Submission:

- Your submission will be completed by uploading your entire C++ solution, including all source code, as a ZIP file, to the Midterm Project drop box on SLATE, before the due date/time.
- You must also upload a link to the group GitHub repo used to manage this project.

Assignment details and tasks

- Create a new C++/SDL application, called ***SDLLevels***
- Create a GitHub repo. Share the repo details with the entire group. All group members must use this single repo. It will serve to track all group members' submits and contributions to the project
- The application must:
    - Use proper resource and asset management. Resources and assets must be serializable/de-serializable
    - Organize all instances of these resources, rendering functionality, game updates etc. in Levels, that must also be serializable
    - Run at 1920x1080 native resolution
- Create a game that has 2 unique levels
- The first level must:
    - have a grey background (R = G = B = 128)
    - have 10 animated warriors running from the left to the right of the screen (y positions are 10, 110, 210 etc.):
        - **Running speed (random per warrior):** 80 to 100 pixels per second
        - **Scale:** 1.8x
        - **Animation speed:** Between 4.8 and 6. Linearly increase animation speed as running speed increases. I.e., the faster the warrior runs, the faster the animation speed should be. Slowest warrior's animation speed is 4.8. Fastest warrior's animation speed is 6.
    - auto-save the level state after 5 seconds as **Level1.bin**, immediately load the auto-saved level (**Level1.bin**) and continue level execution.
    - have blue labels at the top of the screen showing FPS, Game Time in seconds, and a status indicator showing whether the level has been auto saved at 5 seconds
    - automatically load Level 2 as soon as the first warrior disappears off-screen
- The second level must:
    - have a light-green background (R = 0; G = 128; B = 0)

- contain the same warriors as Level 1
- have 10 animated rocks falling from the top to the bottom of the screen (x positions are 50, 150, 250 etc.):
  - **Falling speed (random per rock):** 80 to 100 pixels per second
  - **Scale:** 1.0x
  - **Animation speed:** Between 4.8 and 6. Linearly increase animation speed as falling speed increases. I.e., the faster the rock falls, the faster the animation speed should be. Slowest rock's animation speed is 4.8. Fastest rock's animation speed is 6.
- start playing a warrior death animation when a rock hits a warrior:
  - the rock that hit the warrior must be removed from the level
  - the warrior must be removed from the level as soon as the death animation is complete
- contain the same blue labels as Level 1
- auto-save the level state after 5 seconds as **Level2.bin**, immediately load the auto-saved level (**Level2.bin**) and continue level execution.
- automatically quit the application as soon as the first warrior disappears off-screen, or if all warriors are dead

Sample output:


Frames Per Second: 56
Game Time: 8
Auto Saved: Yes

Code Documentation

October 26, 2024

Contents

www.assignmentcompletion.com

www.assignmentcompletion.com

1 Introduction

In this project, I developed a game application using C++ and SDL, focusing on key aspects of game development such as rendering, level management, and asset handling. The objective was to build a functioning game environment with multiple levels, incorporating efficient resource management and smooth gameplay mechanics. Throughout the project, I implemented game logic, handled animations, and ensured optimization for better performance. This documentation outlines the technical details of my approach, including the design decisions, code structure, and the challenges encountered during development.

2 C++ Code

2.1 `Asset.cpp`

```cpp
#include "Asset.h"

// Initialize the static ObjectPool for Asset instances.
// This pool manages the reuse of Asset objects to optimize memory usage.
ObjectPool<Asset>* Asset::Pool;

// Constructor: Initializes the Asset object with default values.
// The GUID is set to an empty string, the data size is initialized to 0, and the data pointer is set to null.
Asset::Asset()
{
    m_GUID = "";        // Set the GUID to an empty string.
    m_dataSize = 0;     // Initialize the size of the data to 0.
    m_data = nullptr;   // Set the data pointer to null (no data yet).
}

// Destructor: Default destructor for the Asset class.
// Since no dynamic memory is allocated directly by the class, no special cleanup is needed here.
```


www.assignmentcompletion.com

```cpp
Asset::~Asset() {}

// ToString method: Outputs the Asset's details (GUID and data size) to the console.
// This method is useful for debugging and logging purposes to get information about the asset.
void Asset::ToString()
{
    cout << "Asset GUID: " << m_GUID << endl;        // Output the asset's GUID.
    cout << "Asset Data Size: " << m_dataSize << endl; // Output the size of the asset's data.
}
```

Listing 1: C++ and SDL Program - Asset.cpp

2.2 Asset.h

```cpp
#ifndef ASSET_H
#define ASSET_H

#include "StandardIncludes.h"
#include "ObjectPool.h"

// The Asset class is designed to represent a generic asset with a unique identifier (GUID)
// and associated data. This class also uses an object pool to optimize memory allocation
// and deallocation.

class Asset
{
public:
    // Constructor: Initializes the asset object.
    Asset();

    // Destructor: Cleans up resources when the asset is destroyed.
    virtual ~Asset();
```

```
19
20      // Returns the globally unique identifier (GUID) of
            the asset.
21      string GetGUID() { return m_GUID; }
22
23      // Sets the GUID of the asset.
24      void setGUID(string _guid) { m_GUID = _guid; }
25
26      // Returns the size of the data associated with the
            asset.
27      int GetDataSize() { return m_dataSize; }
28
29      // Sets the size of the data associated with the asset.
30      void SetDataSize(int _size) { m_dataSize = _size; }
31
32      // Returns a pointer to the asset's data.
33      byte* GetData() { return m_data; }
34
35      // Sets the asset's data by providing a pointer to the
            data.
36      void SetData(byte* _data) { m_data = _data; }
37
38      // Resets the asset to its initial state.
39      void Reset() { }
40
41      // Converts the asset information into a string
            representation (implementation missing).
42      void ToString();
43
44      // Static object pool used for managing Asset
            instances, to improve performance by reusing
45      // allocated memory instead of frequently allocating
            and deallocating memory.
46      static ObjectPool<Asset>* Pool;
47
48  private:
49      // Unique identifier (GUID) for the asset.
50      string m_GUID;
51
52      // The size of the data associated with the asset.
```

```
    int m_dataSize;

    // Pointer to the data associated with the asset.
    byte* m_data;
};

#endif // ASSET_H
```

Listing 2: C++ and SDL Program - Asset.h

2.3 AssetController.cpp

```
#include "AssetController.h"
#include "FileController.h"

// Initialize the static StackAllocator pointer.
// The stack is used to allocate memory for assets in a
   stack-like manner.
StackAllocator* AssetController::Stack = nullptr;

// Constructor: Initializes the AssetController object.
AssetController::AssetController() {}

// Initialize method: Sets up the stack allocator for the
   AssetController and creates the Asset pool.
// This method takes a stack size (_stackSize) as an
   argument and allocates memory for the stack.
void AssetController::Initialize(int _stackSize)
{
    // Create a new StackAllocator and allocate the stack
       with the specified size.
    Stack = new StackAllocator();
    AssetController::Stack->AllocateStack(_stackSize);

    // Create a new object pool for managing Asset
       instances.
    Asset::Pool = new ObjectPool<Asset>();
}

```

www.assignmentcompletion

```cpp
// Destructor: Cleans up resources when the AssetController is destroyed.
AssetController::~AssetController()
{
    // Clear all assets and release resources.
    Clear();
}

// Clear method: Releases all assets from the pool, clears the stack memory, and empties the asset map.
// This ensures that all memory is properly cleaned up when the AssetController is no longer needed.
void AssetController::Clear()
{
    // Loop through all assets in the map and release them back to the object pool.
    for (auto const& x : m_assets)
    {
        Asset::Pool->ReleaseResource(x.second);
    }

    // If the object pool exists, delete it and set it to nullptr.
    if (Asset::Pool != nullptr)
    {
        delete Asset::Pool;
        Asset::Pool = nullptr;
    }

    // Clear the stack memory.
    Stack->ClearMemory();

    // Clear the map of loaded assets.
    m_assets.clear();
}

// GetAsset method: Retrieves an asset based on its GUID. If the asset is already loaded, it returns the cached asset.
```


www.assignmentcompletion

```cpp
// Otherwise, it loads the asset from the file, stores it in the asset map, and returns it.
Asset* AssetController::GetAsset(string _guid)
{
    // If the asset is already loaded, return the cached asset.
    if (m_assets.count(_guid) != 0)
    {
        return m_assets[_guid];
    }

    // Otherwise, load the asset.
    Asset* asset = Asset::Pool->GetResource();  // Get a new asset from the pool.
    asset->setGUID(_guid);                      // Set the asset's GUID.
    asset->SetDataSize(FileController::Instance().GetFileSize(_guid));
        // Get the asset's size from the file controller.
    asset->SetData(Stack->GetMemory(asset->GetDataSize()));
                    // Allocate memory for the asset's data in the stack.
    FileController::Instance().ReadFile(_guid, asset->GetData(), asset->GetDataSize());  // Load the asset data from the file.

    // Add the newly loaded asset to the asset map for future use.
    m_assets[_guid] = asset;

    // Return the loaded asset.
    return asset;
}
```

Listing 3: C++ and SDL Program - AssetController.cpp

2.4 AssetController.h

```cpp
#ifndef ASSET_CONTROLLER_H
#define ASSET_CONTROLLER_H
```

```
3
4  #include "StandardIncludes.h"
5  #include "StackAllocator.h"
6  #include "Asset.h"
7
8  // The AssetController class manages assets in the
       application and provides access
9  // to assets using their GUIDs. It uses a stack allocator
       for memory management
10 // and follows the Singleton design pattern to ensure a
       single instance exists.
11
12 class AssetController : public Singleton<AssetController>
13 {
14 public:
15     // Constructor: Initializes the AssetController object.
16     AssetController();
17
18     // Destructor: Cleans up resources when the
           AssetController is destroyed.
19     virtual ~AssetController();
20
21     // Initializes the AssetController with a specified
           stack size for the allocator.
22     void Initialize(int _stackSize);
23
24     // Clears all managed assets and resets the asset
           storage.
25     void Clear();
26
27     // Retrieves an asset by its GUID from the internal
           asset map. Returns nullptr if the asset is not
           found.
28     Asset* GetAsset(string _guid);
29
30     // Static stack allocator for managing memory of the
           assets.
31     static StackAllocator* Stack;
32
33 private:
```


www.assignmentcompletion

```
    // A map that stores assets, using their GUIDs as keys
       for fast lookup.
    map<string, Asset*> m_assets;
};

#endif //ASSET_CONTROLLER_H
```

Listing 4: C++ and SDL Program - AssetController.h

2.5 BasicStructs.h

```
#ifndef BASICSTRUCTS_H
#define BASICSTRUCTS_H

#ifndef _WIN32
#include <Windows.h>
#else
// Define a byte type for platforms that don't include
   Windows headers.
typedef unsigned char byte;
#endif

// The Color struct represents an RGBA color value using
   8-bit channels for red, green, blue, and alpha.
struct Color
{
    // Constructor: Initializes the color with the
       specified red, green, blue, and alpha values.
    Color(byte _r, byte _g, byte _b, byte _a)
    {
        R = _r;
        G = _g;
        B = _b;
        A = _a;
    }

    // Red, Green, Blue, and Alpha channels, each stored
       as a byte (0-255).
    byte R;
```


www.assignmentcompletion

```
    byte G;
    byte B;
    byte A;
};

// The Point struct represents a point on a 2D plane, defined by its X and Y coordinates.
struct Point
{
    // Constructor: Initializes the point with the specified X and Y coordinates.
    Point(unsigned int _x, unsigned int _y)
    {
        X = _x;
        Y = _y;
    }

    // X and Y coordinates of the point.
    unsigned int X;
    unsigned int Y;
};

// The Rect struct represents a rectangular area in 2D space, defined by two corner points (X1, Y1) and (X2, Y2).
struct Rect
{
    // Constructor: Initializes the rectangle using two corner points (X1, Y1) and (X2, Y2).
    Rect(unsigned int _x1, unsigned int _y1, unsigned int _x2, unsigned int _y2)
    {
        X1 = _x1;
        Y1 = _y1;
        X2 = _x2;
        Y2 = _y2;
    }

    // Coordinates of the first corner of the rectangle.
    unsigned int X1;
```


assignmentcompletion

```
    unsigned int Y1;

    // Coordinates of the opposite corner of the rectangle.
    unsigned int X2;
    unsigned int Y2;
};

#endif //BASICSTRUCTS_H
```

