Objectives

The objectives of this project are to:

1. to demonstrate and implement all major concepts of object oriented programming
2. to make a Database Management system for file operations and persistent memory
3. to make a terminal graphics library and use it to develop a snake game

Introduction

This project is an implementation of the classic Snake game using C++ and incorporates several essential software engineering and object-oriented programming (OOP) concepts. It consists of multiple components, including the game engine, graphics library, database management, and a user interface implemented in the main file.

Graphics Library Documentation

The Graphics Library is a lightweight efficient graphics library for terminal. It provides a range of essential features for manipulating colors, pixels, grids, and complex shapes (glyphs), this library simplifies the process of creating interactive and visually appealing ascii graphics.

Code Organization

The library is organized into a C++ namespace named “Graphics.” It defines several classes and structs within this namespace to encapsulate related functionality. Header files (`header.hxx`) declare class and function prototypes, while source files provide implementations.

Color Management

The `Color` class is responsible for managing color attributes and text formatting. It stores foreground and background colors as `rgb` structs, with options to set color intensity. The class also supports text formatting options, such as bold and italic. The `put()` method generates ANSI escape codes to apply these attributes, while `reset()` resets all color and formatting attributes.

Example of setting a color and generating escape codes:

```
Color foregroundColor({255, 0, 0}, {0, 0, 0});
// the color class can also take some special flags like blinking/bold text/ita
lic text etc in the next argument, whicha re defined as static
std::string colorCode = foregroundColor.put();
std::string resetCode = Color::reset();
```

Pixel Management

The `Pixel` class represents an individual character cell with associated color attributes. It allows the setting of both the character cell content and color. It can be displayed using the `<<` operator, which

applies color formatting.

Example of creating and displaying a pixel:

```
Color pixelColor({255, 0, 0}, {0, 0, 0});
Pixel pixel("A", &pixelColor);
std::cout << pixel; // Displays a red letter "A"
```

Grid Management

The `Grid` class is a 2D grid of pixels and is used to create a virtual console screen. It provides methods to access and manipulate individual pixels within the grid. The `display()` method outputs the grid to the console, while `clean()` clears all pixels. the static function `clear()` clears the terminal

Example of creating a grid and displaying it:

```
Grid screen(80, 24); // Create an 80×24 grid
screen.display();    // Display the empty grid
```

Gylphs

The `Gylph` class is used for creating complex graphical elements composed of multiple pixels. It takes arrays of character cells and colors to define the appearance of the element. The `plot()` method is intended for adding the Gylph to a Grid.

Example of creating a Gylph and plotting it on a Grid:

```
std::vector<char[4]> gylphChars = {{". .", " . ", ". ."}};
std::vector<Color> gylphColors = {{Color({255, 0, 0}, {0, 0, 0})}};
Boundary gylphBoundary = {{0, 0}, 3, 3};
Gylph g(gylphChars, gylphColors, gylphBoundary);
g.plot(screen); // Add the Gylph to the Grid
```

Database Library Documentation

The Database Management Library is a C++ library designed to facilitate the creation and management of a simple database system. It provides classes and functions for defining database schemas, creating tables, inserting, retrieving, and updating data within tables, and handling array data structures. This database was intended to be made lightweight and easy to use

Library Overview

The library consists of the following key components:

- **Database**: Represents a database and manages tables within it.
- **Table**: Represents a table within a database and provides methods for interacting with rows.

- **Schema**: Defines the structure of a table, including fields and their data types.
- **Row**: Represents a row in a table and allows for data retrieval and manipulation.
- **ArraySchema**: Defines the structure of an array field within a table.
- **ArrayCell**: Represents an element within an array field.
- **Cell**: Represents a basic data cell within a table.

Features

The Database Management Library offers the following features:

- Creation and management of databases and tables.
- Definition of database schemas, including fields and data types.
- Storing and retrieving data from tables.
- Support for array data structures within tables.
- Basic error handling and data validation.

Working explanation

The database data are stored in binary format. As binary data is not formatted in any order, a schema is used for each table to read and write data. The schema ensures portability of the stored data.

The schema contains field names and data types, while writing data to a datafile, the order remains the same as the schema's order. In case of strings, the first 4 Bytes contain the string length. Reading is done in the same order as the schema.

