1. File Structure

```
🗃️ CapybaraChronicles/
├── 📁 include/
│   ├── game_state.hpp
│   ├── save_system.hpp
│   ├── run_game.hpp
│   ├── ascii_art.hpp
│   ├── 📂 Levels/
│   │   ├── lvl1.hpp
│   │   ├── ...
│   │   ├── lvl10.hpp
│   └── Minigames/
│       ├── spin_the_wheel.hpp
│       ├── memory_match.hpp
│       ├── ...
├── 📁 src/
│   ├── main.cpp
│   ├── save_system.cpp
│   ├── run_game.cpp
│   ├── ascii_art.cpp
│   ├── 📂 Levels/
│   │   ├── lvl1.cpp
│   │   ├── ...
│   │   ├── lvl10.cpp
│   └── 📂 Minigames/
│       ├── spin_the_wheel.cpp
│       ├── memory_match.cpp
│       ├── ...
├── 📁 saves/
```

2. Key Files

`main.cpp`

- The **starting** program.
- Enumerates or creates a `saves/` directory, loads existing saves or starts new.
- Instantiates a `GameState` and calls `runGame(state)`.

`run_game.cpp`

- Manages the **main loop** of the game, calling each `lvlN.cpp` in numerical order.
- Applies the **end-of-level** logic, e.g., subtracting 10 eatingResources or shortfall from health.

`save_system.(hpp/cpp)`

- Provides `saveGame(...)` and `loadGame(...)` functions that read/write the `GameState` struct to a text file.

`game_state.hpp`

- Defines the **GameState** struct with:
  - `levelNumber`
  - `health`
  - `eatingResources`
  - `otherResources`
  - `sessionName`

`ascii_art.(hpp/cpp)`

- Holds **ASCII frames**, printing cover pages, animations, etc.

2.1 `game_state.hpp`

Working:

- Defines the `GameState` struct, which holds all essential game variables (health, resources, etc.).
- Acts and minigames modify the `GameState` to reflect the player's progress.

```
#ifndef GAME_STATE_HPP
#define GAME_STATE_HPP

#include <string>

// Tracks the player's essential data
struct GameState {

    int levelNumber        = 1;    // Which level or act we're on
    int health             = 100;  // The player's health
    int eatingResources   = 10;   // Food/water resources
    int otherResources    = 0;    // Additional resource category
    std::string sessionName = "default"; // Session name to identify the
save file

};

#endif
```

Explanation:

- sessionName: Identifies the save file.
- levelNumber: Tracks the player's progress through levels 1 to 10.
- health: Represents the player's vitality; reaching 0 results in game over.
- eatingResources: Represents sustenance; required to maintain health.
- otherResources: Represents materials or other items used within the game.

2.2 `run_game.hpp` / `run_game.cpp`

Working:

- `runGame` is called from `main` and loops over levels (lvl1, lvl2, …, lvl10).
- At each level's end, it may deduct resources or shortfall from health, check if health <= 0, etc. If `levelNumber` exceeds 10, the game ends.

Header: `include/run_game.hpp`

```
#ifndef RUN_GAME_HPP
#define RUN_GAME_HPP

#include "game_state.hpp"

// The main loop that calls each level in order
void runGame(GameState &state);

#endif
```

Source: `src/run_game.cpp`

```
#include "run_game.hpp"

// Example includes for levels (all 10)
#include "Levels/lvl1.hpp"
#include "Levels/lvl2.hpp"
// ...
#include "Levels/lvl10.hpp"

#include "save_system.hpp" // if we unify saves here
```

```
#include "ascii_art.hpp" // for graphics (GAME OVER)
#include <iostream>
using namespace std;

void runGame(GameState &state)
{
    bool running = true;

    while (running)
    {
        switch (state.levelNumber)
        {
        case 1:  lvl1(state);  break;
        case 2:  lvl2(state);  break;
        // ...
        case 10: lvl10(state); break;
        default:
            running = false;
            break;
        }

        // Check if the player has completed all levels
        if (state.levelNumber > 10) {
            running = false;
            break;
        }

        if (running)
        {
            const int toDeduct = 10; // Amount to deduct each level

            if (state.eatingResources >= toDeduct)
            {
                state.eatingResources -= toDeduct;
                cout << "\n(End-of-level: -10 eatingResources)\n";
            }
            else
            {
                int shortfall = toDeduct - state.eatingResources;
                state.eatingResources = 0;
                state.health -= shortfall;
                cout << "\n(End-of-level: Not enough eating resources. "
                     << shortfall << " health points deducted)\n";
            }

            // Check if health has dropped to zero or below
            if (state.health <= 0)
            {
                clearScreen();
                cout << "\nYour health has dropped to 0!\n";
                GameOverScreen();
                running = false;
            }
        }
```

```
            // If still running, save the session
            if (running)
            {
                cout << "\n\nSaving progress:";
                string filename = "saves/" + state.sessionName + ".txt";
                bool ok = saveGame(state, filename);
                if (ok)
                    cout << "\n[Saved to " << filename << "]\n";
                else
                    cout << "\n[Save failed]\n";
                delay(2);
            }
        }
    }
```

