CSE232 - Project 3

April 2025

Project Overview

In this project, you'll take on the role of a game AI developer by designing a simple brain that controls a player in a 2D terminal-based game. Your task is to join an ongoing project and implement one of its core components: A somewhat intelligent agent that can play the game autonomously.

The game provides features that allow you to play and learn its simple mechanics through terminal graphics. Think of it as programming your own little game bot and watching it come to life in the terminal.

This is the most important project of the course, but it's also designed to be fun and flexible. You can experiment with different strategies, basic rules, greedy methods, search algorithms, memory implementations, or even your own creative twists. The main requirement is to write your code in a clean, object-oriented style, within a class called Brain (in brain.cpp) that integrates into a larger game. While you don't need to fully understand the entire codebase, it's important to pick up small details from it to see how the brain can be created and connected.

The project is worth **130 points**. Submit it through D2L. Late submissions are accepted up to 2 days after the deadline with a 25% penalty per day.

Make sure to read the full project instructions before you begin. This project is designed to give you both creative freedom and hands-on experience working within a realistic codebase. **However, keep in mind that this is not a project you can complete in a single day, even if you're a skilled programmer.**

Given the high stakes, start early, understand the assignment, plan your approach, implement your ideas incrementally, and test your code thoroughly. In this project, there is also an opportunity to recover up to 10% of your lost project score if you put in the extra effort. If you felt stuck, ask questions on Piazza, join helprooms and interact with your TAs!

Assignment Deliverable

The deliverables for this assignment are the following file(s):

brain.cpp – brain implementation file

brain.h – brain header file

DO NOT submit any other files. Your submission file names must exactly match the specified names. Any slight difference (e.g., extra characters in the file names, uppercase letters, submitting additional or different files, or using compressed formats such as `.zip` or `.rar`, etc.) will result in your

project not being graded. You should not modify the contents of any files other than `brain.cpp` or `brain.h`, and you must not add any additional implementation or header files to the project. You may create other classes or external functions within the same `brain.cpp` or `brain.h` files if you deem it necessary.

Assignment Specifications

Let's start by running the game on your local computer. Make sure you have a recent version of a C++ compiler installed and that your system is configured to run C++20 code.

1. Open the "Project03" folder with your IDE (e.g., Visual Studio Code). Make sure you are in the correct working directory in the terminal. You can compile the project files using `g++` directly with the following command:

   ```
   g++ -std=c++20 -Wall main.cpp Game/game.cpp Game/player.cpp GameAI/brain.cpp Game/enemy.cpp
   ```

   (If you prefer to use `make` for easier testing/compilation, instructions on how to use the Makefile are provided below.)

2. Assuming "a.out" is the name of your executable, you can run the code in multiple modes. Let's begin by playing the game as a human player using the keyboard, with full map awareness! Run the executable with the `-testhuman` option:

   ```
   ./a.out -testhuman
   ```

   You should see:

   ```
   Stage: 0 Score: 0 Moves: 0
   1        2       3       4        5        6
   ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
   + v +    +    0  +++++ + +        +X X X X +      T     +
   +   +  + + 00 00 +           B    +            T  T   T +
   +   +  + + 00 00 +   +++++        +        +   T      T w
   +   +  + + 00  00+       ++++++   +        +   T  T   T +
   +   +  + + 0   0 +++++++ ++++++   D        +   T  T   T +
   +      + +       +     + + A+++   +        +      T     +
   +   ++++ +0      +   + + +        +        +   T  T   T +
   +   ++++      0  D   +   +        + X X X X+   T      T w
   ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
   ```

   You are playing as a character represented by "v" in the top-left corner of the map. Your goal is to progress through six maze-like stages to reach one of the two victory points, represented by "w", located on the right side of the map.

   "+" signs are walls (or possibly semi-short fences because you can see over them). Use the "W", "A", "S", "D" keys to move around! You cannot move through walls. You can also press "N" to skip a game cycle without moving, and "X" to manually exit the game.