Listing 5: C++ and SDL Program - BasicStructs.h

2.6 FileController.cpp

```
#include "FileController.h"

// Constructor: Initializes the FileController object.
// Sets the file handle to null, the read success flag to
   false, and initializes the thread object.
FileController::FileController()
{
    m_handle = nullptr;         // File handle is initially
       null (no file open).
    m_readSuccess = false;      // Initially, no file has
       been read successfully.
    m_thread = {};              // Initialize the thread
       object.
}

// Destructor: Cleans up any resources when the
   FileController is destroyed.
// Since no dynamic memory is allocated here, the
   destructor is empty.
FileController::~FileController()
{
}

// GetCurDirectory method: Retrieves the current working
   directory as a string.
// Uses platform-specific functionality to get the
   directory, and asserts if it fails.
```

www.assignmentcompletion

```cpp
string FileController::GetCurDirectory()
{
    char buff[FILENAME_MAX];  // Buffer to hold the directory path.
    M_ASSERT(GetCurrentDir(buff, FILENAME_MAX) != nullptr, "Could not get current directory.");
    return string(buff);      // Return the directory path as a string.
}

// GetFileSize method: Opens a file in binary mode and returns its size in bytes.
// If the file cannot be opened or an error occurs, it returns -1.
int FileController::GetFileSize(string _filePath)
{
    m_handle = nullptr;  // Reset the file handle.

    // Attempt to open the file in binary mode ("rb" = read binary).
    if (fopen_s(&m_handle, _filePath.c_str(), "rb") != 0 || !m_handle)
    {
        std::cerr << "Error: Could not open file: " << _filePath << std::endl;
        return -1;
    }

    // Seek to the end of the file to determine its size.
    if (fseek(m_handle, 0, SEEK_END) != 0)
    {
        std::cerr << "Error: Could not seek to end of file: " << _filePath << std::endl;
        fclose(m_handle);  // Close the file on error.
        return -1;
    }

    // Get the current position in the file (which is the file size).
    int fileSize = std::ftell(m_handle);
```


assignmentcompletion

```
    if (fileSize == -1L)
    {
        std::cerr << "Error: Could not determine file size: " << _filePath << std::endl;
        fclose(m_handle);  // Close the file on error.
        return -1;
    }

    fclose(m_handle);  // Close the file after determining the size.
    return fileSize;
}

// ReadFile method: Reads the contents of a file into the provided buffer.
// Returns true if the file was read successfully, false otherwise.
bool FileController::ReadFile(string _filePath, unsigned char* _buffer, unsigned int _bufferSize)
{
    m_handle = nullptr;  // Reset the file handle.
    m_readSuccess = false;  // Reset the success flag.

    // Open the file in binary mode and ensure it was opened successfully.
    M_ASSERT(fopen_s(&m_handle, _filePath.c_str(), "rb") == 0, "Could not open file.");
    if (m_handle != nullptr)
    {
        // Attempt to read the specified number of bytes into the buffer.
        M_ASSERT(fread(_buffer, 1, _bufferSize, m_handle) == _bufferSize, "All bytes not read from file.");
        M_ASSERT(ferror(m_handle) == 0, "Error reading from file.");  // Check for file read errors.
        M_ASSERT(fclose(m_handle) == 0, "Could not close file");      // Ensure the file is closed properly.
```

www.assignmentcompletion

```
        m_readSuccess = true;  // Mark the read operation as successful.
    }

    // If the thread is still joinable, detach it to avoid blocking.
    if (m_thread.joinable())
    {
        m_thread.detach();
    }

    return m_readSuccess;
}

// ReadFileAsync method: Reads the contents of a file asynchronously.
// This method spawns a separate thread to read the file in the background.
void FileController::ReadFileAsync(string _filePath, unsigned char* _buffer, unsigned int _bufferSize)
{
    // Create a new thread to run the ReadFile method in the background.
    m_thread = std::thread(&FileController::ReadFile, this, _filePath, _buffer, _bufferSize);
}
```

Listing 6: C++ and SDL Program - FileController.cpp

2.7 FileController.h

```
#ifndef FILE_CONTROLLER_H
#define FILE_CONTROLLER_H

#include "StandardIncludes.h"

// The FileController class is responsible for managing file operations such as reading files,
// getting file sizes, and handling asynchronous file reads. It follows the Singleton design pattern
```

```
// to ensure only one instance is used throughout the
   application.

class FileController : public Singleton<FileController>
{
public:
    // Constructor: Initializes the FileController object.
    FileController();

    // Destructor: Cleans up resources when the
       FileController is destroyed.
    virtual ~FileController();

    // Checks if the file read operation has finished by
       verifying if the thread is no longer joinable.
    bool GetFileReadDone() { return !m_thread.joinable(); }

    // Returns whether the file read operation was
       successful or not.
    bool GetFileReadSuccess() { return m_readSuccess; }

    // Retrieves the current working directory of the
       application.
    string GetCurDirectory();

    // Returns the size of the file at the specified file
       path.
    int GetFileSize(string _filePath);

    // Synchronously reads a file from the specified file
       path into the provided buffer.
    // Returns true if the read operation is successful,
       false otherwise.
    bool ReadFile(string _filepath, unsigned char*
       _buffer, unsigned int _bufferSize);

    // Asynchronously reads a file from the specified file
       path into the provided buffer.
    // The read operation is performed on a separate
       thread.
```


assignmentcompletion

```
    void ReadFileAsync(string _filePath, unsigned char*
       _buffer, unsigned int _bufferSize);

private:
    // File handle for managing file operations.
    FILE* m_handle;

    // Thread for asynchronous file reading.
    thread m_thread;

    // Boolean flag indicating whether the file read
       operation was successful.
    bool m_readSuccess;
};

#endif //FILE_CONTROLLER_H
```

Listing 7: C++ and SDL Program - FileController.h

2.8 GameController.cpp

```
#include "GameController.h"
#include "Renderer.h"
#include "SpriteSheet.h"
#include "SpriteAnim.h"
#include "TTFont.h"
#include "Timing.h"
#include "Level.h"

// Constructor: Initializes the GameController with
   default values.
// The game starts in LEVEL1, and autoSave is set to false.
GameController::GameController()
    : currentState(GameState::LEVEL1), gameTime(0.0f),
       autoSaved(false)
{
    m_sdlEvent = {};  // Initialize SDL event handling.
}

```

```
// Destructor: Default destructor for GameController.
GameController::~GameController() {}

// Timing instance used to manage delta time and frame rates.
Timing* t = &Timing::Instance();

// RandomNumber function: Generates a random number between min and max (inclusive).
// This function may be relocated or refactored later as necessary.
int RandomNumber(int min, int max)
{
    int random = (min + rand() % (max - min + 1));
    return random;
}

// RunGame method: This is the main game loop, handling initialization, game state transitions,
// rendering, and logic for both levels (LEVEL1 and LEVEL2).
void GameController::RunGame()
{
    // Initialize AssetController with a stack size of 10MB for asset memory management.
    AssetController::Instance().Initialize(10000000);

    // Create and initialize the Renderer instance for window and rendering setup.
    Renderer* r = &Renderer::Instance();
    r->Initialize(1920, 1080);  // Set the window size (1920x1080).

    // Initialize font for rendering text in the game.
    TTFont* font = new TTFont();
    font->Initialize(20);

    // Get the window size for future use (if needed).
    Point ws = r->GetWindowSize();

```


assignmentcompletion

```
    // Create object pools for SpriteSheet and SpriteAnim
       to manage resources efficiently.
    SpriteSheet::Pool = new ObjectPool<SpriteSheet>();
    SpriteAnim::Pool = new ObjectPool<SpriteAnim>();

    // Placeholder for sprite sheet resources (loading
       disabled).
    SpriteSheet* sheetRock =
       SpriteSheet::Pool->GetResource();

    // Initialize level objects for LEVEL1 and LEVEL2.
    Level* level1 = new Level(r, font);
    Level* level2 = new Level(r, font);

    // Main game loop, continues until SDL_QUIT event is
       triggered.
    while (m_sdlEvent.type != SDL_QUIT)
    {
        t->Tick();  // Update delta time and frame count.
        gameTime += t->GetDeltaTime();  // Increment game
           time.
        SDL_PollEvent(&m_sdlEvent);  // Poll for SDL
           events (e.g., input, window events).

        // Handle game state (LEVEL1 or LEVEL2).
        switch (currentState)
        {
        case GameState::LEVEL1:
        {
            // Run logic for Level 1, passing deltaTime
               for frame consistency and gameTime for
               total time.
            level1->RunLevel1Logic(t->GetDeltaTime(),
               gameTime);

            // Auto-save after 5 seconds of game time.
            if (!autoSaved && gameTime >= 5.0f)
            {
                ofstream writeStream("level1.bin",
                   ios::out | ios::binary);
```

```
                level1->Serialize(writeStream);  // Save the level state to a binary file.
                autoSaved = true;
                level1->SetAutoSaveStatus("AutoSave:Yes");
                writeStream.close();

                // Load the saved level for testing purposes.
                Level* loadedLevel = new Level(r, font);
                ifstream readStream("level1.bin", ios::in | ios::binary);
                loadedLevel->Deserialize(readStream);
                readStream.close();
            }

            // Transition to Level 2 if the condition is triggered.
            if (level1->Level2TransitionTriggered())
            {
                currentState = GameState::LEVEL2;  // Switch to LEVEL2 state.
                level1->SetAutoSaveStatus("AutoSave:No");
                autoSaved = false;
                gameTime = 0.0f;  // Reset game time.
                std::cout << "Transitioning to Level 2" << std::endl;
            }

            break;
        }
        case GameState::LEVEL2:
        {
            // Run logic for Level 2, passing deltaTime and gameTime.
            level2->RunLevel2Logic(t->GetDeltaTime(), gameTime);

            // Auto-save after 5 seconds of game time in Level 2.
            if (!autoSaved && gameTime >= 5.0f)
```


www.assignmentcompletion

```
            {
                ofstream writeStream("level2.bin",
                   ios::out | ios::binary);
                level2->Serialize(writeStream);
                autoSaved = true;
                level2->SetAutoSaveStatus("AutoSave:Yes");
                writeStream.close();

                // Load the saved level for testing
                   purposes.
                Level* loadedLevel = new Level(r, font);
                ifstream readStream("level2.bin", ios::in
                   | ios::binary);
                loadedLevel->Deserialize(readStream);
                readStream.close();
            }

            // Quit the game if the condition to end Level
               2 is triggered.
            if (level2->Level2EndTriggered())
            {
                SDL_Quit();  // Exit the game by quitting
                   SDL.
            }

            break;
        }
        }

        // Present the rendered frame on the screen.
        SDL_RenderPresent(r->GetRenderer());
    }

    // Clean up and delete the object pools after the game
       loop exits.
    delete SpriteAnim::Pool;
    delete SpriteSheet::Pool;

    // Shut down the renderer and clean up SDL resources.
    r->Shutdown();
```

```
}
```