All data are loaded in a void pointer, which can be casted to any type according to schema

Documentation

Here are some code snippets and usage examples to show how to use the library:

Creating a Database

```
#include "header.hxx"
using namespace DB;
Database db("folder"); // Create a new database or load an existing one
```

Creating a Table

```
Schema schema;
schema.addField("name", Schema::text);
schema.addField("age", Schema::integer);
db.add("table", &schema); // Create a table named "my_table" with the defined s
chema
```

Adding Rows and Retrieving Data

```
//Retriving data
```

```
DB::Row* aa = db["scorecard"]["easy"];
int* rt = (int*)((*aa)[f].get());

//Writing data
DB::Row& row = db["scorecard"].row("easy");
row[f].set(DB::Schema::integer, (void*)&game.score);
row.save();
```

Handling Array Fields

```
Row &row = Calendar["Events"].row("210101");
  row("events").push_back(ArrayCell());
  row("events")[0]["name"].set(Schema::text, vstr("1esty"));
  row("events")[0]["description"].set(Schema::text, vstr("qqq"));
  row.save(); //saved to disk

//Retriving an entry
Row *aa = Calendar["Events"]["210101"];
std::cout << (char *)((*aa)("events")[0]["name"].get()) << std::endl;
```

Snake Game Engine Documentation

The Snake Game Engine is implemented as a C++ library, encapsulating key game-related functionalities and structures. It employs various Object-Oriented Programming (OOP) concepts to achieve modularity and maintainability.

Game Class

The `Game` class serves as the central component of the engine. It encapsulates the entire game state and provides methods for game initialization, updates, and rendering. Key attributes and methods of the `Game` class include:

- **Attributes**:
  - `Snake snake`: An instance of the `Snake` class representing the game's snake.
  - `Food* current_food`: A pointer to the current food instance, either `RegularFood` or `SpecialFood`.
  - `Graphics::Grid grid`: An instance of the `Grid` class for rendering the game grid.
  - `int* mp`: Pointer to an integer array for tracking the game grid state.
  - `int level, rows, columns`: Game level, number of rows, and columns.
  - `DIRECTIONS curr`: Current direction of the snake.
  - `int score`: Player's current score.
  - `bool hasfood`: Indicates whether there's food on the grid.
- **Methods**:

- `Game(int columns, int rows, int level)`: Constructor for initializing the game.
- `bool move(DIRECTIONS direction, bool ate)`: Moves the snake in the specified direction. and also checks for self collision and game end
- `int &operator()(int i, int j)`: Overloaded operator for accessing grid cells.
- `void plot()`: Updates the grid to reflect the snake's current position.
- `void serve()`: Places food on the grid.
- `void display()`: Displays the game grid.
- `bool collides()`: Checks if the snake collides with food.

Snake Class

The `Snake` class represents the game's snake. It is a friend class of `Game`, allowing it to access and manipulate the game's internal state. It consists of a `std::list` of `Point` objects, where each `Point` represents a segment of the snake.

Snake Movement Mechanism

The snakes body was defined as a list of blocks. The list data structure was chosen efficiency. it does not requires large movement of data for appending data at front or end, and removal of last element is also O(1)

Food Classes

The `Food` class is an abstract base class that defines the common interface for regular and special food. It includes the `give_points()` method, which returns the number of points awarded when the snake consumes the food.

- `RegularFood` and `SpecialFood` classes derive from `Food` and implement the `give_points()` method to provide different point values.

Engine Operation

1. **Initialization**: The `Game` class is instantiated with the desired grid dimensions and level. It initializes the game grid, snake, and sets the current food type.

2. **Game Loop**: The game enters a continuous loop where it waits for player input and updates the game state accordingly.

3. **Player Input**: Player input is captured via the `inputLoop()` function, which sets the direction for the snake (up, down, left, right) based on the user's keypress.

4. **Game Logic**: The `gameLoop()` function handles the core game logic. It updates the snake's position, checks for collisions, manages food, and calculates the player's score.

5. **Rendering**: The `plot()` function updates the grid to reflect the snake's current position, and the `serve()` function places food on the grid.

6. **Collision Handling**: Collisions are checked using `collides()`, and if the snake collides with food, the player's score is updated.

7. **Game Over**: If the snake collides with itself or the grid boundaries, the game ends, and the player's score is saved to the database.

