

TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS

A PROJECT REPORT ON

**“RegEx2DFA Converter”**

Submitted by:

**Aman Dhaubanjar (079BCT010)**

**Ankit Pokhrel (079BCT016)**

**Ankit Prasad Kisi (079BCT017)**

SUBMITTED TO:

DEPARTMENT OF ELECTRONICS AND COMPUTER

ENGINEERING

IOE, Pulchowk Campus

Kathmandu Nepal

August 01, 2024

**Acknowledgement**

We would like to extend our sincere gratitude to **Mr. Daya Sagar Baral**, Lecturer, Department of Electronics and Computer Engineering, Pulchowk Campus, Institute of Engineering, for his guidance and support throughout this semester teaching us Object Oriented concepts in great detail.

In our project, we pay special thanks for the author and developer of C++, **Bjarne Stroustrup**, whose book laid the foundation for us to write effective code. Additionally, we would like to acknowledge the developers and maintainers of Graphviz, an open-source graph visualization software. Their contributions have been invaluable in providing a robust and flexible tool that has greatly facilitated the visualization of our DFA graphs.

Additionally, we are grateful to **Mr. Anuj Ghimire** for his invaluable insights on Automata, which have provided us with the knowledge to work in this field.

**ABSTRACT**

This report presents the development process, algorithm analysis, and outcomes of our “**RegEx2DFA Converter**” project. Automata, in simple terms, are abstract machines used to recognize patterns and process strings, such as regular expressions, according to specific rules. Our project aims to develop a program that aids students and educators in the field of theory of computation by automating the construction of Deterministic Finite Automata (DFA). This tool eliminates the need for manual DFA construction, often done through a trial-and-error approach, by leveraging the power of C++ and its object-oriented programming capabilities.

The project involved designing algorithms to address various challenges encountered during the development process, such as converting regular expressions into a computer-understandable format, parsing this format using our coded algorithms to generate a Non-Deterministic Finite Automaton (NFA), and subsequently converting the NFA into a DFA. The final output is a “.dot” file, which can be utilized by Graphviz library to visually represent the graphs.

Throughout the project, we applied principles from our Object-Oriented Programming course, gaining practical experience in software design, algorithms, and teamwork. We collaborated closely, utilizing each team member’s strengths to overcome challenges and make informed logical decisions. Our project supervisors, Mr. Daya Sagar Baral and Mr. Anuj Ghimire, provided valuable guidance and feedback, ensuring the project aligned with our learning objectives. As a result, we successfully designed and developed a program that transforms regular expressions into DFAs, effectively applying theoretical knowledge to practical implementation.

## **CONTENTS**

1. ACKNOWLEDGEMENT ……………………………………………………….1
2. ABSTRACT ……………………………………………………………………...2
3. OBJECTIVES ………………………………………………………………........4
4. INTRODUCTION ..……………………………………………………………...5
5. APPLICATIONS ……...…………………………………………………………6
6. LITERATURE SURVEY ………………………………………………………..7
7. EXISTING SYSTEM ..…………………………………………………………...8
8. METHODOLOGY ……………………………………………………………….9
9. IMPLEMENTATION …………………………………………………………...10
10. RESULTS ………………………………………………………………………..12
11. PROBLEMS FACED AND SOLUTIONS ………………………………………14
12. LIMITATIONS AND FUTURE ENHANCEMENTS …………………………..15
13. CONCLUSION AND RECOMMENDATION ………………………………….16
14. REFERENCES …………………………………………………………………..17

# **OBJECTIVES**

* **To Automate DFA Construction:**

Develop a tool that automatically converts regular expressions into Deterministic Finite Automata (DFA), eliminating the need for manual construction and reducing potential errors.

* **To Support Educational Use:**

Create a resource that can be used in educational settings to teach the concepts of regular expressions, NFAs, and DFAs.

* **To Develop a User-Friendly Interface:**

Implement an intuitive interface that allows users to input regular expressions, view the corresponding NFA and DFA, and visualize the results using Graphviz.

* **To Apply Object-Oriented Principles:**

Utilize C++'s object-oriented programming features to manage data structures and algorithms efficiently, providing a practical application of concepts learned in the Object-Oriented Programming course.

* **To Provide Accurate and Efficient Conversion:**

Ensure the program accurately and efficiently converts regular expressions to DFAs, handling edge cases and providing reliable results.