Listing 8: C++ and SDL Program - GameController.cpp

2.9 GameController.h

```
#ifndef GAME_CONTROLLER_H
#define GAME_CONTROLLER_H

#include "StandardIncludes.h"

// Enumeration representing different game states.
// LEVEL1 and LEVEL2 are two possible levels in the game.
enum GameState
{
    LEVEL1,  // Represents the first level of the game.
    LEVEL2   // Represents the second level of the game.
};

// The GameController class manages the game's core loop, states, and events.
// It uses the Singleton design pattern to ensure that only one instance exists during the game's runtime.
class GameController : public Singleton<GameController>
{
public:
    // Constructor: Initializes the GameController object.
    GameController();

    // Destructor: Cleans up resources when the GameController is destroyed.
    virtual ~GameController();

    // The main game loop. This function runs the game by handling events, updating game states,
    // and rendering content based on the current game state.
    void RunGame();

```

www.assignmentcompletion.com

```
    // Boolean flag to indicate if the game has been auto-saved.
    bool autoSaved;

private:
    // SDL event structure used for handling input events such as keyboard and mouse input.
    SDL_Event m_sdlEvent;

    // The current state of the game (e.g., LEVEL1 or LEVEL2).
    GameState currentState;

    // The elapsed game time, used for updating the game state over time.
    float gameTime;
};

#endif //GAME_CONTROLLER_H
```

Listing 9: C++ and SDL Program - GameController.h

2.10 Level.cpp

```
#include "Level.h"
#include <random>
#include <stdbool.h>

// Constructor: Initializes the Level object with default values and generates random speeds, warrior sheets, and rock sheets.
Level::Level(Renderer* renderer, TTFont* font)
    : rectX(0.0f), rectAsh(0.0f), scale(1.8f),
    spriteWidth(69), spriteHeight(44), currentFrame(0),
    renderer(renderer), font(font), autoSaved(false), m_autoSaveStatus("AutoSave:No"), autoSaveMsgTimer(0.0f),
    m_warriorXPositions(10, 0.0f), m_rockYPositions(10, 0.0f),
```

```
    m_warriorIsAlive(10, true), m_rockIsAlive(10, true),
       m_warriorDeathState(10, false),
    scaleRock(1.0f), spriteWidthRock(20),
       spriteHeightRock(20),
    viewportEdge(1920)
{
    if (!isGenerated) GenerateRandomSpeeds();
    GenerateWarriorSheets();
    GenerateRockSheets();
}

// Destructor: Default destructor for the Level object.
Level::~Level() {}

// Serialize method: Saves the level's data (such as
   positions and dimensions) to the provided output stream.
void Level::Serialize(std::ostream& _stream)
{
    _stream.write(reinterpret_cast<char*>(&rectX),
       sizeof(rectX));
    _stream.write(reinterpret_cast<char*>(&rectAsh),
       sizeof(rectAsh));
    _stream.write(reinterpret_cast<char*>(&currentFrame),
       sizeof(currentFrame));
    _stream.write(reinterpret_cast<char*>(&scale),
       sizeof(scale));
    _stream.write(reinterpret_cast<char*>(&spriteWidth),
       sizeof(spriteWidth));
    _stream.write(reinterpret_cast<char*>(&spriteHeight),
       sizeof(spriteHeight));
    std::cout << "Level Saved Successfully" << std::endl;
    Resource::Serialize(_stream); // Call base class
       serialization.
}

// Deserialize method: Loads the level's data from the
   provided input stream.
void Level::Deserialize(std::istream& _stream)
{
```

www.assignmentcompletion

```
    _stream.read(reinterpret_cast<char*>(&rectX),
       sizeof(rectX));
    _stream.read(reinterpret_cast<char*>(&rectAsh),
       sizeof(rectAsh));
    _stream.read(reinterpret_cast<char*>(&currentFrame),
       sizeof(currentFrame));
    _stream.read(reinterpret_cast<char*>(&scale),
       sizeof(scale));
    _stream.read(reinterpret_cast<char*>(&spriteWidth),
       sizeof(spriteWidth));
    _stream.read(reinterpret_cast<char*>(&spriteHeight),
       sizeof(spriteHeight));
    Resource::Deserialize(_stream); // Call base class
       deserialization.
    std::cout << "Level Loaded Successfully" << std::endl;
}

// SetAutoSaveStatus method: Updates the auto-save status
   message.
void Level::SetAutoSaveStatus(const std::string& _status)
{
    m_autoSaveStatus = _status;
}

// InitializeWarriorPositions method: Sets all warrior X
   positions to 0.
void Level::InitializeWarriorPositions(std::vector<float>
   _warriorXPositions)
{
    _warriorXPositions.assign(10, 0.0f);
}

// GenerateRandomSpeeds method: Generates random speeds
   for the warriors.
void Level::GenerateRandomSpeeds() {
    std::random_device rd;
    std::uniform_int_distribution<int> dist(80, 100); //
       Random speed between 80 and 100.

    m_randSpeeds.resize(10); // Resize to hold 10 speeds.
```

```
    for (int& speed : m_randSpeeds) {
        speed = dist(rd); // Store the random speed.
        std::cout << "Generated Speed: " << speed <<
           std::endl;
    }

    isGenerated = true;
}

// GenerateWarriorSheets method: Loads the warrior sprite
   sheets and assigns animations (run and death).
void Level::GenerateWarriorSheets()
{
    for (int i = 0; i < 10; i++)
    {
        SpriteSheet* sheet =
           SpriteSheet::Pool->GetResource();
        sheet->Load("../Assets/Textures/Warrior.tga");
        sheet->SetSize(17, 6, 69, 44);
        sheet->AddAnimation(EN_AN_RUN, 6, 7,
           static_cast<float>(m_randSpeeds[i]) / 100.0f *
           6.0f);
        sheet->AddAnimation(EN_AN_DEATH, 27, 10,
           static_cast<float>(m_randSpeeds[i]) / 100.0f *
           6.0f);

        m_warriorSheets.push_back(sheet);
    }
}

// GenerateRockSheets method: Loads the rock sprite sheets
   and assigns falling animation.
void Level::GenerateRockSheets()
{
    for (int i = 0; i < 10; i++)
    {
        SpriteSheet* sheet =
           SpriteSheet::Pool->GetResource();
        sheet->Load("../Assets/Textures/Rock.tga");
        sheet->SetSize(1, 4, 20, 20);
```

```
98          sheet->AddAnimation(ROCK_FALL, 0, 4,
                static_cast<float>(m_randSpeeds[i]) / 100.0f *
                6.0f);
99
100         m_rockSheets.push_back(sheet);
101     }
102 }
103
104 // CheckCollision function: Checks if two rectangles are
    overlapping (used for collision detection).
105 bool CheckCollision(float x1A, float y1A, float x2A, float
    y2A,
106     float x1B, float y1B, float x2B, float y2B)
107 {
108     return !(x1A > x2B || x2A < x1B || y1A > y2B || y2A <
        y1B);
109 }
110
111 // RunLevel1Logic method: Handles the logic for running
    Level 1, including rendering warriors and updating the
    GUI.
112 void Level::RunLevel1Logic(float deltaTime, float gameTime)
113 {
114     int offsets[] = { 10, 110, 210, 310, 410, 510, 610,
        710, 810, 910 }; // Y-offsets for warriors.
115
116     renderer->SetDrawColor(Color(128, 128, 128, 255));
117     renderer->ClearScreen();
118
119     // Update and render warriors.
120     for (int i = 0; i < 10; i++)
121     {
122         m_warriorXPositions[i] += m_randSpeeds[i] *
                deltaTime;
123         renderer->RenderTexture(m_warriorSheets[i],
                m_warriorSheets[i]->Update(EN_AN_RUN,
                deltaTime),
124             Rect(m_warriorXPositions[i], offsets[i],
                    m_warriorXPositions[i] + spriteWidth *
                    scale, offsets[i] + spriteHeight * scale));
```

```
125     }
126
127     // Display FPS and game time on the GUI.
128     std::string fps = "FPS: " +
           std::to_string(Timing::Instance().GetFPS());
129     font->Write(renderer->GetRenderer(), fps.c_str(),
           SDL_Color{ 0, 0, 255 }, SDL_Point{ 100, 0 });
130
131     std::string time = "Game Time: " +
           std::to_string(static_cast<int>(gameTime));
132     font->Write(renderer->GetRenderer(), time.c_str(),
           SDL_Color{ 0, 0, 255 }, SDL_Point{ 250, 0 });
133
134     font->Write(renderer->GetRenderer(),
           m_autoSaveStatus.c_str(), SDL_Color{ 0, 0, 255 },
           SDL_Point{ 500, 0 });
135 }
136
137 // Level2TransitionTriggered method: Checks if any warrior
       has reached the viewport edge, triggering the
       transition to Level 2.
138 bool Level::Level2TransitionTriggered()
139 {
140     for (int i = 0; i < 10; i++)
141     {
142         if (m_warriorXPositions[i] >= viewportEdge -
               spriteWidth)
143         {
144             std::cout << "Warrior " << i << " was first!"
                   << std::endl;
145             m_warriorXPositions = std::vector<float>(10,
                   0.0f); // Reset warrior positions.
146             return true;
147         }
148     }
149
150     return false; // No warrior has crossed the viewport
           edge yet.
151 }
152
```

```
153 // RunLevel2Logic method: Handles the logic for running
       Level 2, including rendering warriors, rocks, and
       checking for collisions.
154 void Level::RunLevel2Logic(float deltaTime, float gameTime)
155 {
156     int offsets[] = { 10, 110, 210, 310, 410, 510, 610,
          710, 810, 910 };  // Y-offsets for warriors.
157     int offsetsRock[] = { 10, 110, 210, 310, 410, 510,
          610, 710, 810, 910 };  // X-offsets for rocks.
158
159     renderer->SetDrawColor(Color(0, 128, 0, 255));
160     renderer->ClearScreen();
161
162     // Update and render warriors.
163     for (int i = 0; i < 10; i++)
164     {
165         if (m_warriorIsAlive[i])
166         {
167             if (!m_warriorDeathState[i])
168             {
169                 m_warriorXPositions[i] += m_randSpeeds[i]
                      * deltaTime;
170                 renderer->RenderTexture(m_warriorSheets[i],
                      m_warriorSheets[i]->Update(EN_AN_RUN,
                      deltaTime),
171                     Rect(m_warriorXPositions[i],
                         offsets[i], m_warriorXPositions[i]
                         + spriteWidth * scale, offsets[i] +
                         spriteHeight * scale));
172             }
173             else
174             {
175                 if
                      (m_warriorSheets[i]->GetCurrentClip(EN_AN_DEATH)
                      == 36)
176                 {
177                     m_warriorIsAlive[i] = false; // Mark
                         warrior as dead once the death
                         animation completes.
178                 }
```