   Stage 1 is a maze. Find your way out to score some points! Passing each stage grants incremental points.

In Stage 2, you're required to gather all the food represented by "0" for the door, marked "D", to open and allow progression to the next stage.

Stage 3 is a spiral maze that sometimes requires you to move in the opposite direction.

Stage 4 is a flag-capture mission: go to point "A" to pick up the flag, then proceed to point "B" to place it and open the stage door "D".

In Stage 5, you must dodge moving enemies to avoid being re-spawned at the stage entrance. To succeed, stay calm, learn the enemy patterns, plan ahead, use "N" to skip turns if needed, and find your way out.

Stage 6 is a minefield or rather, a trap field. Avoid all traps represented by "T" and reach one of the victory points ("w") on the right side of the map.

Score is consisted of passing stages, eating food, picking/placing flags and remaining number of moves in case of winning (winning with the less number of moves results in higher scores).

3. Try finishing the game to familiarize yourself with the mechanics. Note that you are only allowed a maximum of 1000 moves.

4. Now that you have a better idea of how the game works, let's make the computer play on its own! The initial `brain.cpp` contains a random AI that has an equal chance of moving in any direction or doing nothing. Let's see how it performs in the game. Try the following command:

   ```
   ./a.out -testvisual
   ```

   You can exit the program by pressing `Ctrl + C` (or `Command + C` on Mac).

5. So far, you've tested the game in two modes: once as a human player, and once with a random AI. In both cases, you had full visibility of the map. However, this is not how the game is normally played. Try the following command:

   ```
   ./a.out -human
   ```

   In this mode, you play with limited character vision. Your character can only see a short distance ahead and has restricted awareness of its sides and what's behind it. Try winning the game in this mode—it's more challenging, but definitely possible!

6. Finally, let's see a random AI in action under limited vision with the following command:

   ```
   ./a.out -visual
   ```

   Again, you can exit the program the same way. You can also run the program with no options to quickly test your developed brain with no visuals.

Your task for Project 3 is to replace the random logic in the files named "brain.cpp/h" with logic that can pass the first four stages of the game in a single run. If your logic passes stages 5 and 6, you will receive extra credit!

Attached to the project specification, you'll find the files and the code base related to the game. Below are additional details about the structure of the program, implementation hints, and other important notes to help you succeed in this project. Make sure to continue reading. Take a break if you need to, but be sure to read the entire PDF before you begin!

Project Structure

The following is the structure of the project files and their relevance within the project:

```
Project03/
    Game/
        enemy.cpp       // Enemy blueprint
        enemy.h
        game.cpp        // Game core
        game.h
        player.cpp      // Player blueprint
        player.h
    GameAI/
        brain.cpp       // Player AI brain
        brain.h
    Maps/
        L1.map          // Main game map
        L2.map          // Other map
        L3.map          // Other map
    main.cpp            // Main project file
    Makefile            // Project make file
    manual_interface.h  // Utility functions for taking keyboard input
```

While you technically don't need to read through the game code in detail, it's important to understand how the project is structured so that you can see how your own code fits in. There are three directories and three files in the project root:

- **Game/** contains three implementation files: `enemy.cpp`, `player.cpp`, and `game.cpp`.
  - **enemy.cpp/h:** Defines the blueprint of an enemy entity in the game, including a function that describes how it moves.
  - **player.cpp/h:** Defines the blueprint of a player entity, including functions for setting/getting the player's position and direction, and for re-spawning the player.
  - **game.cpp/h:** Implements the entire game logic, including map loading, progression of each game cycle, and the win condition. It also handles moving the player based on the input received each cycle.
- **GameAI/** contains `brain.cpp/h`, which define the blueprint for the player's AI brain. **These are the two files you should modify to implement your brain logic and submit for grading.**
- **Maps/** contains map files. Once you understand the game logic, you're welcome to create new maps to challenge yourself or to test specific stages. Note that only `L1.map` is a shared map used for grading, but the game should work with any well-formatted map file.
- **main.cpp** is the main entry point of the program and contains the game loop.
- **manual_interface.h** provides helper functions to handle keyboard input for human players.
- **Makefile** is the project's make file. You can either compile manually using `g++`, or use `make` (install it if it's not already available on your system). If you're unfamiliar with makefiles, a quick web search will help!