* **To Generate Visual Representations:**

Produce visual representations of the DFAs in DOT format, compatible with graph visualization tools like Graphviz, to aid in the analysis and understanding of the automata.

**INTRODUCTION**

Automata theory studies abstract machines, or automata, and their role in solving computational problems. It has roots in early 20th-century work by Alan Turing and Alonzo Church, who developed foundational concepts like the **Turing machine** and lambda calculus. These ideas led to the Church-Turing thesis, which asserts that any computational problem solvable by algorithms can be performed by a **Turing machine**. Automata theory, including the study of finite automata and regular expressions introduced by Stephen Kleene, is essential for understanding and designing computational systems and languages.

Before diving into automata, it's important to familiarize ourselves with some fundamental terms:

1. **Alphabet**: A finite set of symbols, usually represented as Σ, used to construct strings. For example, *Σ = {a, b}*.
2. **String**: A sequence of symbols from an alphabet. For example, *"abba"* is a string over the alphabet *{a, b}*.
3. **Language**: A set of strings formed from an alphabet. A language can be finite or infinite. For example, the language *{a, ab, abb}* consists of strings made from the alphabet *{a, b}*.
4. **Regular Expression**: A symbolic representation used to describe regular languages. Regular expressions can define patterns within strings, such as *"a\*b"* meaning zero or more *'a'*s followed by *a 'b'.*
5. **State**: A condition or situation in the computational process of an automaton, represented by a **circle** in diagrams.
6. **Transition**: The movement from one state to another in an automaton, triggered by input symbols. Transitions are depicted **as arrows** between states.
7. **Initial State**: The state where an automaton begins processing. It is usually represented by an **arrow pointing** to it without a source state.
8. **Accepting (or Final) State**: A state in which an automaton accepts a string as part of the language. It is represented by a **double circle**.

Now that we have a basic understanding of the terms used in Automata, let's focus on some operators we used in regular expressions and how they actually generate languages. In our project, we focused on three fundamental operators: union (+), concatenation (.), and Kleen closure. (\*). Here’s a concise explanation of each:

1. **Concatenation (.)**: This operator connects two expressions in sequence. For instance, if we have the expressions *"a"* and *"b",* the concatenation *"a.b"* results in the string *"ab"*. It signifies that the symbols or expressions should appear one after the other.
2. **Union (+)**: The union operator indicates a choice among multiple options. For example*, "a + b"* matches either *"a"* or *"b"*. It allows the creation of expressions where multiple possibilities are considered valid.
3. **Kleen Closure (\*)**: This operator represents zero or more repetitions of the preceding element. For instance, *"a\*"* can match *"", "a", "aa", "aaa",* and so on. It is used to specify that a particular pattern can repeat any number of times, including not appearing at all.

Let’s quickly look at some examples to be clearer in this:

* ***(a+b)\*****: { "", "a", "b", "aa", "ab", "ba", "bb", "aaa", "aab", "aba", "abb", ... }*
* ***(ab)\*****: { "", "ab", "abab", "ababab", ... }*
* ***ab(a+b)\*****: { "ab", "aba", "abb", "abaa", "abab", "abba", "abbb", ... }*
* ***a\*b****: { "b", "ab", "aab", "aaab", ... }*

**How String Process in Automata?**

In automata theory, an automaton processes a string by starting from an initial state (denoted by an **arrow** pointing to it). As it reads each symbol from the string, the **automaton transitions** to a corresponding state based on predefined rules. This sequence of transitions continues until all symbols in the string have been processed. If the automaton reaches a final state, denoted by a **double circle**, after processing the entire string, then the string is considered **accepted** by the automaton.

There are two main types of automata that can process strings in this way:

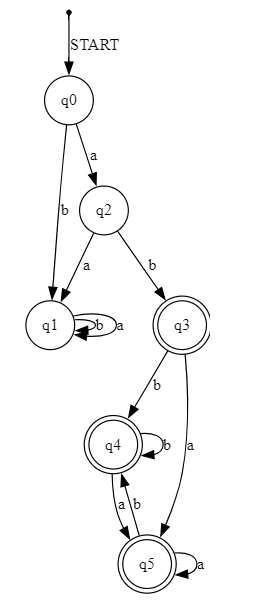
1. **Deterministic Finite Automaton (DFA)**: A DFA has **exactly one transition** for each symbol from every state, leading to a unique path through the automaton for any given input string.
2. **Non-deterministic Finite Automaton (NFA)**: An NFA can have **multiple transitions** for the same symbol from a single state, including transitions that don't consume any symbols **epsilon transitions(ε).** An NFA accepts a string if there is at least one path from the initial state to a final state that corresponds to the string.