www.assignmentcompletion

```
                else
                {
                    renderer->RenderTexture(m_warriorSheets[i],
                       m_warriorSheets[i]->Update(EN_AN_DEATH,
                       deltaTime),
                        Rect(m_warriorXPositions[i],
                           offsets[i],
                           m_warriorXPositions[i] +
                           spriteWidth * scale, offsets[i]
                           + spriteHeight * scale));
                }
            }
        }
    }

    // Update and render rocks.
    for (int i = 0; i < 10; i++)
    {
        if (m_rockIsAlive[i])
        {
            m_rockYPositions[i] += m_randSpeeds[i] *
               deltaTime;
            renderer->RenderTexture(m_rockSheets[i],
               m_rockSheets[i]->Update(ROCK_FALL,
               deltaTime),
                Rect(offsetsRock[i], m_rockYPositions[i],
                   offsetsRock[i] + spriteHeightRock,
                   m_rockYPositions[i] + spriteWidthRock *
                   scaleRock));
        }
    }

    // Display FPS and game time on the GUI.
    std::string fps = "FPS: " +
       std::to_string(Timing::Instance().GetFPS());
    font->Write(renderer->GetRenderer(), fps.c_str(),
       SDL_Color{ 0, 0, 255 }, SDL_Point{ 100, 0 });

    std::string time = "Game Time: " +
       std::to_string(static_cast<int>(gameTime));
```

```
    font->Write(renderer->GetRenderer(), time.c_str(),
       SDL_Color{ 0, 0, 255 }, SDL_Point{ 250, 0 });

    font->Write(renderer->GetRenderer(),
       m_autoSaveStatus.c_str(), SDL_Color{ 0, 0, 255 },
       SDL_Point{ 500, 0 });

    // Collision detection between warriors and rocks.
    for (int i = 0; i < 10; i++)
    {
        if (m_warriorIsAlive[i])
        {
            for (int j = 0; j < 10; j++)
            {
                if (m_rockIsAlive[j])
                {
                    if
                       (CheckCollision(m_warriorXPositions[i],
                       offsets[i], m_warriorXPositions[i]
                       + spriteWidth, offsets[i] +
                       spriteHeight,
                        offsetsRock[j],
                           m_rockYPositions[j],
                           offsetsRock[j] +
                           spriteWidthRock,
                           m_rockYPositions[j] +
                           spriteHeightRock))
                    {
                        // Rock hits the warrior: mark
                           rock as "dead" and switch the
                           warrior's animation to death.
                        m_rockIsAlive[j] = false;
                        m_warriorDeathState[i] = true;
                    }
                }
            }
        }
    }
}
```

```
// Level2EndTriggered method: Checks if the level should
   end, either by a warrior reaching the edge or all
   warriors being dead.
bool Level::Level2EndTriggered()
{
    // Check if any warrior reached the viewport edge or
       if all warriors are dead.
    for (int i = 0; i < 10; i++)
    {
        if (m_warriorXPositions[i] >= viewportEdge -
           spriteWidth ||
           std::find(m_warriorIsAlive.begin(),
           m_warriorIsAlive.end(), true) ==
           m_warriorIsAlive.end())
        {
            return true;
        }
    }
    return false; // No end condition triggered yet.
}
```

Listing 10: C++ and SDL Program - Level.cpp

2.11 Level.h

```
#ifndef LEVEL_H
#define LEVEL_H

#include "StandardIncludes.h"
#include "SpriteSheet.h"
#include "Renderer.h"
#include "Timing.h"
#include "TTFont.h"

// The Level class represents a game level, handling game
   logic, assets, rendering, and timing for the level.
// It includes methods for saving and loading level
   states, as well as running specific logic for different
   levels (Level 1 and Level 2).
```

```
12
13 class Level : public Resource
14 {
15 public:
16     // Constructor: Initializes the Level with a renderer
       and font for rendering text and visuals.
17     Level(Renderer* renderer, TTFont* font);
18
19     // Overloaded constructor: Initializes the Level with
       two sprite sheets, a renderer, and a font.
20     Level(SpriteSheet* sheet, SpriteSheet* sheet1,
       Renderer* renderer, TTFont* font);
21
22     // Destructor: Cleans up resources when the Level is
       destroyed.
23     ~Level();
24
25     // Serializes (saves) the level's current state to an
       output stream (e.g., file).
26     void Serialize(std::ostream& _stream);
27
28     // Deserializes (loads) the level's state from an
       input stream (e.g., file).
29     void Deserialize(std::istream& _stream);
30
31     // Runs the game logic specific to Level 1. Takes
       deltaTime for smooth updates and gameTime for game
       progression.
32     void RunLevel1Logic(float deltaTime, float gameTime);
33
34     // Runs the game logic specific to Level 2. Similar to
       Level 1 logic but for Level 2 scenarios.
35     void RunLevel2Logic(float deltaTime, float gameTime);
36
37     // Sets the auto-save status message, used to indicate
       the status of automatic saving.
38     void SetAutoSaveStatus(const std::string& _status);
39
40     // Returns true if the transition from Level 1 to
       Level 2 has been triggered.
```


assignmentcompletion

```
41      bool Level2TransitionTriggered();
42
43      // Returns true if the end of Level 2 has been
           triggered.
44      bool Level2EndTriggered();
45
46      // Boolean flag to indicate if the level has been
           auto-saved.
47      bool autoSaved;
48
49  private:
50      // Variables for storing level data, used for saving
           and loading.
51      float rectX;                  // X position of the
           rectangle (or player).
52      float rectAsh;                // Ash-related data,
           likely a position or state.
53      float scale;                  // Scale for rendering.
54      int spriteWidth;              // Width of the sprite.
55      int spriteHeight;             // Height of the sprite.
56      float scaleRock;              // Scale for the rocks
           in the level.
57      int spriteWidthRock;          // Width of the rock
           sprite.
58      int spriteHeightRock;         // Height of the rock
           sprite.
59      int currentFrame;             // Current frame in the
           animation sequence.
60
61      std::vector<int> m_randSpeeds;             // Random
           speeds for entities in the level (e.g., warriors).
62      std::vector<bool> m_warriorIsAlive;        // Boolean
           flags to track which warriors are alive.
63      std::vector<bool> m_rockIsAlive;           // Boolean
           flags to track which rocks are "alive" (active).
64      std::vector<bool> m_warriorDeathState;     // Tracks
           the death state of each warrior.
65
66      std::vector<SpriteSheet*> m_warriorSheets; // Sprite
           sheets for warriors.
```


assignmentcompletion

```
    std::vector<SpriteSheet*> m_rockSheets;    // Sprite
       sheets for rocks.

    std::vector<float> m_warriorXPositions;    // X
       positions of the warriors.
    std::vector<float> m_rockYPositions;       // Y
       positions of the rocks.

    int viewportEdge;           // Edge of the viewport,
       used to determine rendering boundaries.
    bool isGenerated = false;   // Flag indicating if the
       level has been generated.

    // Private methods for internal operations.

    // Generates random speeds for entities like warriors.
    void GenerateRandomSpeeds();

    // Initializes the X positions of warriors in the
       level.
    void InitializeWarriorPositions(vector<float>
       _warriorXPositions);

    // Generates sprite sheets for the warriors.
    void GenerateWarriorSheets();

    // Generates sprite sheets for the rocks.
    void GenerateRockSheets();

    // Checks if two rectangular areas (Rect a and Rect b)
       are colliding (intersecting).
    bool IsColliding(const Rect& a, const Rect& b);

    // Timer for displaying the auto-save message.
    float autoSaveMsgTimer;

    // The auto-save status message.
    string m_autoSaveStatus;

```


www.assignmentcompletion

```
    // Dependencies for the level (provided by external
       classes).

    // Sprite sheet for the level's main elements (e.g.,
       warriors).
    SpriteSheet* sheet;

    // Sprite sheet for the rocks in the level.
    SpriteSheet* sheetRock;

    // Renderer used for drawing sprites and other
       graphical elements.
    Renderer* renderer;

    // Font used for rendering text (e.g., status
       messages).
    TTFont* font;
};

#endif
```

Listing 11: C++ and SDL Program - Level.h

2.12 ObjectPool.h

```
#ifndef OBJECTPOOL_H
#define OBJECTPOOL_H

#include "StandardIncludes.h"

// The ObjectPool class is a generic template for managing
   object allocation and reuse.
// It minimizes the cost of repeatedly allocating and
   deallocating objects by keeping a pool
// of reusable objects. This class can be used with any
   type `T` by leveraging the power of templates.

template<class T>
class ObjectPool
```

```
{
public:
    // Constructor: Initializes an empty object pool.
    ObjectPool() {}

    // Destructor: Cleans up the pool by deleting all objects created in the pool.
    ~ObjectPool() {
        // Loop through all objects stored in the pool and delete them to release memory.
        for (unsigned int count = 0; count < m_all.size(); count++)
        {
            std::cout << "Deleting pool object " << typeid(T).name() << std::endl;
            delete m_all[count];
        }
        // Clear both the list of all objects and the list of available objects.
        m_all.clear();
        m_available.clear();
    }

    // Retrieves an object from the pool. If no available object exists, a new one is created.
    // If an available object exists, it is reused (pulled from the available list).
    T* GetResource() {
        // If no objects are available in the pool, create a new one.
        if (m_available.empty())
        {
            std::cout << "Creating new pool object. " << typeid(T).name() << std::endl;
            T* resource = new T();
            // Store the newly created object in the list of all objects.
            m_all.push_back(resource);
            return resource;
        }
```


assignmentcompletion

```
        else
        {
            // If an available object exists, reuse it by
               taking it from the available list.
            std::cout << "Reusing existing pool object. "
               << typeid(T).name() << std::endl;
            T* resource = m_available[0];
            // Remove the object from the available list
               since it's in use.
            m_available.erase(m_available.begin());
            return resource;
        }
    }

    // Releases an object back to the pool. The object is
       reset and made available for reuse.
    void ReleaseResource(T* _object) {
        // Reset the object to its initial state (assumes
           the object has a Reset method).
        _object->Reset();
        // Add the object back to the available list for
           future reuse.
        m_available.push_back(_object);
    }

private:
    // Vector to hold the objects that are currently
       available for reuse.
    std::vector<T*> m_available;

    // Vector to hold all objects that have been created,
       whether in use or available.
    std::vector<T*> m_all;
};

#endif // OBJECTPOOL_H
```