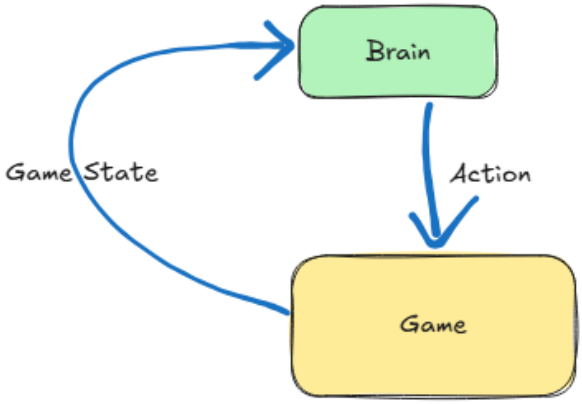

Figure 1: Game's main loop.

Much like many turn-based games, this one revolves around a main loop that advances each game cycle based on player actions. If there's a human player, inputs are read from the keyboard and translated into actions. The game then processes the resulting changes, including moving entities as needed. A human player relies on visual feedback from the terminal to understand the game state.

When using an AI brain, the game must communicate with it by sending information about the current state and receiving an action in return (Figure 1).

This interaction is handled by creating an instance of the `Brain` class (declared in `brain.h`) and invoking the following function:

```
int Brain::getNextMove(GameState &gamestate);
```

This function takes a `GameState` struct as input, which contains the following attributes:

```
struct GameState
{
    int stage{0};                // Current stage based on player position
    int score{0};                // Current score
    int cycle{0};                // Current game cycle
    vector<vector<char>> vision; // 2D vector representing the current vision of
                                 // the player based on facing direction
                                 // May vary in size each cycle
    int pos[2];                  // Player's current position (h, w)
};
```

Among these, `vision` is the most critical variable for decision-making. When the player is not near a map edge, this is typically a 5x7 or 7x5 matrix (Figure 2). If the player is near the edge, the vision matrix is smaller due to limited visibility. A challenge in the project is that the vision cone has different sizes depending on the player's direction and position. It may even return empty rows at times. This is intentional. Make sure to account for possible inconsistencies when working with player vision to avoid runtime issues.

The return value of `getNextMove()` must be an integer from 0 to 4, mapping to the following actions:

```
0 // Do nothing
1 // Go Up    (same as pressing W as a human player)
2 // Go Left  (same as pressing A as a human player)
```

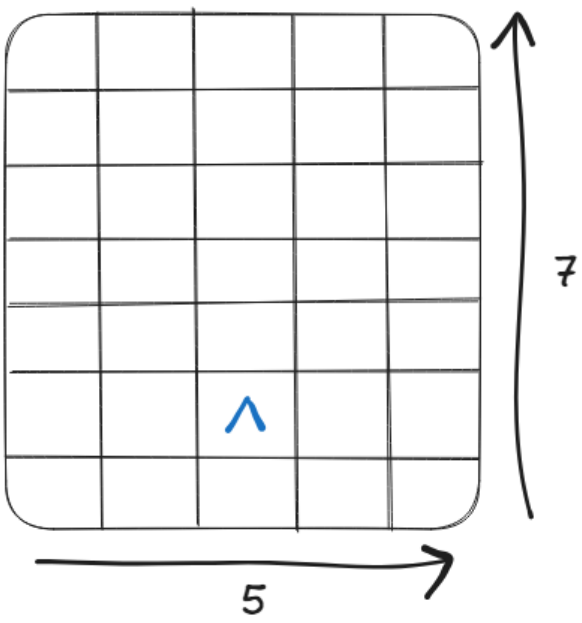

Figure 2: Vision cone of a player facing up.

```
    3 // Go Down  (same as pressing S as a human player)
    4 // Go Right (same as pressing D as a human player)
```

