Jose Arellano

5/13/25

Final project – C.A.T investigations game

Introduction

For my project i decided to make a mystery text based adventure game. In this game you work in an organization called C.A.T Investigations. There are 3 people in this organization, two girls and one boy. The two girls are called Claire and asper and the guy's name is tom. In my game they are hunting down a serial killer who thinks his motives are bringing justice to his wife's death. This game is very story driven and the player has to make a series of choices in each act. There are good and bad choices you can make throughout the game and that affects your moral alignment, which shall be shown at the end of the game. Programming this Project was pretty fun because it helped me learn more about making games, which is something i want to do in the future and it also shows me how challenging programming is sometimes. Overall, programming this project was both a challenging and rewarding experience. I enjoyed designing the story, implementing gameplay mechanics, and some algorithms I made I believe the game is entertaining not only because of its noir elements but also because it offers players the opportunity to immerse themselves in a game where you are a detective and you question your own choices and own morality while playing the game.

Algorithm 1 Combat Algorithm:

```cpp
void CombatSystem() {
    // Setting up player and enemy stats
    int playerHP = 100;
    int enemyHP = 100;
    int playerDamage = 15;
    int enemyDamage = 12;

    // Seed the random number generator
    srand(static_cast<unsigned int>(time(0)));

    cout << "\n--- Combat Begins! ---\n";
    cout << "You are facing Elias, the vengeful killer.\n";

    // Combat loop
    while (playerHP > 0 && enemyHP > 0) {
        // Player's turn
        int choice;
        cout << "\nYour HP: " << playerHP << " | Elias's HP: " << enemyHP << endl;
        cout << "1. Attack\n";
        cout << "2. Defend\n";
        cout << "Choose an action: ";
        cin >> choice;

        if (choice == 1) {
            // Player attacks
            cout << "You strike Elias for " << playerDamage << " damage!\n";
            enemyHP -= playerDamage;
        } else if (choice == 2) {
            // Player defends (reduces damage)
            cout << "You brace yourself, reducing incoming damage.\n";
        } else {
            cout << "Invalid choice. You waste your turn.\n";
        }

        // Check if the enemy is defeated
        if (enemyHP <= 0) {
            cout << "\nYou have defeated Elias!\n";
            break;
        }

        // Enemy's turn (random choice)
        int enemyAction = rand() % 2 + 1;
        if (enemyAction == 1) {
            // Enemy attacks
            if (choice == 2) {
                cout << "Elias attacks, but your defense reduces the damage!\n";
                playerHP -= (enemyDamage / 2);
            } else {
                cout << "Elias strikes you for " << enemyDamage << " damage!\n";
                playerHP -= enemyDamage;
            }
        } else {
            // Enemy defends (no action)
            cout << "Elias prepares himself, watching you carefully.\n";
        }

        // Check if the player is defeated
        if (playerHP <= 0) {
            cout << "\nElias has defeated you...\n";
            break;
        }
    }

    // End of combat
    if (playerHP > 0) {
        cout << "\nYou have survived the fight!\n";
    } else {
        cout << "\nYou have fallen...\n";
    }
}
```

Purpose: Allow the player to fight Elias in a turn-based style, making combat engaging.

1. Initialize Combat Stats:
    a. Set playerHP to 100 and enemyHP (Elias) to 100.
    b. Set playerDamage to 15 and enemyDamage to 12.
    c. Randomly seed the game for enemy actions (srand(time(0))).

2. Start Combat Loop (While Player or Elias is Alive):
    a. Display player and enemy HP.
    b. Prompt the player to choose an action:
        i. Option 1: Attack (deal playerDamage to Elias).
        ii. Option 2: Defend (reduce incoming damage by half).
        iii. Invalid Input: Player loses their turn.

3. Check if Elias is Defeated:
    a. If enemyHP <= 0, display a victory message and end combat.

4. Elias's Turn (Random Action):
    a. Randomly choose one of two actions:
        i. Attack: If player is defending, damage is halved.
        ii. Defend: Elias prepares, no action.

5. Check if Player is Defeated:
    a. If playerHP <= 0, display a defeat message.
6. Repeat Loop Until One Character Reaches 0 HP.
7. End Combat with a Victory or Defeat Message.

The overall time complexity of the algorithm is O(n) because where 'n' is the number of combat rounds. The longer the fight lasts, the more actions are processed, but each action is handled in constant time.

Purpose of the Combat Algorithm:

The reason why i made the combat algorithm in C.A.T Investigations is to create an engaging, interactive battle experience for the player. This system allows the player to actively participate in the conflict with Elias, the vengeful killer, rather than passively watching the story unfold. By giving the player direct control over their actions—whether to attack or defend—the combat algorithm adds tension, strategy, and a sense of agency to the game.