Listing 12: C++ and SDL Program - ObjectPool.h

www.assignmentcompletion

2.13 Renderer.cpp

```cpp
#include "Renderer.h"

// Constructor: Initializes the Renderer object and sets default values for all member variables.
Renderer::Renderer()
{
    m_window = nullptr;  // Will hold the SDL window object.
    m_renderer = nullptr;  // Will hold the SDL renderer object.
    m_srcRect = {};  // Source rectangle for the texture (defines a sub-region of the texture to display, useful for sprite sheets).
    m_destRect = {};  // Destination rectangle for rendering on the screen.
    m_surface = nullptr;  // SDL surfaces are 2D images, later converted to textures.
    m_viewPort = {};  // The portion of the screen visible for rendering.
}

// Destructor: Calls Shutdown to clean up resources and avoid memory leaks.
Renderer::~Renderer()
{
    Shutdown();
}

// Initialize method: Sets up the SDL window and renderer with the specified resolution.
// This method also initializes all SDL systems and asserts if initialization fails.
void Renderer::Initialize(int _xResolution, int _yResolution)
{
    M_ASSERT((SDL_Init(SDL_INIT_EVERYTHING) >= 0), "Could not Init");  // Initialize SDL with all its systems (video, audio, etc.).
```

```
25      m_window = SDL_CreateWindow("SDL Window",
            SDL_WINDOWPOS_UNDEFINED, SDL_WINDOWPOS_UNDEFINED,
            _xResolution, _yResolution, SDL_WINDOW_SHOWN);  //
            Create the SDL window.
26      M_ASSERT(m_window != nullptr, "Failed to initialize
            SDL window.");  // Assert if window creation fails.
27      m_renderer =
            SDL_CreateRenderer(Renderer::Instance().GetWindow(),
            -1, SDL_RENDERER_PRESENTVSYNC);  // Create the SDL
            renderer and associate it with the window.
28      M_ASSERT(m_renderer != nullptr, "Failed to initialize
            SDL renderer.");  // Assert if renderer creation
            fails.
29  }
30
31  // Shutdown method: Cleans up resources (textures,
        renderer, and window) to avoid memory leaks.
32  void Renderer::Shutdown()
33  {
34      // Destroy all textures and free up GPU memory.
35      for (auto it = m_textures.begin(); it !=
            m_textures.end(); it++)
36      {
37          SDL_DestroyTexture(it->second);
38      }
39      m_textures.clear();  // Clear the texture map.
40
41      // Destroy the renderer.
42      if (m_renderer != nullptr)
43      {
44          SDL_DestroyRenderer(m_renderer);
45      }
46
47      // Destroy the window.
48      if (m_window != nullptr)
49      {
50          SDL_DestroyWindow(m_window);
51      }
52
53      // Shut down all SDL subsystems.
```


w.assignmentcompletion

```
    SDL_Quit();
}

// SetDrawColor method: Sets the draw color for the renderer (used for drawing primitives like rectangles).
void Renderer::SetDrawColor(Color _color)
{
    SDL_SetRenderDrawColor(m_renderer, _color.R, _color.G, _color.B, _color.A);
}

// ClearScreen method: Clears the screen with the current draw color.
void Renderer::ClearScreen()
{
    SDL_RenderClear(m_renderer);
}

// RenderPoint method: Renders a point at the specified position on the screen.
void Renderer::RenderPoint(Point _position)
{
    SDL_RenderDrawPoint(m_renderer, _position.X, _position.Y);
}

// RenderLine method: Renders a line between two points defined by the provided rectangle.
void Renderer::RenderLine(Rect _points)
{
    SDL_RenderDrawLine(m_renderer, _points.X1, _points.Y1, _points.X2, _points.Y2);
}

// RenderRectangle method: Renders the outline of a rectangle.
void Renderer::RenderRectangle(Rect _rect)
{
    m_destRect.x = _rect.X1;
    m_destRect.y = _rect.Y1;
```


assignmentcompletion

```
86      m_destRect.w = _rect.X2 - _rect.X1;
87      m_destRect.h = _rect.Y2 - _rect.Y1;
88      SDL_RenderDrawRect(m_renderer, &m_destRect);
89  }
90
91  // RenderFillRectangle method: Renders a filled rectangle.
92  void Renderer::RenderFillRectangle(Rect _rect)
93  {
94      m_destRect.x = _rect.X1;
95      m_destRect.y = _rect.Y1;
96      m_destRect.w = _rect.X2 - _rect.X1;
97      m_destRect.h = _rect.Y2 - _rect.Y1;
98      SDL_RenderFillRect(m_renderer, &m_destRect);
99  }
100
101 // GetSDLTexture method: Retrieves or creates an SDL
        texture for the given Texture object.
102 SDL_Texture* Renderer::GetSDLTexture(Texture* _texture)
103 {
104     Asset* asset = _texture->GetData();  // Retrieve the
            asset associated with the texture.
105     string guid = asset->GetGUID();  // Get the asset's
            GUID (unique identifier).
106
107     // If the texture has already been created, return it
            from the texture map.
108     if (m_textures.count(guid) != 0)
109     {
110         return m_textures[guid];
111     }
112
113     // If the texture does not exist, create it from the
            raw image data.
114     ImageInfo* ii = _texture->GetImageInfo();
115     m_surface = SDL_CreateRGBSurfaceFrom(
116         asset->GetData() +
                _texture->GetImageInfo()->DataOffset,
117         ii->Width, ii->Height, ii->BitsPerPixel,
118         ii->Width * ii->BitsPerPixel / 8,
119         0x00FF0000, 0x0000FF00, 0x000000FF, 0xFF000000
```


www.assignmentcompletion

```
120     );
121     SDL_Texture* texture =
           SDL_CreateTextureFromSurface(m_renderer, m_surface);
122     SDL_FreeSurface(m_surface);  // Free the surface after
           creating the texture.
123     m_surface = nullptr;
124     m_textures[guid] = texture;  // Store the texture in
           the texture map.
125
126     return texture;
127 }
128
129 // RenderTexture method: Renders a texture at the
       specified point on the screen.
130 void Renderer::RenderTexture(Texture* _texture, Point
       _point)
131 {
132     m_destRect.x = _point.X;
133     m_destRect.y = _point.Y;
134     m_destRect.w = _texture->GetImageInfo()->Width;
135     m_destRect.h = _texture->GetImageInfo()->Height;
136     M_ASSERT((SDL_RenderCopyEx(m_renderer,
           GetSDLTexture(_texture), NULL, &m_destRect, 0,
           NULL, SDL_FLIP_VERTICAL) >= 0), "Could not render
           texture.");
137 }
138
139 // GetWindowSize method: Returns the size of the window
       managed by the renderer.
140 Point Renderer::GetWindowSize()
141 {
142     int w, h;
143     SDL_GetWindowSize(m_window, &w, &h);
144     return Point(w, h);
145 }
146
147 // SetViewport method: Sets the viewport for rendering
       (defines the portion of the screen to render to).
148 void Renderer::SetViewport(Rect _viewport)
149 {
```

```
150     m_viewPort.x = _viewport.X1;
151     m_viewPort.y = _viewport.Y1;
152     m_viewPort.w = _viewport.X2 - _viewport.X1;
153     m_viewPort.h = _viewport.Y2 - _viewport.Y1;
154     SDL_RenderSetViewport(m_renderer, &m_viewPort);
155 }
156 
157 // RenderTexture method (overloaded): Renders a texture in
       a specific location defined by a rectangle.
158 void Renderer::RenderTexture(Texture* _texture, Rect _rect)
159 {
160     m_destRect.x = _rect.X1;
161     m_destRect.y = _rect.Y1;
162     m_destRect.w = _rect.X2 - _rect.X1;
163     m_destRect.h = _rect.Y2 - _rect.Y1;
164     M_ASSERT((SDL_RenderCopyEx(m_renderer,
           GetSDLTexture(_texture), NULL, &m_destRect, 0,
           NULL, SDL_FLIP_VERTICAL) >= 0), "Could not render
           texture.");
165 }
166 
167 // RenderTexture method (overloaded): Renders a specific
       part of a texture to a specified destination.
168 void Renderer::RenderTexture(Texture* _texture, Rect
       _srcRect, Rect _destRect)
169 {
170     // Define the destination rectangle's size and
           position.
171     m_destRect.x = _destRect.X1;
172     m_destRect.y = _destRect.Y1;
173     m_destRect.w = _destRect.X2 - _destRect.X1;
174     m_destRect.h = _destRect.Y2 - _destRect.Y1;
175 
176     // Define the source rectangle (the part of the
           texture to render).
177     m_srcRect.x = _srcRect.X1;
178     m_srcRect.y = _texture->GetImageInfo()->Height -
           _srcRect.Y2;  // Adjust for SDL's inverted Y-axis.
179     m_srcRect.w = _srcRect.X2 - _srcRect.X1;
180     m_srcRect.h = _srcRect.Y2 - _srcRect.Y1;
```

```
181
182     // Copy the texture from the source rectangle to the
            destination rectangle.
183     M_ASSERT((SDL_RenderCopyEx(m_renderer,
            GetSDLTexture(_texture), &m_srcRect, &m_destRect,
            0, NULL, SDL_FLIP_VERTICAL) >= 0), "Could not
            render texture");
184 }
```

Listing 13: C++ and SDL Program - Renderer.cpp

2.14 Renderer.h

```
1  #ifndef RENDERER_H
2  #define RENDERER_H
3
4  #include "StandardIncludes.h"
5  #include "Texture.h"
6
7  class Asset;
8
9  // The Renderer class is responsible for handling all
       graphical rendering operations in the game.
10 // It uses SDL (Simple DirectMedia Layer) to manage window
       creation, drawing operations, and texture rendering.
11 // The Renderer follows the Singleton pattern to ensure
       there is only one instance controlling rendering.
12
13 class Renderer : public Singleton<Renderer>
14 {
15 public:
16     // Constructor: Initializes the Renderer object.
17     Renderer();
18
19     // Destructor: Cleans up resources and shuts down SDL
           rendering when the Renderer is destroyed.
20     virtual ~Renderer();
21
22     // Accessors
```


www.assignmentcompletion

```
23
24      // Returns the SDL_Window object associated with the
            Renderer.
25      SDL_Window* GetWindow() { return m_window; }
26
27      // Returns the SDL_Renderer object, which is
            responsible for rendering textures and shapes to
            the screen.
28      SDL_Renderer* GetRenderer() { return m_renderer; }
29
30      // Returns an SDL_Texture pointer associated with the
            given Texture object.
31      // Converts the custom Texture object to an
            SDL-compatible texture for rendering.
32      SDL_Texture* GetSDLTexture(Texture* _texture);
33
34      // Methods
35
36      // Initializes the Renderer by creating a window and
            setting up the SDL rendering context.
37      // The resolution is defined by _xResolution and
            _yResolution (width and height).
38      void Initialize(int _xResolution, int _yResolution);
39
40      // Returns the current window size as a Point object,
            containing width (X) and height (Y).
41      Point GetWindowSize();
42
43      // Sets the drawing color for subsequent render
            operations (e.g., for drawing shapes).
44      void SetDrawColor(Color color);
45
46      // Clears the screen by filling it with the current
            draw color.
47      void ClearScreen();
48
49      // Sets the rendering viewport to the specified
            rectangular area.
50      // Only this area will be affected by rendering
            operations.
```

```
51      void SetViewport(Rect _viewport);
52
53      // Renders a single point at the specified position.
54      void RenderPoint(Point _position);
55
56      // Renders a line between two points defined by a Rect
            (start point and end point).
57      void RenderLine(Rect _points);
58
59      // Renders a rectangle outline using the specified
            Rect.
60      void RenderRectangle(Rect _rect);
61
62      // Renders a filled rectangle using the specified Rect.
63      void RenderFillRectangle(Rect _rect);
64
65      // Renders a texture at the specified point on the
            screen.
66      void RenderTexture(Texture* _texture, Point _point);
67
68      // Renders a texture within the specified destination
            rectangle.
69      void RenderTexture(Texture* _texture, Rect _rect);
70
71      // Renders a portion of a texture (defined by
            _srcRect) onto a destination area (defined by
            _destRect).
72      void RenderTexture(Texture* _texture, Rect _srcRect,
            Rect _destRect);
73
74      // Shuts down the Renderer by destroying the window,
            rendering context, and freeing all textures.
75      void Shutdown();
76
77  private:
78      // SDL window handle for managing the game's window.
79      SDL_Window* m_window;
80
81      // SDL renderer responsible for rendering textures,
            shapes, and other graphical elements.
```


www.assignmentcompletion

```
    SDL_Renderer* m_renderer;

    // SDL source and destination rectangles for rendering
       textures.
    SDL_Rect m_srcRect;   // Defines the source area of
       the texture to render.
    SDL_Rect m_destRect;  // Defines the destination area
       where the texture will be drawn.

    // SDL surface used for temporary storage of images
       before converting to textures.
    SDL_Surface* m_surface;

    // SDL rectangle defining the current viewport
       (rendering area).
    SDL_Rect m_viewPort;

    // A map that holds textures, keyed by their name
       (string), for easy access and reuse.
    map<string, SDL_Texture*> m_textures;
};

#endif //RENDERER_H
```