Below is the base definition of the `getNextMove()` function, which you are expected to replace:

```
int Brain::getNextMove(GameState &gamestate)
{
    // AI logic to determine the next move based on the gamestate

    // Generate a random number between 0 and 4
    int nextMove = std::rand() % 5;
    return nextMove;
}
```

Note that this logic completely ignores the game state and simply generates a random move. This is our placeholder "random AI brain." Your task is to use the information available in each cycle to make smarter decisions and beat the game!

Important: If your function returns a value outside the range `0` to `4`, the game will terminate with a runtime error.

Grading Considrations

- **For grading, the same game map provided in `L1.map` and a slightly modified version will be used. Any hardcoded brain logic that relies on exact paths in `L1.map` is likely to break** (e.g., creating a brain that always moves forward 5 times and then turns right because it works in the original map). This restriction does not apply to small hardcoded patterns that are genuinely helpful in decision-making.

- You should not make assumptions about the order of the stages. However, you will not be tested on any maps that require techniques beyond those needed for `L1.map`, nor on maps with entirely new mechanics.
- Your code logic will be tested three times, and the highest score among those runs will determine your grade.
- At this point in the semester, you are expected to write code that compiles successfully. Your grade will be primarily based on successful compilation and the score your brain logic achieves. Passing four stages across both test cases will earn you the full 130 project points. If your code logic passes all six stages, you may receive up to an additional 10% in project weight as bonus points. Further details will be shared later.
- Your `brain.cpp` file must include logic that you have written yourself. While you're free to explore and implement various algorithms to reach your goals, you are not allowed to use non-standard C++ libraries or any AI libraries to create, train, or use black-box AI models. Simply put, you need to process the game state yourself and make decisions based on that.
- Your code won't be graded on time efficiency, but it should run within a reasonable time frame (complete within 1000 game cycles and under 5 seconds).

Tips for Testing Your Code

- Make your changes directly in the `brain.cpp/h` files. As with any other C++ program, compile/debug your code first and test it using the instructions provided above. Make sure to practice incremental development.
- To quickly see the score your `brain.cpp` achieves, run the program without any command-line options. Generally, passing four stages results in a score above 100. Running the program without options also allows you to print custom debugging messages.
- To observe how your brain performs in a slower, visual mode, use the `-visual` option.
- While entirely optional, it's a good idea to open the `.map` files in the `Maps/` directory to understand how the stages are structured or even to design your own. Just keep in mind that you'll need a solid understanding of how game entities behave and how maps are formatted to make meaningful changes. Make small changes first.
- Use the `-map` option to test your brain logic with a custom map. Make sure each level is surrounded by walls to prevent the player or enemies from going out of bounds. Also, ensure that map files contain no extra characters. Use `Ctrl/Command + A` in your text editor or IDE to select all content and remove trailing whitespace. Any extra whitespace might break loading the map during runtime.

Implementation Hints and Starting Points

Here are some ideas and hints that you might find helpful. You have a lot of freedom in this project to create your own solution, and there are many different ways to approach it. You're welcome to use any of the following suggestions or take your own path entirely. However, it may not make sense to implement all the ideas below at once, as some might conflict or be less effective for certain stages in game.