Why It Is Important:

The combat algorithm is crucial because it transforms the game from a simple text-based story into an interactive adventure. It forces players to make decisions under pressure, balancing risk and reward. Players must decide between attacking to defeat Elias quickly or defending to protect themselves from his unpredictable strikes. This mechanic is essential to maintaining player engagement, providing a sense of challenge, and making each encounter feel unique due to the random behavior of Elias. The combat system also directly connects to the story's themes of survival, morality, and justice, making it a core part of the player's experience.

Algorithm 2 Story display:

```
void readStoryFile(const string& filename) {
    ifstream storyFile(filename);
    string line;

    if (storyFile.is_open()) {
        while (getline(storyFile, line)) {
            if (line == "<<PAUSE>>") {
                cout << "\n-- Press Enter to continue --\n";
                cin.ignore(numeric_limits<streamsize>::max(), '\n');
                cin.get();
            } else {
                cout << line << endl;
            }
        }
        storyFile.close();
    } else {
        cout << "Error: Could not open story file: " << filename << endl;
    }
}
```

Purpose: To present the game's narrative by reading text from external files.

1. Initialize File Reading:

- The program attempts to open a text file (ifstream storyFile(filename)).
- If the file does not exist, an error message is displayed.

Time Complexity: O(1) (Constant Time) - File is only opened once.

2. Read the File Line by Line:

- The program uses a while loop to read each line of the file.
- Each line is checked for special commands ("<<PAUSE>>").
- If the line is a normal text line, it is displayed immediately.

Time Complexity: O(n) (Linear Time) - The loop iterates once for each line in the file, where n is the number of lines.

3. Handle Pause Commands:

- If a line contains the special marker "<<PAUSE>>", the program waits for player input (cin.get()).
- This gives the player control over the pacing of the story.

Time Complexity O(1) (Constant Time) - Detecting and processing the pause is instant.

4. End File Reading:

- When the end of the file is reached, the program stops reading and closes the file.
- If the file is missing, an error message is displayed.

Time Complexity: O(1) (Constant Time) - The file is closed instantly.

Purpose of the Story Display Algorithm:

The purpose of the story display algorithm in C.A.T Investigations is to present the game's narrative in a clean, readable format while maintaining player immersion. By reading and displaying text from external files, this algorithm ensures that the game's story is easily manageable and editable without altering the main code. It allows the player to progress through the story at their own pace, with key moments pausing for player input, enhancing engagement.

Why It Is Important:

The story display algorithm is vital because it transforms the game into an interactive storytelling experience. By separating the story text from the main code, it allows for easy updates and expansions to the narrative. This design also makes the game more modular—new story files can be added without changing the game's structure. Furthermore, the pause feature ("<<PAUSE>>") gives players control over the pacing, making emotional or tense moments

more impactful. This system is the backbone of the game's narrative, seamlessly guiding players through the unfolding mystery of Nebcity.

Algorithm 3 moral alignment:

```
int alignmentScore = 0;
```

```
alignmentScore -= 1;
```

```
alignmentScore += 1;
```

```
// Function to show the player's final alignment
//morality algorthm
//will show at the epilogue portion of the game
void ShowFinalAlignment() {
    cout << "\n=== Final Moral Alignment ===\n";
    if (alignmentScore > 0) {
        cout << "You are a Compassionate Detective.\n";
        cout << "You chose empathy and understanding over violence.\n";
    }
    else if (alignmentScore < 0) {
        cout << "You are a Ruthless Hunter.\n";
        cout << "You believed that justice requires strength and sacrifice.\n";
    }
    else {
        cout << "You are a Grey Investigator.\n";
        cout << "You walked the line between justice and vengeance.\n";
    }
}
```

Purpose: A system where each choice in your game adds or takes away a point based on the choice you made.

1. Initialize Alignment Score:

    The variable alignmentScore is set to 0 at the start of the game.

    int alignmentScore = 0;

Time Complexity:O(1) - Constant time. This is a single operation.

2. Player Makes Choices Throughout the Game:

2. Player Makes Choices Throughout the Game:

- Each choice the player makes can affect their morality.
  Compassionate Choice: Adds +1 to alignmentScore.
  Ruthless Choice: Subtracts -1 from alignmentScore.

Time Complexity: O(1) - Constant time per choice. Each choice is processed instantly.

3. Calculate Final Alignment at the End of the Game:

The final alignment is determined by evaluating the alignmentScore:

If the score is positive, the player is a "Compassionate Detective".
If the score is negative, the player is a "Ruthless Hunter".
If the score is zero, the player is a "Grey Investigator".

Time Complexity:

- O(1) - Constant time. This final check is always immediate, regardless of the number of choices made.

Complete Time Complexity:

- O(1) - Constant time.This is because each choice and the final calculation happen instantly, and the number of choices does not change the calculation speed.

Purpose:

The purpose of the morality system algorithm is to track and measure the player's ethical decisions throughout the game. By assigning positive or negative points to player choices, the system determines the player's final moral alignment: Compassionate Detective, Ruthless Hunter, or Grey Investigator. This adds depth to the narrative, making player choices feel meaningful.