Explanation:

- Game Loop: Continuously runs while `running` is `true`.
- Level Invocation: Uses a `switch` statement to call the appropriate level function based on `levelNumber`.
- Resource Deduction: At the end of each level, deducts 10 `eatingResources`. If insufficient, the shortfall is subtracted from `health`.
- Health Check: If `health` drops to 0 or below, the game ends with a game over screen.
- Saving Progress: Automatically saves the game after each level if the game is still running.

2.3 `save_system.hpp` and `save_system.cpp`

Purpose: Handles the saving and loading of the game state to and from files, enabling players to persist and resume their progress.

Header: `include/save_system.hpp`

```
#ifndef SAVE_SYSTEM_HPP
#define SAVE_SYSTEM_HPP

#include <string>
#include "game_state.hpp"

// Saves the current game state to a file
bool saveGame(const GameState &state, const std::string &filename);

// Loads the game state from a file
bool loadGame(GameState &state, const std::string &filename);

#endif
```

Source: `src/save_system.cpp`

```cpp
// src/save_system.cpp
#include "save_system.hpp"
#include <fstream>
#include <iostream>
using namespace std;

// Function to save the game state to a file
bool saveGame(const GameState &state, const string &filename)
{
    ofstream outFile(filename);
    if (!outFile)
    {
        cerr << "Error: Could not open file for saving.\n";
        return false;
    }
    // Write game state variables line by line
    outFile << state.sessionName << "\n"
            << state.levelNumber << "\n"
            << state.health << "\n"
            << state.eatingResources << "\n"
            << state.otherResources << "\n"
    outFile.close();
    return true;
}

// Function to load the game state from a file
bool loadGame(GameState &state, const string &filename)
{
    ifstream inFile(filename);
    if (!inFile)
    {
        cerr << "Error: Could not open file for loading.\n";
        return false;
    }
    // Read game state variables line by line
    getline(inFile, state.sessionName);
    inFile >> state.levelNumber
           >> state.health
           >> state.eatingResources
           >> state.otherResources;
    inFile.ignore(); // Ignore the remaining newline before reading
capyName
    getline(inFile, state.capyName);
    inFile.close();
    return true;
}
```

Explanation:

- **saveGame:**
  - Opens a file for writing.
  - Writes each `GameState` attribute on separate lines for easy parsing.
  - Returns `true` if saving is successful; otherwise, `false`.
- **loadGame:**
  - Opens a file for reading.
  - Reads each `GameState` attribute in the same order they - were saved.
  - Returns `true` if loading is successful; otherwise, `false`.

2.4 `main.cpp`

Purpose: Serves as the entry point of the program. It handles user interactions for loading existing sessions or starting new ones and initiates the main game loop.