The main file

Threading

Opening Options for Players

```
std::cout << "\nWelcome to Snakeland\n\n" << std::endl;
std::cout << "1. Continue game\n2. New Game\n3. Show Scores\n\nOption: " << st
d::endl;
int q;
while (std::cin >> q) {
    // Game initialization and option handling go here
}
```

In the main file, two threads are created to handle concurrent gameplay:

void gameLoop()

```
void gameLoop();
```

The `gameLoop` function is responsible for managing the game's core logic. It runs in its own thread and handles tasks such as collision detection, player input processing, updating the game state, and managing game over conditions. This threading ensures that the game logic can run independently and efficiently.

`void inputLoop()`

```
void inputLoop();
```

The `inputLoop` function also runs in its own thread and is responsible for capturing player input during the game. It continuously monitors the keyboard for user inputs. When a player presses keys like arrow keys ('w', 'a', 's', 'd'), the `inputLoop` function captures and processes these inputs, allowing for real-time control of the snake's direction.
By utilizing threads, the game can handle gameplay and input simultaneously, providing a responsive and enjoyable gaming experience.

Score Saving

Storing Highest Score

```
int max_easy = 0;
```

```
DB::Database db("Snake");
char* f = "score";
scorecard.setIndex("scorecard").addField(f, DB::Schema::integer);
db.add("scorecard", &scorecard);
```

The `max_easy` variable is used to store the highest score achieved in the “easy” mode of the game. The game initializes the database connection, allowing it to retrieve and update the player’s highest score.

Saving Score on Game Over

```
if (!sst && game.score > 4) {
// >4 is required for gamestart self collision prevention
    death = true;
    DB::Row& row = db["scorecard"].row("easy");
    row[f].set(DB::Schema::integer, (void*)&game.score);
    row.save();
    std::cout << "Game Over" << std::endl;
    exit(0);
}
```

The main file provides players with options when starting the game. The player’s choice is captured using `std::cin`, and based on the selected option, appropriate actions are taken, such as game initialization.

Special Templated `getch` Function

```
template <class T>
T getch(void) {
    // ...
}
```

The `getch` function is a templated function that allows the program to capture a single character of user input without echoing it to the console. It uses system calls and terminal settings to achieve this functionality. This function is used to capture player input in real-time during the game, allowing the snake’s movement to respond to the player’s keypresses. it is templated to return either int or char value.

Discussion

Used OOP Concepts

Inheritance:

Inheritance is a fundamental object-oriented programming (OOP) concept, and it’s used in various parts of the codebase.

Use Case:

In the `Engine` namespace, there is an example of inheritance in the `RegularFood` and `SpecialFood` classes. These classes inherit from the base class `Food`. This allows for code reuse and a common interface for different types of food objects.

Snippet:

```
class Food {
 protected:
  Boundary boundary;

 public:
  virtual int give_points() = 0;
};

class RegularFood : public Food {
 public:
  int give_points();
};

class SpecialFood : public Food {
  int give_points();
};
```

Abstract Class:

An abstract class is a class that cannot be instantiated and is meant to be subclassed. In the codebase, the `Food` class serves as an abstract class.

Use Case:

The `Food` class declares a pure virtual function `give_points()`, making it an abstract class. Subclasses like `RegularFood` and `SpecialFood` must provide an implementation for this function.

Snippet:

```
class Food {
 public:
  virtual int give_points() = 0;
};
```

Polymorphism:

Polymorphism allows objects of different classes to be treated as objects of a common base class.

Polymorphism is used extensively in the game engine.

Use Case:

Polymorphism is demonstrated when different food objects, such as `RegularFood` and `SpecialFood`, are treated as instances of the base class `Food`. This enables a unified way of interacting with different food types.

Snippet:

```
Food* current_food; // Polymorphic usage of food objects
current_food = new RegularFood;
```

Operator Overloading:

```
class Row {
  Schema* scheme;
  std::fstream file;
  std::string location;
  std::string index;
  std::map<std::string, Cell> cells;
  std::map<std::string, std::vector<ArrayCell>> array_cells;
  bool loaded = false;

 public:
  friend class Table;
  Row(Schema* scheme, std::string location);
  ~Row();
  Cell& operator[](std::string fieldname);
  std::vector<ArrayCell>& operator()(std::string fieldname);
  bool load();
  bool del_row();
  bool del_arr_index();
  Cell setField();
  bool save();
};
```

Files:

The codebase involves file operations, such as reading and writing game data to files.

