Financial Trading Simulator Game is a comprehensive tool that introduces users to the world of financial markets. While there's always room for improvement and expansion, the existing structure provides a solid foundation for learning and engagement. As we continue to refine and expand the game, we look forward to providing an even more immersive and educational experience for our users.

8. Bibliography.

Sweigart, A. (2012). *Making Games with Python and Pygame*. Retrieved from https://spada.uns.ac.id/pluginfile.php/621901/mod_resource/content/1/makinggames.pdf

Markowitz, H. (1987). *Mean-Variance Analysis in Portfolio Choice and Capital Markets*. (pp. 1-43). Retrieved from https://books.google.ch/books?hl=fr&lr=&id=eJ8QUsgfZ8wC&oi=fnd&pg=PR9&dq=HM+Markowitz+portfolio&ots=t6xZOj7mhG&sig=-17zWoaFXUDnWUy8zAxkYdJ4w-U&redir_esc=y#v=onepage&q=HM%20Markowitz%20portfolio&f=false

McGugan, W. (2007). *Beginning Game Development with Python and Pygame: From Novice to Professional*. (pp. 1-34). Retrieved from https://books.google.ch/books?hl=fr&lr=&id=Kn8nCgAAQBAJ&oi=fnd&pg=PR14&dq=pygame&ots=eMYVnwDdhK&sig=_8yuhHFgEArdwY6xPfxpALwiHFM&redir_esc=y#v=onepage&q=pygame&f=false