Listing 14: C++ and SDL Program - Renderer.h

2.15 Resource.cpp

```
#include "Resource.h"

// Initialize the static ObjectPool for Resource instances.
ObjectPool<Resource>* Resource::Pool;

// Constructor: Initializes the Resource object with
   default values.
Resource::Resource()
{
    m_val1 = 0;             // Initialize m_val1 to 0.
    m_val2 = 0.1;           // Initialize m_val2 to 0.1.
```

```
    m_val3 = {};              // Initialize m_val3 to an
       empty char.
    m_subResource = nullptr; // Initialize the pointer to
       the subResource to null.
    m_asset = nullptr;       // Initialize the pointer to
       the associated asset to null.
}

// Destructor: Default destructor for Resource class. No
   specific cleanup required here.
Resource::~Resource()
{
}

// AssignNonDefaultValues method: Sets non-default values
   for testing or specific use cases.
// This method sets values for the resource and
   sub-resources to demonstrate serialization.
void Resource::AssignNonDefaultValues()
{
    // This method can be used to assign specific values
       to the resource and sub-resource if needed.
    // It's currently not used, but you can implement
       logic here as required for testing or debugging.
}

// ToString method: Outputs the values of the Resource
   object for debugging purposes.
void Resource::ToString()
{
    // The method prints a basic message. This can be
       expanded to show the actual values of the resource.
    cout << "BASE RESOURCE" << endl;
}

// SerializeAsset method: Serializes the asset's GUID into
   the output stream.
void Resource::SerializeAsset(std::ostream& _stream,
   Asset* _asset)
{
```


assignmentcompletion

```
    byte guidLength = 0;

    // If the asset exists, get the length of its GUID.
    if (_asset != nullptr)
    {
        guidLength = _asset->GetGUID().length();
    }

    // Write the length of the GUID to the stream.
    _stream.write(reinterpret_cast<char*>(&guidLength),
       sizeof(guidLength));

    // If the GUID has a length, write the actual GUID to
       the stream.
    if (guidLength > 0)
    {
        _stream.write(_asset->GetGUID().c_str(),
           guidLength);
    }
}

// DeserializeAsset method: Deserializes the asset's GUID
   from the input stream and loads the corresponding asset.
void Resource::DeserializeAsset(std::istream& _stream,
   Asset*& _asset)
{
    byte guidLength = 0;

    // Read the length of the GUID from the stream.
    _stream.read(reinterpret_cast<char*>(&guidLength),
       sizeof(byte));

    // If the GUID has a length, read the GUID from the
       stream and load the asset.
    if (guidLength > 0)
    {
        char guid[255];  // Buffer to hold the GUID.
        _stream.read(guid, guidLength);  // Read the GUID.
        guid[guidLength] = 0;  // Null-terminate the
           string.
```


assignmentcompletion

```
71
72          // Load the asset using the AssetController and
                the deserialized GUID.
73          _asset =
                AssetController::Instance().GetAsset(std::string(guid));
74      }
75  }
76
77  // Serialize method: Serializes the resource's data,
        including its sub-resources and assets, into the output
        stream.
78  void Resource::Serialize(std::ostream& _stream)
79  {
80      // You can implement this method to serialize member
            variables and sub-resources.
81      // Here is an example of serializing the asset:
82      SerializeAsset(_stream, m_asset);
83  }
84
85  // Deserialize method: Deserializes the resource's data,
        including its sub-resources and assets, from the input
        stream.
86  void Resource::Deserialize(std::istream& _stream)
87  {
88      // You can implement this method to deserialize member
            variables and sub-resources.
89      // Here is an example of deserializing the asset:
90      DeserializeAsset(_stream, m_asset);
91  }
```

Listing 15: C++ and SDL Program - Resource.cpp

2.16 Resource.h

```
1  #ifndef RESOURCE_H
2  #define RESOURCE_H
3
4  #include "Serializable.h"
5  #include "ObjectPool.h"
```


assignmentcompletion

```cpp
#include "AssetController.h"

// The Resource class is a base class for any game resources that need to be serialized (saved/loaded).
// It extends the Serializable interface and provides methods for handling resource serialization
// and deserialization, as well as assigning non-default values. It also uses an object pool
// for memory management of Resource instances.

class Resource : public Serializable
{
public:
    // Constructor: Initializes a new Resource object.
    Resource();

    // Destructor: Cleans up the resource and frees associated memory.
    virtual ~Resource();

    // Serializes the resource by writing its data to an output stream (e.g., saving to a file).
    virtual void Serialize(std::ostream& _stream);

    // Deserializes the resource by reading its data from an input stream (e.g., loading from a file).
    virtual void Deserialize(std::istream& _stream);

    // Assigns values to the resource that are not the default values (for setting up non-default states).
    virtual void AssignNonDefaultValues();

    // Converts the resource into a string representation (for debugging or logging purposes).
    virtual void ToString();

    // Static object pool for managing Resource instances, optimizing memory allocation.
    static ObjectPool<Resource>* Pool;

```


assignmentcompletion

```
protected:
    // Template method for serializing a pointer to another object.
    // If the pointer is not null, it serializes the object. If the pointer is null, it writes a flag indicating that.
    template<class T>
    void SerializePointer(std::ostream& _stream, T* _pointer) {
        byte exists = 1;
        if (_pointer != nullptr)
        {
            _stream.write(reinterpret_cast<char*>(&exists), sizeof(byte));
            _pointer->Serialize(_stream);
        }
        else
        {
            exists = 0;
            _stream.write(reinterpret_cast<char*>(&exists), sizeof(byte));
        }
    }

    // Template method for deserializing a pointer from an input stream.
    // It reads the existence flag and deserializes the object if the flag indicates it exists.
    template<class T>
    void DeserializePointer(std::istream& _stream, T* _pointer) {
        byte exists = 0;
        _stream.read(reinterpret_cast<char*>(&exists), sizeof(exists));
        if (exists == 1)
        {
            _pointer = T::Pool->GetResource();
            _pointer->Deserialize(_stream);
        }
    }
```

```
67
68      // Serializes an Asset object by writing it to the
            output stream.
69      void SerializeAsset(std::ostream& _stream, Asset*
            _asset);
70
71      // Deserializes an Asset object from the input stream
            and assigns it to the provided pointer reference.
72      void DeserializeAsset(std::istream& _stream, Asset*&
            _asset);
73
74  private:
75      // Private members storing some values that would be
            serialized and deserialized.
76      int m_val1;         // Example integer value.
77      double m_val2;      // Example double value.
78      char m_val3;        // Example character value.
79
80      // Pointer to a sub-resource, allowing nested
            resources to be serialized/deserialized.
81      Resource* m_subResource;
82
83      // Pointer to an asset associated with the resource.
84      Asset* m_asset;
85  };
86
87  #endif //RESOURCE_H
```

Listing 16: C++ and SDL Program - Resource.h

2.17 SDLLevels.cpp

```
1   #include "GameController.h"
2
3   int main()
4   {
5       // Retrieve the singleton instance of GameController
            and run the game.
6       // This initializes and starts the main game loop.
```

```
    GameController::Instance().RunGame();

    // Return 0 to indicate successful execution of the
       program.
    return 0;
}
```

Listing 17: C++ and SDL Program - SDLLevels.cpp

2.18 Serializable.h

```
#ifndef SERIALIZABLE_H
#define SERIALIZABLE_H

#include "StandardIncludes.h"

// The Serializable class is an abstract base class that
   defines the interface for any object
// that needs to support serialization and
   deserialization. Classes inheriting from Serializable
// must implement methods to save their data to an output
   stream and load their data from an input stream.

class Serializable
{
public:
    // Constructor: Initializes a Serializable object.
    Serializable() {}

    // Virtual destructor: Ensures that derived classes
       clean up properly when destroyed.
    virtual ~Serializable() {}

    // Pure virtual function for serializing the object's
       data to an output stream (e.g., saving to a file).
    // This must be implemented by any class that inherits
       from Serializable.
    virtual void Serialize(ostream& _stream) = 0;

```

www.assignmentcompletion

```
    // Pure virtual function for deserializing the
       object's data from an input stream (e.g., loading
       from a file).
    // This must be implemented by any class that inherits
       from Serializable.
    virtual void Deserialize(istream& _stream) = 0;
};

#endif //SERIALIZABLE_H
```

Listing 18: C++ and SDL Program - Serializable.h

2.19 Singleton.h

```
#ifndef SINGLETON_H
#define SINGLETON_H

// The Singleton class is a template that implements the
   Singleton design pattern,
// ensuring that only one instance of a class exists
   throughout the application.
// It provides global access to that single instance,
   which is lazily initialized
// when first accessed.

template <typename T>
class Singleton
{
public:
    // Returns the single instance of the class `T`.
    // The instance is created the first time this
       function is called (lazy initialization).
    static T& Instance()
    {
        static T instance;  // Static local variable that
           holds the single instance of the class.
        return instance;
    }

```

```
protected:
    // Protected constructor ensures that only derived
       classes can instantiate Singleton.
    Singleton() {}

    // Virtual destructor to allow proper cleanup in
       derived classes.
    virtual ~Singleton() {}

public:
    // Delete copy constructor to prevent copying of the
       Singleton instance.
    Singleton(Singleton const&) = delete;

    // Delete assignment operator to prevent assignment of
       the Singleton instance.
    Singleton& operator=(Singleton const&) = delete;
};

#endif // SINGLETON_H
```

Listing 19: C++ and SDL Program - Singleton.h

2.20 SoundEffect.cpp

```
#include "SoundEffect.h"

// Initialize the static ObjectPool for SoundEffect
   instances.
// This pool is used to manage instances of the
   SoundEffect class efficiently.
ObjectPool<SoundEffect>* SoundEffect::Pool = nullptr;

// Constructor: Initializes the SoundEffect object with
   default values.
SoundEffect::SoundEffect()
{
    m_effect = nullptr;  // Initialize the m_effect
       pointer to null (no sound effect loaded).
```

```
}

// Destructor: Default destructor for the SoundEffect class.
// No specific cleanup required as assets are managed externally.
SoundEffect::~SoundEffect()
{
}

// Serialize method: Serializes the SoundEffect's data, including the sound asset, into the output stream.
void SoundEffect::Serialize(std::ostream& _stream)
{
    // Serialize the sound effect asset (m_effect) into the stream.
    SerializeAsset(_stream, m_effect);

    // Serialize the base class (Resource) data into the stream.
    Resource::Serialize(_stream);
}

// Deserialize method: Deserializes the SoundEffect's data, including the sound asset, from the input stream.
void SoundEffect::Deserialize(std::istream& _stream)
{
    // Deserialize the sound effect asset (m_effect) from the stream.
    DeserializeAsset(_stream, m_effect);

    // Deserialize the base class (Resource) data from the stream.
    Resource::Deserialize(_stream);
}

// ToString method: Outputs information about the SoundEffect object to the console.
// This is useful for debugging to see the state of the sound effect.
```


assignmentcompletion

```cpp
void SoundEffect::ToString()
{
    cout << "SOUND EFFECT" << endl;  // Output the label for the sound effect.

    // If the sound effect asset exists, output its details.
    if (m_effect != nullptr)
    {
        m_effect->ToString();  // Call the asset's ToString method to print its details.
    }

    // Call the base class's ToString method to print Resource details.
    Resource::ToString();
}
```