Use Case:

In the `main` file, the code reads and writes game scores and state information to files using file streams, demonstrating file handling.

Snippet (File Reading):

```
DB::Row* aa = db["scorecard"]["easy"];
int* rt = (int*)((*aa)[f].get());
```

Snippet (File Writing):

```
row[f].set(DB::Schema::integer, (void*)&game.score);
row.save();
```

Encapsulation:

Encapsulation is the concept of bundling data (attributes) and methods (functions) that operate on that data into a single unit, known as a class. It helps in data hiding and abstraction.

Use Case:

Encapsulation is used throughout the codebase. For instance, in the `Engine` namespace, the `Snake` class encapsulates the data and logic related to the snake's movement and fragments.

Snippet (Encapsulation in Snake Class):

```
class Snake {
  friend class Game;

 public:
  std::list<Point> fragments;
};
```

Template

```
template <class T>
T getch(void) {
  struct termios oldattr, newattr;
  int ch;
  tcgetattr(STDIN_FILENO, &oldattr);
  newattr = oldattr;
  newattr.c_lflag &= ~(ICANON | ECHO);
  tcsetattr(STDIN_FILENO, TCSANOW, &newattr);
  ch = getchar();
  tcsetattr(STDIN_FILENO, TCSANOW, &oldattr);
  return ch;
} /* reads from keypress, echoes */
```

STL (Standard Template Library):

The Standard Template Library (STL) is a C++ library that provides various template classes and functions. It's used implicitly in many parts of the codebase, especially with containers like `std::list` and `std::thread`, `stl::map`, `stl::vector`

Composition and Aggregation:

Composition and aggregation are design concepts where a class can contain or be associated with other objects. In the codebase, the `Game` class is composed of other objects, such as `Snake`, `Graphics::Grid`, and `Food`. It also demonstrates aggregation when creating instances of `RegularFood` and `SpecialFood` inside the `Game` class.

Use Case (Composition):

```
class Game {
  // ...
  Snake snake;
  RegularFood rf;
  SpecialFood sf;
  // ...
};
```

Use Case (Aggregation):

```
class Game {
  // ...
  Food *current_food;
  // ...
};
```

Static Classes:

Static classes are classes that cannot be instantiated, and their methods can be called directly on the class itself. While not explicitly used in the codebase, the code uses static methods from user-made libraries like `Graphics::clear()`.

Function Overloading:

```
Grid::Grid(int columns, int rows) : rows(rows), columns(columns) {
  graph = new Pixel*[rows * columns + 1111];
  for (int i = 0; i < rows * columns + 1110; i++) {
    graph[i] = (Pixel*)blank_pixel;
  }
  _set = true;
}
```

```
Grid::Grid() {}
void Grid::set(int c, int r) {
  rows = r;
  columns = c;
  try {
    if (_set) throw "[Grid] second time init\n";
    graph = new Pixel*[rows * columns + 10];
    for (int i = 0; i < rows * columns; i++) {
      graph[i] = (Pixel*)blank_pixel;
    }
  } catch (std::string err) {
    std::cerr << err;
    exit(-1);
  }
}
```

Overriding

```
class Food {
 protected:
  Boundary boundary;

 public:
  virtual int give_points() = 0;
};
class RegularFood : public Food {
 public:
  int give_points();
};
```

Conclusion

In conclusion, this project demonstrates the successful application of Object-Oriented Programming (OOP) concepts to create a classic Snake game.

The game engine, implemented in the `Engine` namespace, illustrates the power of OOP by incorporating inheritance, abstract classes, and polymorphism. It showcases encapsulation with well-defined class structures, operator overloading for easier data access, and templates for generic functionality. The Standard Template Library (STL) is utilized for essential components like lists and threads, while composition and aggregation are employed to build the game environment.
The `Graphics` namespace manages graphical elements, providing a user-friendly interface for rendering the game grid. In contrast, the `DB` namespace handles data storage and retrieval for high scores, demonstrating file I/O and structured data management.

This project represents a scaled-down version of the original objective, which was to develop an open-

world RPG game utilizing cellular automata. However, due to the imposed submission deadline, only the database and graphics modules were completed at the time. There was some uncertainty regarding how to integrate separately generated chunks, and as the project submission deadline approached, it was decided to simplify the game concept to ensure completion within the allocated timeframe.