Why is it important?

The morality system in C.A.T Investigations is crucial because it personalizes the story based on player decisions, making each playthrough unique. As players make choices throughout the game, their moral alignment is directly impacted, encouraging them to explore different ethical

paths. This system not only enhances replayability—prompting players to experience the story from different moral perspectives—but also provides direct feedback on the consequences of their actions. At the end of the game, the player is shown their final alignment, serving as a reflection of their decisions and reinforcing the weight of their choices.

Data structures

For data structures I used two, a vector, and strings. Some simple data structures but they got the job done

Vector:

In C.A.T Investigations, the vector data structure was used to store and manage a list of possible future cases that the player can preview after completing the main story. I chose Vectors because they are dynamic arrays, capable of automatically resizing as elements are added or removed. This flexibility is essential for the game, as it allows new cases to be easily added without altering the main code. The vector structure provides fast access to each case using an index, making it efficient for displaying the list of cases and allowing the player to choose one. This design not only keeps the code clean and organized but also ensures that the game can be easily expanded in the future with additional cases, maintaining scalability and modularity.

Strings:

In C.A.T Investigations, I used the string data structure extensively to handle text, making it a core part of the game's storytelling system. I used strings to store dialogue, display story text, and manage player choices. They were especially important in the readStoryFile() function, where the game reads and displays story text directly from external files. This design allowed me to manage, update, or expand the game's narrative easily without having to modify the main code. I also used strings to store the descriptions of future cases, making it simple to add or change case descriptions as needed. Strings were the perfect choice because they offered flexibility for text manipulation, ensuring that the game's story could be delivered smoothly and that player interactions felt natural.

Opportunity Encountered:

During the design of C.A.T Investigations, I realized an opportunity to make the game's narrative more dynamic and easier to manage. Initially, I planned to write all the story text directly in the code, but I recognized that this would make the game difficult to maintain or expand. Instead, I used an external text file system for the story, loading each chapter from a separate file using the ifstream (file stream) and string data structure. This allowed me to separate the code logic from the narrative, making the story easily editable without modifying the main code. By simply updating or adding text files, I could expand the game with new chapters or future cases. This design also gave me the flexibility to include a pause command ("<<PAUSE>>") that lets players control the pacing of the story, enhancing player engagement.

Another opportunity I encountered during the development of C.A.T Investigations was finding a way to effectively use the vector data structure. Initially, I knew I wanted to use vectors, but I wasn't sure where they would fit into the game. Then I had the idea of creating a post-game scene, where players could preview possible future cases for C.A.T Investigations. This not only gave the vector a clear purpose but also added a sense of continuity and replayability to the game. By using a vector to store a list of case descriptions, I made it easy to add, remove, or modify cases without changing the main code. This design also meant that I could quickly expand the game in future versions simply by adding new cases to the vector, making the game modular and scalable.

Errors encountered:

While developing the game, I encountered an issue where the story files (like **Act_one.txt** and **Epilogue.txt**) would not load properly. Whenever I ran the game, it displayed an error message

saying, "Error: Could not open story file." After troubleshooting, I discovered that the file paths were incorrect because the text files were not in the same directory as the main program. To resolve this, I ensured that all story files were placed in the correct directory and added error handling in the **readStoryFile()** function. The error handling displays a clear message if the file cannot be found, making it easier to identify and fix the problem. This approach not only solved the issue but also made the game more robust, as it could now detect and report missing files clearly.

Another error I encountered during the development of C.A.T Investigations was an issue with my chosen ID. I initially planned to use Visual Studio Code on my computer, but I ran into a problem where the IDE wouldn't work properly due to missing or corrupted JSON files. This prevented the compiler and debugger from functioning correctly, making it impossible to run or test my code. After several attempts to fix the issue, including reinstalling VSCode and trying different configurations, I decided to switch to GitHub Codespaces. By making this big switch I

was able to continue coding without any further IDE issues, allowing me to focus entirely on designing and improving the game.

Overall/what i would make better:

Overall, I am proud of how **C.A.T Investigations** turned out. The game successfully combines interactive storytelling, decision-making, and combat mechanics to create an engaging player experience.

However, there are several improvements I would make in the next version. First, I would expand the combat system to include more player actions, such as special attacks, healing items, or a dodge mechanic. This would make combat more strategic. Second, I would enhance the morality system by adding hidden moral choices that influence the story without being immediately obvious. This would make the game's narrative feel deeper. Third, I would add a save/load system, allowing players to continue their progress or explore different choices without restarting. Finally, I would expand the future cases system, making it possible for players to choose a case and play it directly, turning the game into a full detective adventure with multiple cases. These improvements would make the game more dynamic, engaging, and replayable.

Overall i hope you enjoyed the project, and I will definitely improve upon it more.