Introduction

For our group project, we created an immersive, story-based quiz application titled *Fractured in Time*. The concept blends interactive trivia with an adventurous time-travel narrative. Players take on the role of a time traveler, hurtled across different historical eras with repairing a broken timeline by answering questions correctly.

The quiz features three levels, each tied to a specific time period, and presents a mix of multiple-choice and true/false questions. The player's goal is to complete the journey with as many correct answers as possible, earning one of three ranks based on their final score: **Time Lost** (0–60%), **Chrono Guide** (61–89%), or **Master of Time** (90–100%).

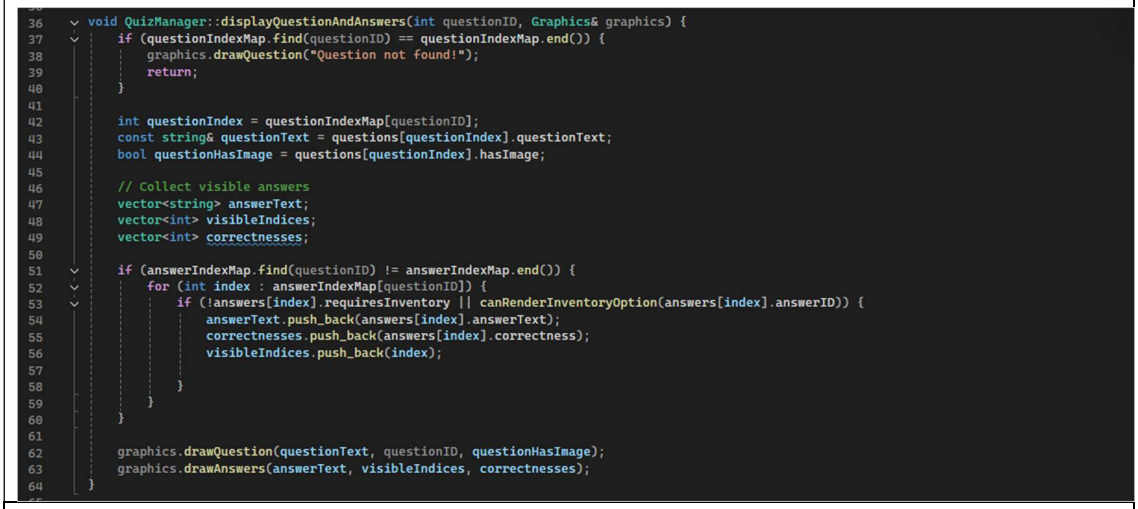
The application was developed collaboratively as a **console-based experience** and uses Microsoft Access database to store questions.

Our goal was to create a fun, replayable learning experience that rewards both knowledge and curiosity. Through teamwork, storytelling, and interactive design, *Fractured in Time* turns a simple quiz into a race through history—where every correct answer brings you one step closer to home.

Programming Techniques

1. Function

Screenshot:



```
void QuizManager::displayQuestionAndAnswers(int questionID, Graphics& graphics) {
    if (questionIndexMap.find(questionID) == questionIndexMap.end()) {
        graphics.drawQuestion("Question not found!");
        return;
    }

    int questionIndex = questionIndexMap[questionID];
    const string& questionText = questions[questionIndex].questionText;
    bool questionHasImage = questions[questionIndex].hasImage;

    // Collect visible answers
    vector<string> answerText;
    vector<int> visibleIndices;
    vector<int> correctnesses;

    if (answerIndexMap.find(questionID) != answerIndexMap.end()) {
        for (int index : answerIndexMap[questionID]) {
            if (!answers[index].requiresInventory || canRenderInventoryOption(answers[index].answerID)) {
                answerText.push_back(answers[index].answerText);
                correctnesses.push_back(answers[index].correctness);
                visibleIndices.push_back(index);

            }
        }
    }

    graphics.drawQuestion(questionText, questionID, questionHasImage);
    graphics.drawAnswers(answerText, visibleIndices, correctnesses);
}
```

Motivation:

The displayQuestionAndAnswers() function in QuizManager was created to organize and handle the complex task of showing both questions and their corresponding answers. Using a function here was necessary to avoid hardcoding the logic, which would have made the code repetitive, harder to manage, and less flexible. By putting this process into a single function, it becomes easy to manage changes to how questions or answers are displayed, ensuring consistency across the game. The function also properly filters visible answers, checking inventory requirements where needed, which keeps the gameplay experience dynamic and accurate.