- Depending on your approach, you may need to write a lot of code. Break it down by defining separate functions for specific tasks. Test each function or new piece of logic before moving on. This will make debugging easier and help with building more advanced behavior.
- This is a game that progresses from left to right. The victory point(s) are always located on the right side of the map.
- You can use either random or deterministic strategies. A guided random approach is perfectly acceptable. For instance, move randomly, but not aimlessly: avoid doing nothing or repeatedly going backward unless it makes sense.
- A straightforward and effective approach is to "look around" and take action based on what you see. If you don't like what's ahead, turning to face a different direction might be a smart move.
- Use all the information available to your advantage. For example, you can detect when the stage changes! This allows you to design specific strategies for different stages after identifying their objectives.
- You can implement memory to track your previous moves and avoid unnecessary repetition. You can also use flags to remember certain events. Advanced approaches might even reconstruct the map internally based on the player's vision over time.
- Your code may implement "persistent" memory. Since your logic is tested three times, you're allowed to create a temporary `.txt` file to store what you learned in earlier runs and use that knowledge in later attempts. Of course, when submitting, you should still only include `brain.cpp/h` and not any extra files generated at runtime.
- Try wall-hugging! Or maybe you might be able to adapt techniques you learned from the Spiral Lab problem and apply them here.

Receiving Help, AI Usage & Collaboration

- Before anything else, rely on yourself and try to implement the algorithms you have in mind on your own. Make sure to do thorough debugging to identify where problems may be occurring.
- There's nothing wrong with seeking help. Join help rooms, make Piazza posts (if you want to share code, use the insert code button and make it a private post), schedule meetings with us, and talk to your TAs during lab sessions. Believe me, interacting with the staff is not only easy and encouraged, but also the most effective way to get constructive feedback and help. If your TAs don't know the answer to your question, ask them to forward it to the instructors.
- You are allowed to use generative AI tools for this project, as long as you properly attribute their use. However, do not copy and paste blocks of AI-generated code directly into your project. Instead, use these tools for ideas and inspiration, and then type the code yourself! Typing helps reinforce your understanding and builds coding speed by teaching useful shortcuts and patterns. Also, take time to practice debugging and avoid over-relying on AI tools for that purpose. This project is a great opportunity to get familiar with various types of compilation errors. You don't need to submit your AI history for this project.
- You are free to collaborate with friends and classmates to brainstorm ideas for tackling various tasks in the game. However, sharing code or compiling lists of ideas publicly or on Piazza is not

permitted. All coding and implementation must be your own individual work. In a project of this size, it's highly unlikely for two people to independently arrive at identical ideas and structural similarities.

- You may ask your TAs for their opinions on specific implementation details, they'll be happy to help!
- Last but not least, use this project as a chance to grow as a programmer and to develop strong coding habits! If you embrace this as your own adventure, not only will you enjoy the project more, but you'll also learn a lot in the process.

Using Make (Optional)

`Make` files are sets of instructions written in a specific format that help automate the compilation of multiple files using a single command. If you prefer, you can use the provided `Makefile` in the project to compile and run your code more easily. First, ensure that `make` is installed on your system. You can check this by running the following command in the terminal:

```
make -v
```

If the command is not recognized, search online for instructions on how to install `make` for your operating system, and install it. Once installed, confirm it by running the command above again.

After installing `make`, navigate to the project root directory in your terminal and use the following commands:

```
make            // Uses the Makefile to compile your project and create
                // an executable named run.out

make run        // Compiles the project and runs the executable without any flags

make visual     // Compiles and runs the brain AI player in visual mode

make human      // Compiles and runs the game in human mode

make testvisual // Compiles and runs the game with the -testvisual option

make testhuman  // Compiles and runs the game with the -testhuman option
```

You may modify the `Makefile` to adjust configuration parameters if needed. However, you should **not** submit the `Makefile` as part of your deliverables. Only submit the two files: `brain.cpp` and `brain.h`.

Disclaimer

The game engine is over 1000 lines of code and was developed in a short time. While it generally follows good coding conventions, you may notice some deviations from best practices. In the unlikely event that you noticed any sort of bugs, let us know immediately.