For a string to be accepted, the automaton must end in one of its final states after processing all the symbols in the string. **This acceptance indicates that the string is part of the language recognized by the automaton**.

Here is a small example to get into automaton with regular expression:

**Q) Generate DFA with Regular expression: *ab(a+b)\*.***

***Solution:***

***ab(a+b)\*****: { "ab", "aba", "abb", "abaa", "abab", "abba", "abbb", ... }* 

***fig: DFA for ab(a+b)\* expression***

Let's break down the generated DFA, which consists of six states, with q0 as the starting state and q3, q4 and q5 as the ending states. The state transitions are represented by arrows. Let's process a few strings through this DFA:

**Processing "abba":**

* Starting from q0, the DFA transitions to q2 on reading 'a'.
* From q2, it moves to q3 on reading 'b'.
* With another 'b', it transitions to q4.
* Finally, on reading 'a', it reaches q5. Since q5 is an ending state, the string "abba" is **accepted**.

**Processing "baba":**

* Starting from q0, the DFA transitions to q1 on reading 'b'.
* From q1, it stays in q1 on reading 'a', then again on 'b', and once more on 'a'. However, since q1 is not an ending state, the string "baba" is **rejected**.

**APPLICATIONS**

Automata, particularly in language processing automation, has broad applications in various fields, including software engineering, data and text processing, cybersecurity and network analysis, control systems and robotics, bioinformatics, and digital systems design. Some of them are listed below:

 **Compiler Design**:

* Tokenization of source code into keywords, identifiers, and symbols.
* Syntax analysis and error detection.

 **Text Processing**:

* Searching and pattern matching in text editors and search engines.
* Data validation (e.g., email validation, password checking).

 **Natural Language Processing (NLP)**:

* Speech recognition and text-to-speech systems.
* Parsing and understanding the grammatical structure of sentences.

 **Finite State Machines in Robotics and Automation**:

* Control systems for managing different operational states.
* Behavior modeling in robotic systems.

 **Digital Circuit Design**:

* Design and analysis of digital circuits using finite state machines.
* State minimization and optimization.

 **Bioinformatics**:

* Sequence alignment and pattern recognition in DNA/RNA analysis.
* Identifying genes and functional motifs.

 **Speech and Voice Recognition**:

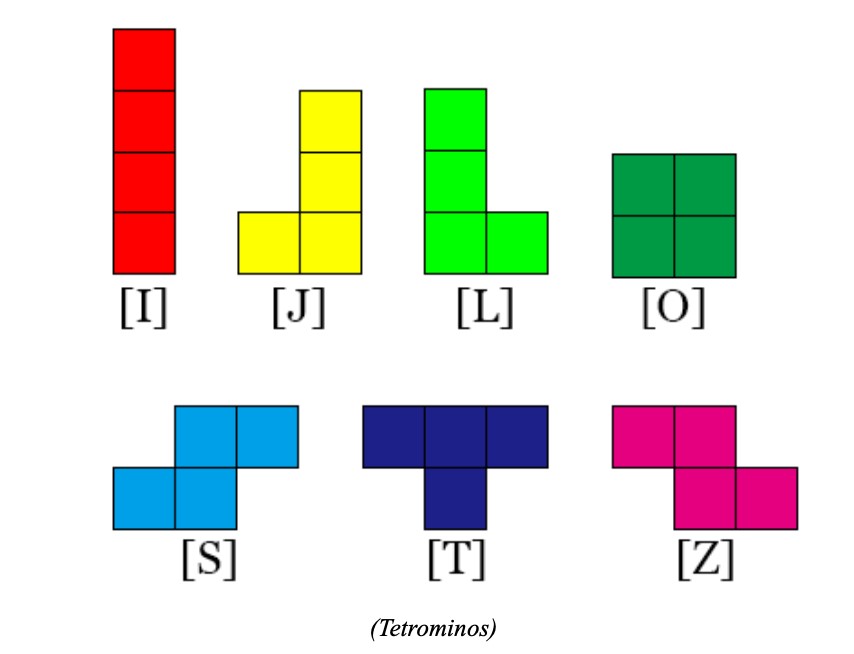
* Modeling phonetic and linguistic patterns.
* Accurate speech recognition and voice command systems.