Source: `src/main.cpp`

```
#include <iostream>
#include <string>
#include <filesystem>
#include "game_state.hpp"
#include "save_system.hpp"
#include "run_game.hpp"
#include "ascii_art.hpp"

using namespace std;
namespace fs = std::filesystem;

int main()
{
    // Display the cover page using ASCII art
    printCoverPage();

    // Ensure the "saves" directory exists; create it if it doesn't
    fs::path saveDir{"saves"};
    if (!fs::exists(saveDir))
    {
        fs::create_directory(saveDir);
    }

    // Gather up to 10 existing save files
    string saveFiles[10];
    int fileCount = 0;

    for (auto &entry : fs::directory_iterator(saveDir))
    {
        if (entry.is_regular_file() && fileCount < 10)
        {
            saveFiles[fileCount++] = entry.path().string();
        }
    }
    // Display existing save files to the user
    cout << "\nAvailable Save Files:\n";
    if (fileCount == 0)
    {
        cout << "[No saves found – will start new]\n";
    }
    else
    {
        for (int i = 0; i < fileCount; i++)
        {
            cout << (i + 1) << ". " << saveFiles[i] << "\n";
        }
        cout << "N. New Session\n";
    }

    // Prompt the user to choose a save file or start a new session
    cout << "\nEnter your Choice (N/1/2...): ";
    string choice;
    cin >> choice;

    GameState state; // Initialize with default values

    if (choice == "N" || choice == "n" || fileCount == 0)
    {
        // Start a new game session
        cout << "Enter session name (no spaces): ";
        cin >> state.sessionName;
        cout << "Name your Capybara: ";
        cin >> state.capyName;
        cout << "\nStarting a NEW game...\n";
    }
    else
    {
        // Attempt to load an existing save file
        int idx = stoi(choice) - 1;
        if (idx >= 0 && idx < fileCount)
        {
            cout << "Loading from: " << saveFiles[idx] << endl;
            bool ok = loadGame(state, saveFiles[idx]);
            if (!ok)
            {
                cout << "Load failed. Starting New.\n";
                // Optionally, you could reset the GameState or handle
differently
            }
            else
            {
                cout << "Load success.\nLevel=" << state.levelNumber
                     << "\nHealth=" << state.health
                     << "\nEatingRes=" << state.eatingResources
                     << "\nOtherRes=" << state.otherResources << "\n";
            }
        }
        else
        {
            cout << "Invalid choice. Starting New.\n";
            // Optionally, prompt again or handle invalid input
        }
    }

    // Display the start page and start the game
    delay(1);                   // Pause for 1 second
    clearScreen();              // Clear the console screen
    printGameStart();           // Print Game Start page
    cout << endl                // Prompt user to press enter
         << "PRESS ENTER TO CONTINUE: ";
    cout.flush();
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    getline(cin, user_input);
    clearScreen();              // Clear the console

    // Start the main game loop
    runGame(state);

    return 0;
}
```

Explanation:

- **Cover Page:** Displays initial ASCII art.
- **Save Directory Check:** Ensures that the saves/ directory exists; creates it if not.
- **Save File Enumeration:** Lists up to 10 existing save files for the user to choose from.
- **User Choice:** Allows the player to select an existing save or start a new session.
- **Game Initialization:** Sets up the GameState based on user input or loaded data.
- **Game Loop Initiation:** Calls runGame(state) to start the main game loop.

2.5. Example Level: Level 1

Purpose: Demonstrates how a single level (Level 1) is implemented, including player choices, resource management, and interaction with a minigame.

Header: `include/Levels/lvl1.hpp`

```
#ifndef LVL1_HPP
#define LVL1_HPP

#include "game_state.hpp"

void lvl1(GameState &state);

#endif
```

Source: `src/Levels/lvl1.cpp`

```
#include "Levels/lvl1.hpp"
#include "Minigames/spin_the_wheel.hpp"
#include "save_system.hpp"
#include "ascii_art.hpp"
#include <iostream>
using namespace std;

void lvl1(GameState &state)
{
    int choice, outcome;
    bool flag = false;

    // Prints "Level 1" and Relevant Graphics:
    gameScreen(1);

    // Prints the player's current resources and game status:
    displayStatus(state);

    cout << "\nCapy must either:\n1. Build a simple shelter\n2. Save
materials\n";

    // Validating the Choice:
    do
    {
        cin >> choice;
        // logic...
    } while (!flag);

    // Makes the system sleep for 3 seconds:
    delay(3);

    // Calls a minigame:
    outcome = spinTheWheel();

    // Adjust resources accordingly:
    if (result = ...)
    {
        // logic
    }
    else
    {
        // ...
    }

    // End-of-level
    state.levelNumber = 2;

}
```

Explanation:

- **Level Screen:** Uses gameScreen(1) to display Level 1.
- **Player Status:** Displays current status using displayStatus(state).
- **Player Choices:** Allows user to respond the scenario based choices.
- **Minigame Interaction:** Allows user to play and get rewarded as per the outcome of the minigames
- **Level Completion:** Notifies the player of level completion and sets up for the next level.