Listing 20: C++ and SDL Program - SoundEffect.cpp

2.21 SoundEffect.h

```cpp
#ifndef SOUND_EFFECT_H
#define SOUND_EFFECT_H

#include "Resource.h"
#include "Asset.h"

// The SoundEffect class represents a sound effect resource in the game, which can be serialized and deserialized.
// It inherits from the Resource class, allowing it to be managed by the game's resource management system.
// The sound effect is linked to an Asset that contains the actual sound data.

class SoundEffect : public Resource
{
public:
```


assignmentcompletion

```
    // Constructor: Initializes a new SoundEffect object.
    SoundEffect();

    // Destructor: Cleans up the SoundEffect object and
       any associated resources.
    virtual ~SoundEffect();

    // Serializes the SoundEffect object by saving its
       state to the provided output stream (e.g., saving
       to a file).
    void Serialize(std::ostream& _stream) override;

    // Deserializes the SoundEffect object by loading its
       state from the provided input stream (e.g., loading
       from a file).
    void Deserialize(std::istream& _stream) override;

    // Converts the SoundEffect's details into a string
       for debugging or logging purposes.
    void ToString() override;

    // Object pool for managing SoundEffect instances,
       allowing for efficient memory usage through reuse.
    static ObjectPool<SoundEffect>* Pool;

private:
    // Pointer to an Asset that holds the actual sound
       data for the sound effect.
    Asset* m_effect;
};

#endif // SOUND_EFFECT_H
```

Listing 21: C++ and SDL Program - SoundEffect.h

2.22 SpriteAnim.cpp

```
#include "SpriteAnim.h"

```

```
// Initialize the static ObjectPool for SpriteAnim instances.
ObjectPool<SpriteAnim>* SpriteAnim::Pool;

// Constructor: Initializes the SpriteAnim object by clearing the memory.
SpriteAnim::SpriteAnim()
{
    ClearMemory();  // Resets all member variables to their default values.
}

// Destructor: Default destructor for the SpriteAnim class.
SpriteAnim::~SpriteAnim()
{
}

// Create method: Initializes the animation parameters (frame details).
// _clipStart: The starting frame of the animation.
// _clipCount: The number of frames in the animation.
// _clipSpeed: The speed of the animation.
void SpriteAnim::Create(short _clipStart, short _clipCount, float _clipSpeed)
{
    m_clipStart = _clipStart;           // Set the starting frame of the animation.
    m_clipCount = _clipCount;           // Set the number of frames in the animation.
    m_clipSpeed = _clipSpeed;           // Set the animation speed.
    m_clipCurrent = _clipStart;         // Set the current frame to the start of the animation.
    m_clipEnd = m_clipStart + m_clipCount;  // Calculate the ending frame of the animation.
}

// ClearMemory method: Resets the animation parameters to their default values.
```

```cpp
// This is used to ensure all animation data is properly initialized.
void SpriteAnim::ClearMemory()
{
    m_clipStart = 0;     // Reset starting frame.
    m_clipCount = 0;     // Reset frame count.
    m_clipCurrent = 0;   // Reset current frame to 0.
}

// Update method: Updates the animation frame based on the delta time and animation speed.
// _deltaTime: The time passed since the last frame update, used for consistent frame timing.
void SpriteAnim::Update(float _deltaTime)
{
    // Increment the current frame based on the animation speed and delta time.
    m_clipCurrent += m_clipSpeed * _deltaTime;

    // If the current frame exceeds the end frame, reset it to the starting frame.
    if (m_clipCurrent > m_clipEnd)
    {
        m_clipCurrent = m_clipStart;
    }
}

// Serialize method: Writes the state of the SpriteAnim object to the output stream.
// This includes the clip start, count, and speed, along with any base class (Resource) data.
void SpriteAnim::Serialize(std::ostream& _stream)
{
    // Write the animation parameters to the stream.
    _stream.write(reinterpret_cast<char*>(&m_clipStart), sizeof(m_clipStart));
    _stream.write(reinterpret_cast<char*>(&m_clipCount), sizeof(m_clipCount));
    _stream.write(reinterpret_cast<char*>(&m_clipSpeed), sizeof(m_clipSpeed));
```


www.assignmentcompletion

```
61
62      // Serialize the base class (Resource) data.
63      Resource::Serialize(_stream);
64  }
65
66  // Deserialize method: Reads the state of the SpriteAnim
        object from the input stream.
67  // This includes reading the clip start, count, speed, and
        base class data.
68  void SpriteAnim::Deserialize(std::istream& _stream)
69  {
70      // Read the animation parameters from the stream.
71      _stream.read(reinterpret_cast<char*>(&m_clipStart),
            sizeof(m_clipStart));
72      _stream.read(reinterpret_cast<char*>(&m_clipCount),
            sizeof(m_clipCount));
73      _stream.read(reinterpret_cast<char*>(&m_clipSpeed),
            sizeof(m_clipSpeed));
74
75      // Recalculate the ending frame based on the start and
            count.
76      m_clipEnd = m_clipStart + m_clipCount;
77
78      // Deserialize the base class (Resource) data.
79      Resource::Deserialize(_stream);
80  }
81
82  // ToString method: Outputs information about the
        SpriteAnim object for debugging.
83  // This includes details about the animation's start frame
        and frame count.
84  void SpriteAnim::ToString()
85  {
86      // Print the animation parameters.
87      cout << "SPRITE ANIMATION: ";
88      cout << "Clip Start: " << m_clipStart;
89      cout << "Clip Count: " << m_clipCount << endl;
90
91      // Call the base class's ToString method to print
            additional resource information.
```


assignmentcompletion

```
    Resource::ToString();
}
```

Listing 22: C++ and SDL Program - SpriteAnim.cpp

2.23 SpriteAnim.h

```
#ifndef SPRITEANIM_H
#define SPRITEANIM_H

#include "Resource.h"

// The SpriteAnim class represents an animation sequence for sprites, allowing for smooth transitions
// between different frames (clips). It inherits from the Resource class, meaning it can be serialized
// and deserialized for saving/loading purposes. The class uses an object pool to manage memory efficiently.

class SpriteAnim : public Resource
{
public:
    // Constructor: Initializes the SpriteAnim object.
    SpriteAnim();

    // Destructor: Cleans up resources used by the SpriteAnim object.
    virtual ~SpriteAnim();

    // Accessors

    // Returns the current clip index in the animation sequence.
    short GetClipCurrent() { return (short)m_clipCurrent; }

    // Methods

    // Serializes the animation data to an output stream (e.g., saving to a file).
```

```
27      virtual void Serialize(std::ostream& _stream) override;
28
29      // Deserializes the animation data from an input
           stream (e.g., loading from a file).
30      virtual void Deserialize(std::istream& _stream)
           override;
31
32      // Converts the animation details into a string for
           debugging or logging purposes.
33      virtual void ToString() override;
34
35      // Clears any memory or resets the animation to its
           default state.
36      void ClearMemory();
37
38      // Creates a new animation sequence starting at
           `_clipStart`, with `_clipCount` frames,
39      // and the animation runs at a speed determined by
           `_clipSpeed`.
40      void Create(short _clipStart, short _clipCount, float
           _clipSpeed);
41
42      // Updates the current animation frame based on the
           elapsed time (`_deltaTime`).
43      void Update(float _deltaTime);
44
45      // Object pool for managing SpriteAnim instances,
           optimizing memory usage by reusing objects.
46      static ObjectPool<SpriteAnim>* Pool;
47
48  private:
49      // Members
50
51      // Starting clip index of the animation sequence.
52      short m_clipStart;
53
54      // Number of clips (frames) in the animation sequence.
55      short m_clipCount;
56
```

www.assignmentcompletion

```
57      // Current clip index in the animation sequence, which
           changes as the animation plays.
58      float m_clipCurrent;
59
60      // Speed at which the animation progresses through its
           frames.
61      float m_clipSpeed;
62
63      // Ending clip index, calculated as m_clipStart +
           m_clipCount - 1.
64      short m_clipEnd;
65  };
66
67  #endif // SPRITEANIM_H
```

Listing 23: C++ and SDL Program - SpriteAnim.h

2.24 SpriteSheet.cpp

```
1   #include "SpriteSheet.h"
2
3   // Initialize the static ObjectPool for SpriteSheet
       instances.
4   // This pool allows for efficient memory reuse and
       management of SpriteSheet objects.
5   ObjectPool<SpriteSheet>* SpriteSheet::Pool;
6
7   // Constructor: Initializes the SpriteSheet object by
       setting default values for rows, columns, and clip
       sizes.
8   SpriteSheet::SpriteSheet()
9   {
10      m_rows = 0;         // Initialize the number of rows in
           the sprite sheet.
11      m_columns = 0;      // Initialize the number of columns
           in the sprite sheet.
12      m_clipSizeX = 0;    // Initialize the width of each
           clip (sprite frame).
13      m_clipSizeY = 0;    // Initialize the height of each
           clip (sprite frame).
```


www.assignmentcompletion.com

```
}

// Destructor: Clears the animations map to free up resources.
SpriteSheet::~SpriteSheet()
{
    m_animations.clear();  // Clear the map containing animations to avoid memory leaks.
}

// SetSize method: Configures the size of the sprite sheet.
// _rows: Number of rows in the sprite sheet.
// _columns: Number of columns in the sprite sheet.
// _clipSizeX: Width of each individual sprite frame.
// _clipSizeY: Height of each individual sprite frame.
void SpriteSheet::SetSize(byte _rows, byte _columns, byte _clipSizeX, byte _clipSizeY)
{
    m_rows = _rows;         // Set the number of rows.
    m_columns = _columns;   // Set the number of columns.
    m_clipSizeX = _clipSizeX;  // Set the width of each frame.
    m_clipSizeY = _clipSizeY;  // Set the height of each frame.
}

// AddAnimation method: Adds a new animation to the sprite sheet.
// _name: The name of the animation (e.g., idle, run).
// _clipStart: The index of the first frame in the animation.
// _clipCount: The number of frames in the animation.
// _clipSpeed: The speed of the animation.
void SpriteSheet::AddAnimation(AnimationNames _name, short _clipStart, short _clipCount, float _clipSpeed)
{
    SpriteAnim* anim = SpriteAnim::Pool->GetResource();
        // Get a new SpriteAnim object from the pool.
    anim->Create(_clipStart, _clipCount, _clipSpeed);
        // Initialize the animation.
```