# 6 LITERATURE SURVEY

It all began with a puzzle-loving software engineer named Alexey Pajitnov, who created ”Tetris” in 1984 while working for the Dorodnitsyn Computing Centre of the Soviet Academy of Sciences, a research and development center in Moscow created by the government. Pajitnov was inspired by a puzzle game called ”pentominoes,” in which different wooden shapes made of five equal squares are assembled in a box. Pajitnov imagined the shapes falling from above into a glass, with players controlling the shapes and guiding them into place. Pajitnov adapted the shapes to four squares each and programmed the game in his spare time, dubbing it ”Tetris.” The name combined the Latin word ”tetra” — the numerical prefix ”four,” for the four squares of each puzzle piece — and ”tennis,” Pajitnov’s favorite game. And when he shared the game with his co-workers, they started playing it — and kept playing it and playing it. These early players copied and shared ”Tetris” on floppy disks, and the game quickly spread across Moscow. When Pajitnov sent a copy to a colleague in Hungary, it ended up on display in a software exhibit at the Hungarian Institute of Technology, where it came to the attention of Robert Stein, owner of Andromeda Software Ltd., who was visiting the exhibit from the United Kingdom. ”Tetris” intrigued Stein. He tracked down Pajitnov in Moscow, but ultimately the game’s fate lay in the hands of a new Soviet agency, Elektronorgtechnica (Elorg), created to oversee foreign distribution of Soviet-made software. Elorg licensed the game to Stein, who then licensed it to distributors in the U.S. and the U.K. — Spectrum HoloByte and Mirrorsoft Ltd — The New York Times reported in 1988. According to the Times, ”Tetris” was the first software created in the Soviet Union to be sold in America. In Brown’s book, the unusual story of ”Tetris” is interwoven with an exploration of gaming: why people do it, how it changes them and how it brings people together. Pajitnov himself began this journey simply because he loved games and puzzles and wanted to share them with the world. And in the process, ”Tetris” took on a life of its own.

# 7 EXISTING SYSTEM

Tetris is a puzzle video game created by soviet software engineer Alexey Pajitnoovin 1984. It has been published by several companies for multiple platforms. The vanilla version of the game consisted of a board of dimension 10\*18. It consisted of seven blocks of different shapes called tetromino blocks which were dropped continuously from the top of the board in a random order. It is this version of the game that we are taking as reference. The player has to smartly place each block in the board so as to fill a line. Filling a line will clear that particular row and all the blocks above that row will drop on that row. Blocks above this row will drop on this row and so on. Basically, the player has to prevent the board from filling otherwise the game will end displaying the final score. The vertical motion of the dropping block is automated and updates on each game tick so players cannot control the motion of the block along the column but they can rotate blocks in all directions and move them left and right along a row as they fall from the top. Blocks can’t be moved once they stop dropping. Players can also see the preview of the next block in the side window. Today, Tetris can be found in a wide range of devices from PCs to mobile phones, with each adding new features to the already existing system.



# 8 METHODOLOGY

This project aimed to develop a Tetris game using Object-Oriented Programming principles and the SFML library for graphics. The project was implemented using the CLion Integrated Development Environment and utilized CMake for crossplatform build support. Version control and collaborative coding was facilitated through Git and GitHub.

For detailed history of development of this game, Link to our GitHub Repository: *https://github.com/Aashish079/Tetris*

To ensure a structured and maintainable codebase, we have designed the class hierarchy for the game. Different classes are created by inheriting from the parent class State, to represent game objects such as the GamePlayState class, GameOverState class, HighScoreState tetrominoes, and FileManager. We have employed OOP concepts such as encapsulation, inheritance, and polymorphism to achieve modularity and code reusability in this project. The game logic is implemented within the GamePlayState class. Each class have private and public member variables and member functions that handle specific aspects of the game, such as moving and rotating tetrominoes, checking for line clears, and updating the score. The SFML library is utilized for rendering graphics and handling user input.

To ensure efficient functioning, smooth transistioning and good development experience we have used starter code of SFML named TheStateMachine which stores and renders different states such as MainMenuState, GameplayState and GameOverState.

To facilitate collaboration among team members, we utilized Git and GitHub for version control. We had total of 7 different branches for different features. Regular commits were performed to the repository and timely pull request were initiated from different branches into main branch. GitHub’s issue tracking feature was also utilized to manage and prioritize tasks. Proper documentation was maintained to provide an overview of the project structure, class hierarchy, and function specifications..