2.6. Example Minigame: Memory Match

Purpose: A minigame where the player must remember and identify a randomly selected card. Correct guesses reward the player with resources, while incorrect ones penalize their health.

Header: `include/Minigames/memory_match.hpp`

```
#ifndef MEMORY_MATCH_HPP
#define MEMORY_MATCH_HPP

// Function to execute the memory match minigame
bool memoryMatch();

#endif // MEMORY_MATCH_HPP
```

*Header:** `src/Minigames/memory_match.cpp`

```
// src/Minigames/memory_match.cpp
#include "Minigames/memory_match.hpp"
#include "ascii_art.hpp"
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

// Function to perform the memory match minigame
bool memoryMatch()
{
    string answer;
    int card;

    srand(time(nullptr));      // Seed the random number generator
    card = (rand() % 3) + 1;   // Randomly select a card (1, 2, or 3)

    printCards(card);          // Display the selected card using ASCII
art

    cout << "\nWhat card was it: ";
    cin >> answer;

    // Check if the player's answer matches the selected card
    if (((card == 1) && (answer == "9")) ||
        ((card == 2) && (answer == "S")) ||
        ((card == 3) && (answer == "P")))
    {
        return true; // Player guessed correctly
    }
    else
    {
        return false; // Player guessed incorrectly
    }
}
```

Explanation:

- **Random Selection:** Chooses a random card between 1 and 3.
- **Display:** Uses `printCards` to show the selected card in ASCII art.
- **User Guess:** Prompts the player to guess which card was displayed.
- **Validation:** Determines if the player's guess is correct:
  - Card 1: Should input `"9"`.
  - Card 2: Should input `"S"`.
  - Card 3: Should input `"P"`.
- **Outcome:**
  - **Correct Guess:** Returns `true`, rewarding the player.
  - **Incorrect Guess:** Returns `false`, penalizing the player's health.

2.7. `ascii_art.cpp`

Purpose: Enhances the game's visual appeal by animating minigames, capybara, etc. using ASCII art frames.

```
Code to large to be documented
```

Explanation:

- **printCoverPage**: Displays the game's ASCII art cover.
- **clearScreen**: Clears the console using ANSI escape codes for a cleaner interface.
- **GameOverScreen**: Shows a game over message in ASCII art when the player loses.
- **printCards**: Displays the selected card in the Memory Match minigame based on the random number.
- **displayWalkingCapybara**: Animates a simple walking capybara using multiple frames and delays.
- **printMinigameName**: Displays the name of the current minigame.
- **printSpinningWheel**: Shows the spinning wheel graphic for the Spin the Wheel minigame.
- **gameScreen**: Clears the screen and displays the current level number.
- **displayStatus**: Shows the player's current status, including name, level, health, and resources.
- **delay**: Pauses the program for a specified number of seconds to control animation timing and user experience.

---

3. Compilation Instructions

To compile the Island Escape game, ensure you have a C++17 compatible compiler (like `g++`) installed. Navigate to the project's root directory and execute the following command:

```
g++ -std=c++17 -Iinclude \
    src/main.cpp \
    src/save_system.cpp \
    src/run_game.cpp \
    src/ascii_art.cpp \
    src/Levels/*.cpp \
    src/Minigames/*.cpp \
    -o Game
```

```
Disclaimer: To ensure proper rendering of ASCII art and avoid display
issues, please use macOS or Linux systems.
```

Running the Game: After successful compilation, run the game using:

```
./Game
```

Note: Ensure that the `saves/` directory exists in the correct relative path as expected by the program. The game will attempt to create it if it doesn't exist.

---

4. Conclusion:

The **Capybara Chronicles** game offers an engaging text-based adventure with multiple levels and minigames, providing a balance of challenge and resource management. Key features include:

- **Structured Codebase:** Organized into i`nclude/` and `src/` directories with subdirectories for levels and minigames.
- **Game State Management:** Utilizes a `GameState` struct to track player progress and resources.
- **Save/Load Functionality:** Allows players to save their progress and resume later.
- **Interactive Minigames: **Incorporates various minigames like Memory Match and Spin the Wheel to diversify gameplay.
- **ASCII Art Enhancements:** Enhances visual appeal with ASCII animations and art for cover pages, game over screens, and character animations.