How have you met the objectives?	Cross (X) the appropriate box	If you think that you have met the objective completely, provide a short explanation to support the claim
Not met		
Partially		
Completely	X	This approach meets the objectives completely by keeping the code clean, reusable, scalable, and easy to maintain, while securely handling all the necessary data.

2. Class

Screenshot:

```
9   class GameReport {
10  private:
11      std::vector<int> correctnessVector;
12      int totalQuestions;
13      int correctAnswers;
14      int score;
15
16  public:
17      GameReport(vector<int>& correctness);
18      void recordQuestion(int QID);
19      void recordAnswer(int index);
20      void printReport() const;
21  };
```

Motivation:

The GameReport class was created to manage all aspects of tracking and reporting player performance in an organized way. Using a class here was the best choice because it groups related data and functions together, making the code more modular, clean, and easy to work with. Instead of scattering performance logic across the program, everything is neatly handled inside GameReport, making it simple to record questions answered, track correct answers made, and print a final summary of performance. By keeping the data private and controlling access through methods, the class also ensures better data integrity and security.

How have you met the objectives?	Cross (X) the appropriate box	If you think that you have met the objective completely, provide a short explanation to support the claim
Not met		
Partially		
Completely	X	The class fully achieves its goals by neatly organizing player performance data, providing clear and focused methods for interaction, and maintaining encapsulation and modularity across the project.

3. Struct

Screenshot:

```
#pragma once
#include <string>

struct Answer {
    int answerID;
    int fromQuestionID;
    int toQuestionID;
    int correctness;
    std::string answerText;
    int inventoryAnswerID;
    bool requiresInventory;
    bool removedAfterPassed;
};

```

Motivation:

We decided to use a struct for the Answer object because it's a lightweight and efficient way to group related data together. It keeps everything organized in one place, making it easy to pass around between functions, store in collections, and work into the rest of our game logic. Since Answer is mainly just holding data and doesn't have much behaviour, a struct felt like the better fit over a class. This choice also helps keep our code cleaner, easier to read, and simpler to maintain in the project.

How have you met the objectives?	Cross (X) the appropriate box	If you think that you have met the objective completely, provide a short explanation to support the claim
Not met		
Partially		
Completely	X	Using a struct helps keep the quiz answer data organized and easy to work with, making it simpler to handle answers and run logic checks within our project.

4. Pointer

Screenshot:

```
126
127     string toString(SQLCHAR* buffer){
128         return std::string(reinterpret_cast<char*>(buffer));
129     }
```

Motivation:

We used a pointer (SQLCHAR* buffer) in this function to efficiently handle the data fetched from the database. Since ODBC returns data as a SQLCHAR*(a pointer to a character array), using reinterpret_cast lets us safely convert it to a char, making it easier to work with in C++ string

functions. This avoids unnecessary copies and keeps things fast, especially when handling large amounts of data. Converting it to a string also allows for easier manipulation of the data. This approach boosts both performance and flexibility.

How have you met the objectives?	Cross (X) the appropriate box	If you think that you have met the objective completely, provide a short explanation to support the claim
Not met		
Partially		
Completely	X	The function utilizes a pointer to handle and manipulate raw character data returned by the ODBC API. By using the pointer, the code avoids unnecessary copies of the data. This also ensures that the function can handle variable-length strings that may be returned from the database.

5. Reference

Screenshot:

```
419             //Check answer button
420             for (Button& b : buttons) {
421                 if (b.shape.getGlobalBounds().contains(mousePos)) {
422                     playClickSound();
423
424                     if (b.isCorrect)
425                         b.shape.setFillColor(sf::Color::Green);
426                     else
427                         b.shape.setFillColor(sf::Color::Red);
428
429                     renderCurrentScreen();
430                     sf::sleep(sf::milliseconds(250)); // half-second delay
431
432                     return b.answerIndex;
433                 }
434             }
```

Motivation:

We used a reference for (Button& b) inside the loop to work directly with the real button objects instead of creating copies. This allows the button to immediately change color (green for correct, red for incorrect) when clicked, giving players instant feedback. Using a reference also makes the code faster and more memory-efficient, helping the game stay responsive and smooth.