We tried our best to take the most systematic and effiecient development approach throughout the development of this project.

# 9 IMPLEMENTATION

With the proposed objective and methodology, the pathway was implemented almost similar to what was proposed.

* Within the first week of undertaking the project, project was chosen and thus,subsequently a rough path was discussed and project proposal was made.
* After the project proposal, a rough sketch of the project schedule was created.
* Since three of our contributor were on three different OS(Windows, MacOSand Linux) we had to spend about a week or two to configure CMake for making our game cross-platform.
* Then we thoroughly studied and used the SFML Starter-Kit TheStateMachinefor managing different pages in the game in most efficient way possible.
* Then we focused on the Core Game-Logic
* Then we added Multimedia resources like Tetromino, Grid, photos for differ-ent buttons.
* We added FileManager Class which updated the Score and retrieved the scorefrom a file.
* Sound was Added to the Game.
* Final Code was tested and debugged
* Final Program was documented and a presentation was prepared.

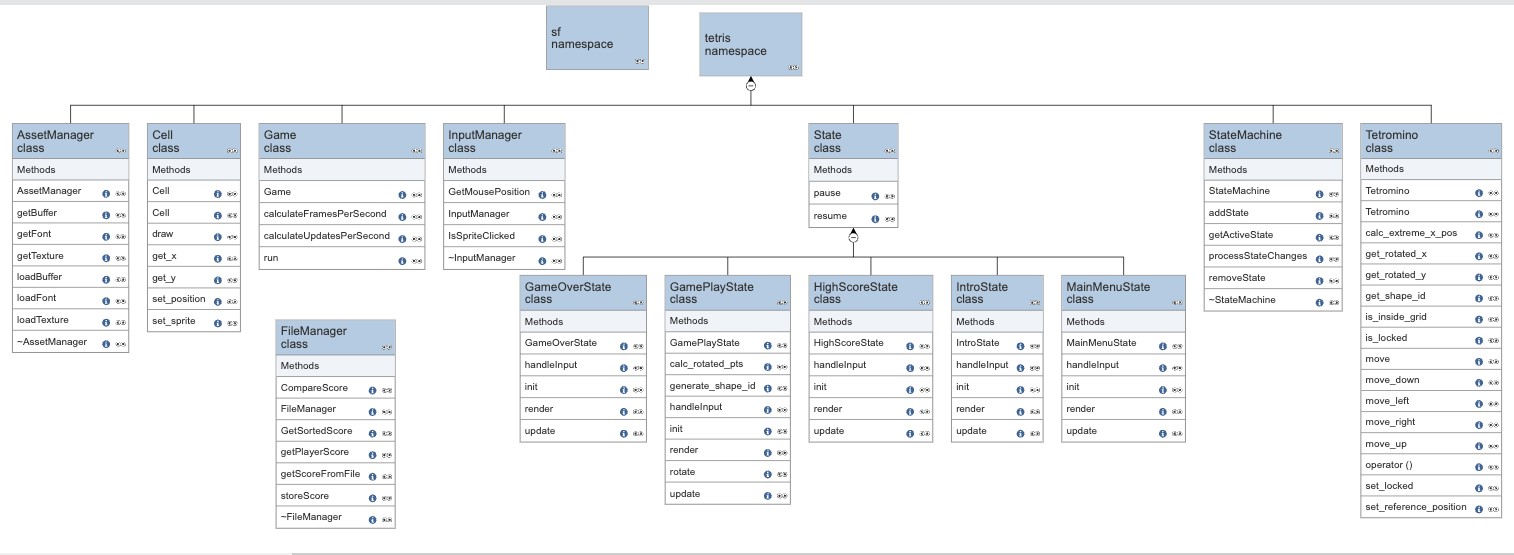


Figure 1: Class Diagram of the Game

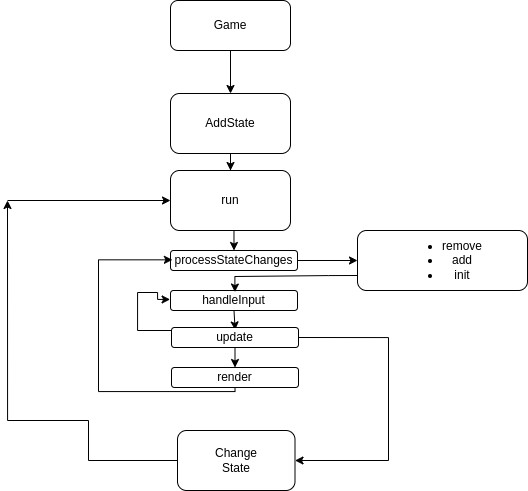


Figure 2: Block Diagram: State Machine

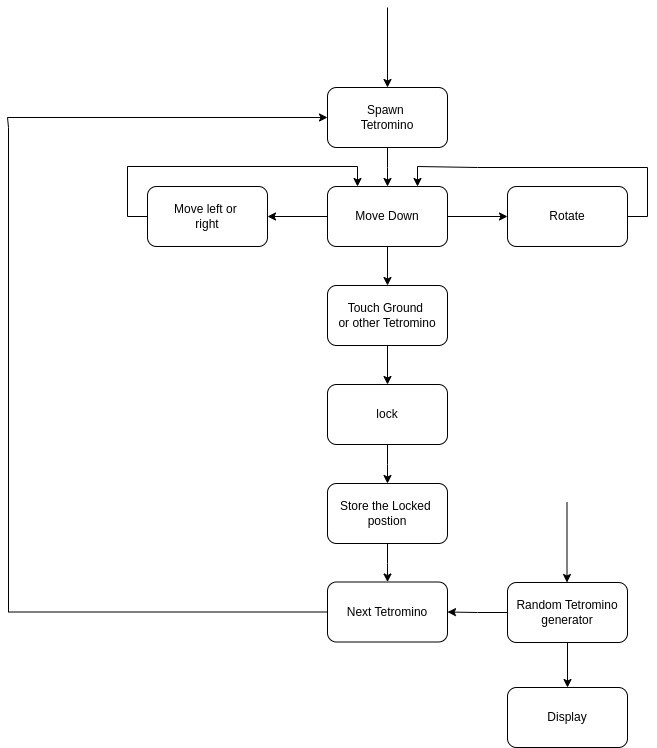


Figure 3: Block Diagram: GamePlay

# 10 RESULTS

After the final program was coded, the result obtained was close to what was expected. Game uses keyboard keys for function. Program starts with a short Intropage (for 2 sec) and then Mainmenu loads-up, with Play, HighScore, and Exit buttons. When played, the game goes into play state and game runs.

When the user gets out, GameOverState loads-up, and he is asked his name for storing his scores in a File. Then he can view top-ten scores in by pressing HighScore button and can press backspace to return back to the mainmenu state. Then pressing the Exit button will close the gameplay window.



Figure 4: IntroState



Figure 5: MainMenuState

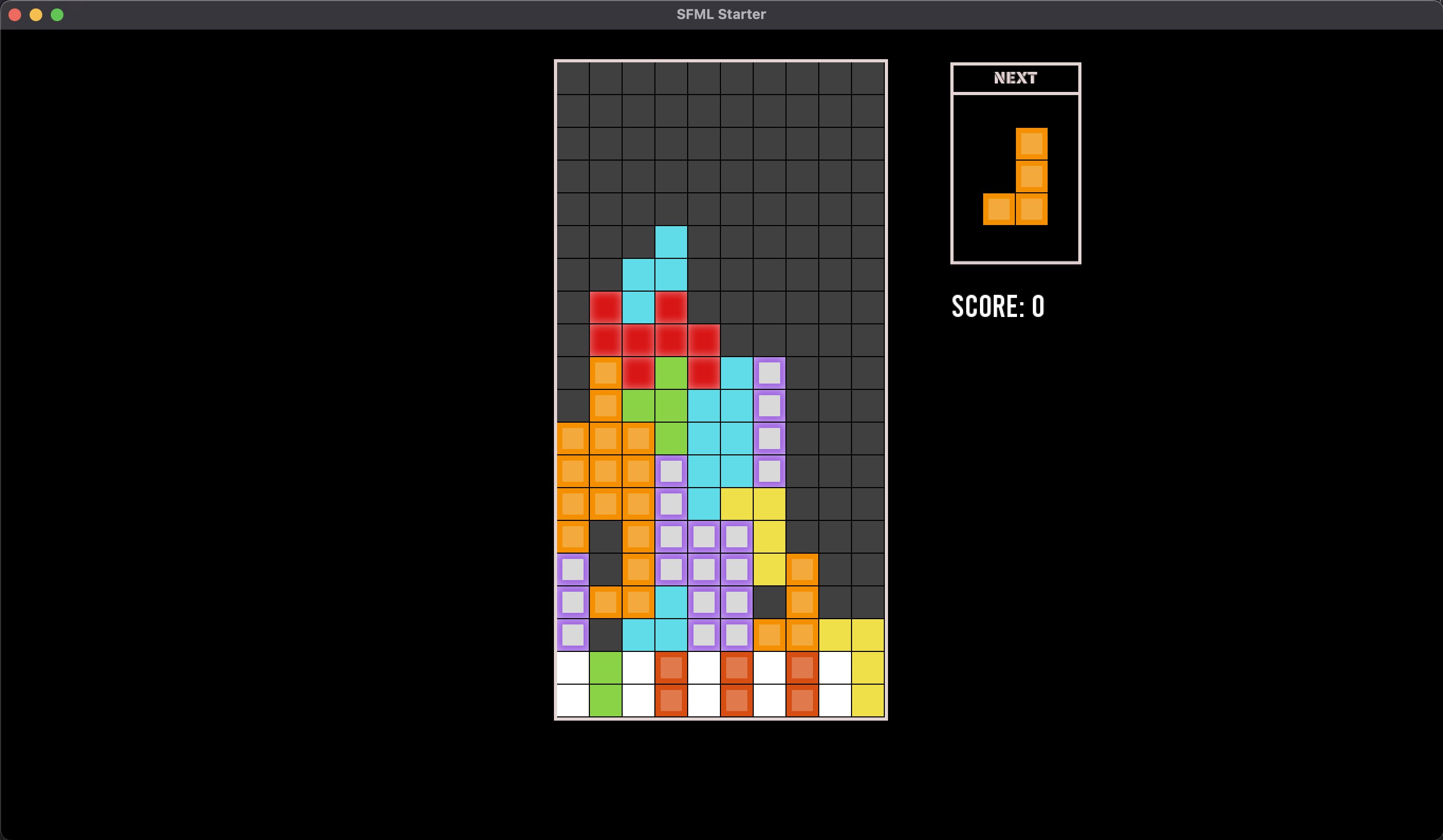


Figure 6: GamePlayState

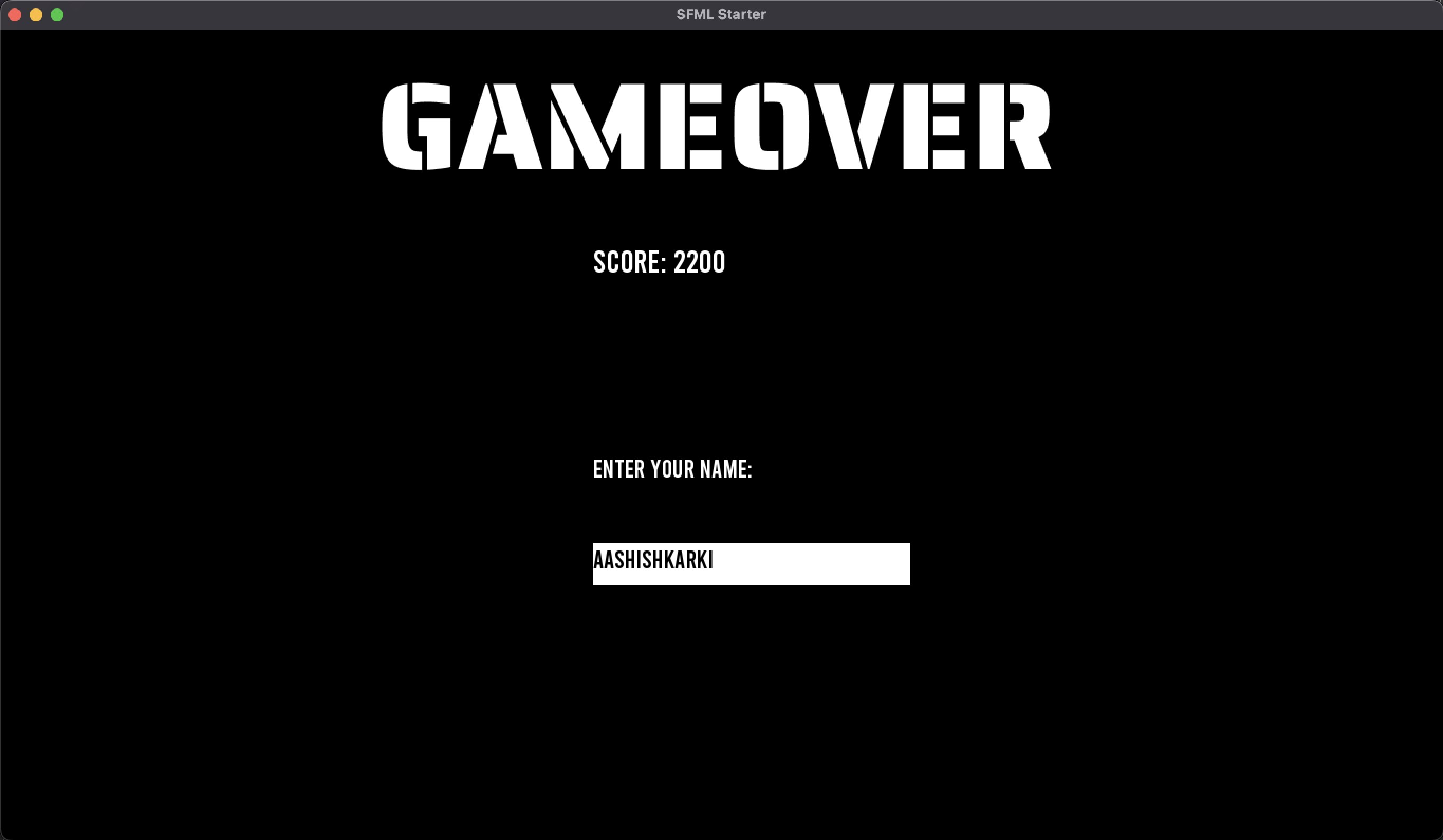


Figure 7: GameOverState



Figure 8: S

# 11 PROBLEMS FACED AND SOLUTIONS

1. Problem: IDE setup using CMake for cross-platform(Windows, Linux, MacOS)

Solution: We used Clion IDE

1. Problem: Managing Different Pages efficiently for better developer experience

Solution: We researched and found SFML Starter-Kit called TheStateMachine

1. Problem: Getting Familiar with SFML

Solution: We went through comprehensive SFML Tutorial and bunch of Tutorial Videos to get good grasp of SFML

1. Problem: Connecting Different Classes

Solution: We used Inheritance and Composition by creating user-defined headerfiles

1. Problem: Using STL for optimal Data and Memory Management Solution: We learnt STL from Course Book and referenced about it from the internet on the go.
2. Problem: Getting and Storing HighScore and User’s Name

Solution: We used Pair and Map data structures for storing and retriving Score and Name of Player.

## 12 LIMITATIONSANDFUTUREENHANCEMENTS

We have tried to include most of the features in our game but with limited time, we could not include everything. In our version of this game, we could have created multiplayer mode but this remains a limitation to our program primarily because we did not have the time to include it in our project. This feature will certainly be added in the next update to the game.

Also we had thought of adding special feature like drop of bomb after clearing multiple rows and could have made UI more responsive adaptable to different window sizes but we couldn’t add them due to lack of time.