How have you met the objectives?	Cross (X) the appropriate box	If you think that you have met the objective completely, provide a short explanation to support the claim
Not met		
Partially		
Completely	X	We used references to directly manipulate the game objects without copying them. This enhances both the performance and the

		efficiency of the event handling system.

6. Struct

Screenshot:
[*Provide a screenshot of your code. This will be code of a vector in your game*]

Motivation:
[*Explain why you decided to specifically use a vector in this area of your code, and the functionality it provides to your code.*]

How have you met the objectives?	Cross (X) the appropriate box	If you think that you have met the objective completely, provide a short explanation to support the claim
Not met		
Partially		
Completely		

7. Data Structures

Screenshot:

```
10  void QuizManager:: buildIndexMaps() {
11      //A map where key: QID and value: Index
12      for (int i = 0; i < questions.size(); i++) {
13          questionIndexMap[questions[i].QID] = i;
14      }
15
16      //A map where key: fromQuestionID and value: Indicies of answers
17      // eg. 100: [0, 1]
18      for (int i = 0; i < answers.size(); i++) {
19          answerIndexMap[answers[i].fromQuestionID].push_back(i);
20      }
21  }
22
```

Motivation:
We used a map to quickly link question IDs to their corresponding questions and answers. Instead of searching through a full list every time, the map lets us instantly find the right question or set of answers based on an ID. This improves the game's performance, makes it more efficient, and keeps it easier to manage as it grows. Maps also keep the code more organized and readable compared to other methods like manually looping through vectors.

How have you met the objectives?	Cross (X) the appropriate box	If you think that you have met the objective completely, provide a short explanation to support the claim
Not met		
Partially		
Completely	X	By using maps, we ensure that accessing questions and answers is fast (O(1) time complexity), even as the game scales. This keeps the game's performance smooth while

		maintaining clean, organized code.

8. Class Template

Screenshot:
[*Provide a screenshot of your code. This will be code of a class template in your game*]

Motivation:
[*Explain why you decided to specifically use a class template in this area of your code, and the functionality it provides to your code.*]

How have you met the objectives?	Cross (X) the appropriate box	If you think that you have met the objective completely, provide a short explanation to support the claim
Not met		
Partially		
Completely		

9. Function Template

Screenshot:

[*Provide a screenshot of your code. This will be code of a function template in your game*]

Motivation:

[*Explain why you decided to specifically use a function template in this area of your code, and the functionality it provides to your code.*]

How have you met the objectives?	Cross (X) the appropriate box	If you think that you have met the objective completely, provide a short explanation to support the claim
Not met		
Partially		
Completely		

10. Operator Overloading

Screenshot:

```
31    void Player::operator+(int score) {
32        this->score += score;
33    }
```

Motivation:

In our game, players collect points as they answer questions correctly and make progress. We chose to overload the + operator so that adding points to the player's score is a clean operation. Instead of calling a normal function every time, using player + 5; makes the code shorter, easier to read, and keeps the flow of gameplay logic smooth.

How have you met the objectives?	Cross (X) the appropriate box	If you think that you have met the objective completely, provide a short explanation to support the claim
Not met		
Partially		
Completely	X	We successfully implemented the operator overloading for the + operator. It allows the player's score to be updated directly and improves the readability and simplicity of the game code.

11. Object Oriented Programming

How have you met the objectives?	Cross (X) the appropriate box	If you think that you have met the objective completely, provide evidence to support the claim
Not met		
Partially		
Completely	X	By using classes like QuizManager, Player, GameReport, and Graphics to organize different parts of the game. Each class has clear responsibilities, with private data and public methods to control access, showing good encapsulation. Inheritance, modularity, and reusability are applied to keep the code structured, easy to maintain, and scalable as the project grows.