Further, we believe that we could further organize our code and make it much cleaner and easier to read. We could have used the object-oriented approach more effectively.

In the future, we can further re-organize the code and make a much faster game. On top of that, we could allow multiplayer over the internet. This would be an exciting feature but we believe it’s outside the scope of the project.

# 13 CONCLUSION AND RECOMENDATION

We designed and built this project using the basic principles of Object Oriented Programming along with Simple and Fast Multimedia Library. The use of OOP easily organized our code into different modules and implement those modules at the required time.

The use of game class showcases this idea, where we have created a game class and just updated and rendered it in the main class. But as we go deeper into the implementation in game class, we see the actual commands. This abstraction is visible throughout the program and it made coding it much easier. Further, the use of a graphic library came in quite handy as well. The design of the UI of the game became very much easier and it saved us from coding thousands of lines of code. We also learned the need for planning, execution, and testing. The game logic is the same as the classical Tetris game, but the design and implementation are different.

# 14 REFERENCES

Websites :

1. https://www.sfml-dev.org/documentation/2.5.1
2. https://github.com/kiswa/SFMLStarter
3. https://geeksforgeeks.com
4. https://stackoverflow.com
5. https://youtube.com

Books:

1. The Secrets of Object Oriented Programming in C++ -Daya Sagar Baral, Diwakar Baral
2. Object-Oriented Programming in C++ - Robert Lafore (Fourth Edition)