www.assignmentcompletion

```
44      m_animations[_name] = anim;  // Add the animation to
            the map.
45  }
46
47  // Update method: Updates the current frame of the
        specified animation based on the delta time.
48  // _name: The name of the animation to update.
49  // _deltaTime: The time that has passed since the last
        frame, used to update the animation smoothly.
50  Rect SpriteSheet::Update(AnimationNames _name, float
        _deltaTime)
51  {
52      short s = m_animations[_name]->GetClipCurrent();  //
            Get the current frame index of the animation.
53      short posX = s % m_columns * m_clipSizeX;         //
            Calculate the X position of the frame in the sprite
            sheet.
54      short posY = s / m_columns * m_clipSizeY;         //
            Calculate the Y position of the frame in the sprite
            sheet.
55
56      Rect r = Rect(posX, posY, posX + m_clipSizeX, posY +
            m_clipSizeY);  // Create a rectangle representing
            the frame.
57
58      m_animations[_name]->Update(_deltaTime);  // Update
            the animation to the next frame if necessary.
59
60      return r;  // Return the rectangle for rendering the
            current frame.
61  }
62
63  // GetCurrentClip method: Retrieves the current frame
        index of the specified animation.
64  // _name: The name of the animation.
65  // Returns the current frame index.
66  int SpriteSheet::GetCurrentClip(AnimationNames _name)
67  {
68      if (m_animations.count(_name) < 0)
69      {
```

```
70          return 0;  // If the animation does not exist,
               return 0 as the default clip index.
71      }
72      return m_animations[_name]->GetClipCurrent();  //
           Return the current frame index.
73  }
74
75  // Serialize method: Serializes the state of the
       SpriteSheet object to the output stream.
76  // This includes saving the size, animations, and
       associated texture.
77  void SpriteSheet::Serialize(std::ostream& _stream)
78  {
79      // Serialize the sprite sheet size information.
80      _stream.write(reinterpret_cast<char*>(&m_rows),
           sizeof(m_rows));
81      _stream.write(reinterpret_cast<char*>(&m_columns),
           sizeof(m_columns));
82      _stream.write(reinterpret_cast<char*>(&m_clipSizeX),
           sizeof(m_clipSizeX));
83      _stream.write(reinterpret_cast<char*>(&m_clipSizeY),
           sizeof(m_clipSizeY));
84
85      // Serialize the number of animations.
86      int count = m_animations.size();
87      _stream.write(reinterpret_cast<char*>(&count),
           sizeof(count));
88
89      // Serialize each animation.
90      for (auto& a : m_animations) {
91          AnimationNames index = a.first;  // Get the
               animation name.
92          _stream.write(reinterpret_cast<char*>(&index),
               sizeof(index));  // Write the animation name.
93          a.second->Serialize(_stream);  // Serialize the
               animation itself.
94      }
95
96      // Serialize the base Texture class data.
97      Texture::Serialize(_stream);
```


v.assignmentcompletion

```
}

// Deserialize method: Restores the state of the SpriteSheet object from the input stream.
// This includes loading the size, animations, and associated texture.
void SpriteSheet::Deserialize(std::istream& _stream)
{
    // Deserialize the sprite sheet size information.
    _stream.read(reinterpret_cast<char*>(&m_rows), sizeof(m_rows));
    _stream.read(reinterpret_cast<char*>(&m_columns), sizeof(m_columns));
    _stream.read(reinterpret_cast<char*>(&m_clipSizeX), sizeof(m_clipSizeX));
    _stream.read(reinterpret_cast<char*>(&m_clipSizeY), sizeof(m_clipSizeY));

    // Deserialize the number of animations.
    int count;
    _stream.read(reinterpret_cast<char*>(&count), sizeof(count));

    // Deserialize each animation.
    for (int c = 0; c < count; c++) {
        AnimationNames index;
        _stream.read(reinterpret_cast<char*>(&index), sizeof(index));  // Read the animation name.
        SpriteAnim* anim = SpriteAnim::Pool->GetResource();  // Get a new SpriteAnim object from the pool.
        anim->Deserialize(_stream);  // Deserialize the animation itself.
        m_animations[index] = anim;  // Add the animation to the map.
    }

    // Deserialize the base Texture class data.
    Texture::Deserialize(_stream);
}
```


assignmentcompletion

```
126
127 // ToString method: Outputs information about the
       SpriteSheet object for debugging.
128 // This includes details about the rows, columns, and clip
       sizes.
129 void SpriteSheet::ToString()
130 {
131     // Print the sprite sheet properties.
132     cout << "SPRITE SHEET:";
133     cout << "  Rows: " << (int)m_rows;
134     cout << "  Columns: " << (int)m_columns;
135     cout << "  ClipSizeX: " << (int)m_clipSizeX;
136     cout << "  ClipSizeY: " << (int)m_clipSizeY << endl;
137
138     // Call the base class's ToString method to print
           additional resource information.
139     Resource::ToString();
140 }
```

Listing 24: C++ and SDL Program - SpriteSheet.cpp

2.25 SpriteSheet.h

```
1  #ifndef SPRITESHEET_H
2  #define SPRITESHEET_H
3
4  #include "Texture.h"
5  #include "SpriteAnim.h"
6  #include "BasicStructs.h"
7
8  // Enum to represent various animation states.
9  // These states represent different animations that can be
       played by the sprite, such as idle, run, attack, etc.
10 enum AnimationNames
11 {
12     EN_AN_IDLE = 0,              // Idle animation
13     EN_AN_RUN,                   // Running animation
14     EN_AN_TWO_COMBO_ATTACK,      // Combo attack animation
15     EN_AN_DEATH,                 // Death animation
```

```
    EN_AN_HURT,                    // Hurt animation
    EN_AN_JUMP_UP_FALL,            // Jump and fall animation
    EN_AN_EDGE_GRAB,               // Edge grab animation
    EN_AN_EDGE_IDLE,               // Edge idle animation
    EN_AN_WALL_SIDE,               // Wall slide animation
    EN_AN_CROUCH,                  // Crouching animation
    EN_AN_DASH,                    // Dashing animation
    EN_AN_DASH_ATTACH,             // Dash attack animation
    EN_AN_SLIDE,                   // Sliding animation
    EN_AN_LADDER_GRAB,             // Ladder grab animation
    ROCK_FALL                      // Rock fall animation
       (for environmental hazards)
};

// The SpriteSheet class represents a sheet of sprites
   that contains multiple animations.
// It inherits from Texture and manages multiple
   animations for a given sprite, allowing for easy
   transitions
// between different animation states. Each animation is
   made up of a series of frames (or clips).
class SpriteSheet : public Texture
{
public:
    // Constructor: Initializes the SpriteSheet object.
    SpriteSheet();

    // Destructor: Cleans up resources used by the
       SpriteSheet object.
    ~SpriteSheet();

    // Methods

    // Serializes the sprite sheet's data to an output
       stream (e.g., saving to a file).
    void Serialize(std::ostream& _stream) override;

    // Deserializes the sprite sheet's data from an input
       stream (e.g., loading from a file).
    void Deserialize(std::istream& _stream) override;
```

```
48
49      // Converts the SpriteSheet's details into a string
            for debugging or logging purposes.
50      void ToString() override;
51
52      // Sets the size of the sprite sheet, specifying the
            number of rows and columns of sprites,
53      // as well as the width and height of each clip
            (sprite).
54      void SetSize(byte _rows, byte _columns, byte
            _clipSizeX, byte clipSizeY);
55
56      // Adds an animation to the sprite sheet, defining its
            name, starting clip, number of clips, and speed.
57      void AddAnimation(AnimationNames _name, short
            _clipStart, short _clipCount, float _clipSpeed);
58
59      // Updates the animation for the given state
            (identified by `_name`) based on the delta time,
60      // and returns the current frame's rectangular
            position on the sprite sheet.
61      Rect Update(AnimationNames _name, float _deltaTime);
62
63      // Returns the index of the current clip in the
            specified animation.
64      int GetCurrentClip(AnimationNames _name);
65
66      // Object pool for managing SpriteSheet instances,
            reducing the cost of memory allocation and
            deallocation.
67      static ObjectPool<SpriteSheet>* Pool;
68
69  private:
70      // Members
71
72      // Number of rows in the sprite sheet (how many
            sprites vertically).
73      byte m_rows;
74
```

```
    // Number of columns in the sprite sheet (how many
       sprites horizontally).
    byte m_columns;

    // Width of each clip (sprite) in the sheet.
    byte m_clipSizeX;

    // Height of each clip (sprite) in the sheet.
    byte m_clipSizeY;

    // Map of animations, where each key is an
       AnimationNames enum and the value is a SpriteAnim
       object
    // that holds the animation data (such as frames and
       speed).
    map<AnimationNames, SpriteAnim*> m_animations;
};

#endif // SPRITESHEET_H
```

Listing 25: C++ and SDL Program - SpriteSheet.h

2.26 StackAllocator.cpp

```
#include "StackAllocator.h"
#include "StandardIncludes.h"

// Constructor: Initializes the StackAllocator object by
   clearing the memory.
StackAllocator::StackAllocator()
{
    ClearMemory();  // Reset all pointers to null and
       clear memory.
}

// Destructor: Clears the allocated memory when the
   StackAllocator is destroyed.
StackAllocator::~StackAllocator()
{
```

```
    ClearMemory();  // Ensure that any allocated memory is
       freed upon destruction.
}

// AllocateStack method: Allocates a stack of memory with
   the given size in bytes.
// _stackSizeBytes: The size of the stack to allocate in
   bytes.
void StackAllocator::AllocateStack(unsigned int
   _stackSizeBytes)
{
    // Allocate memory for the stack.
    m_stackStart = new unsigned char[_stackSizeBytes];

    // Set the memory to zero for the allocated stack.
    memset(m_stackStart, 0, _stackSizeBytes);

    // Set the current stack position to the beginning of
       the stack.
    m_stackPosition = m_stackStart;

    // Set the end of the stack.
    m_stackEnd = m_stackStart + _stackSizeBytes;
}

// GetMemory method: Allocates a block of memory from the
   stack.
// _sizeBytes: The size of the memory block to allocate.
// Returns a pointer to the allocated memory, or nullptr
   if not enough memory is available.
unsigned char* StackAllocator::GetMemory(unsigned int
   _sizeBytes)
{
    unsigned char* hold = m_stackPosition;  // Save the
       current position.

    // Check if there is enough space in the stack.
    if (m_stackPosition + _sizeBytes <= m_stackEnd)
    {
```


www.assignmentcompletion

```
43          m_stackPosition += _sizeBytes;  // Move the stack
               position forward.
44          return hold;  // Return the pointer to the
               allocated memory.
45      }
46
47      // If there is not enough space, return nullptr.
48      return nullptr;
49  }
50
51  // Mark method: Saves the current position in the stack as
       a marker.
52  // This allows memory to be freed back to this position
       later.
53  void StackAllocator::Mark()
54  {
55      m_marker = m_stackPosition;  // Set the marker to the
           current position in the stack.
56  }
57
58  // FreeToMarker method: Frees all memory allocated after
       the marker position.
59  // This effectively rewinds the stack to the marker,
       allowing reallocation from that point.
60  void StackAllocator::FreeToMarker()
61  {
62      m_stackPosition = m_marker;  // Reset the current
           position to the marker.
63      *m_stackPosition = 0;  // Set the memory at the marker
           position to zero.
64  }
65
66  // ClearMemory method: Frees all allocated memory and
       resets all pointers.
67  // This method is used to completely clear the stack.
68  void StackAllocator::ClearMemory()
69  {
70      // If the stack has been allocated, free the memory.
71      if (m_stackStart != nullptr)
72      {
```


assignmentcompletion

```
        delete[] m_stackStart;  // Free the allocated
            stack memory.
    }

    // Reset all pointers to null.
    m_marker = nullptr;
    m_stackStart = nullptr;
    m_stackPosition = nullptr;
    m_stackEnd = nullptr;
}
```

Listing 26: C++ and SDL Program - StackAllocator.cpp

2.27 StackAllocator.h

```
#ifndef STACK_ALLOCATOR_H
#define STACK_ALLOCATOR_H

// The StackAllocator class is a custom memory allocator
   that allocates memory in a stack-like manner.
// It allows for efficient memory allocation by providing
   memory in contiguous blocks and supports
// basic operations such as marking a memory position and
   freeing back to that marker.

class StackAllocator
{
public:
    // Constructor: Initializes a new StackAllocator
       object.
    StackAllocator();

    // Destructor: Cleans up the allocated memory used by
       the stack allocator.
    ~StackAllocator();

    // Accessors

    // Returns the current marker, which represents a
       specific position in the stack where memory can be
```

```
        freed back to.
    unsigned char* GetMarker() { return m_marker; }

    // Returns the number of bytes currently used from the
       stack by calculating the difference between the
       current
    // position (`m_stackPosition`) and the start of the
       stack (`m_stackStart`).
    int GetBytesUsed() { return
       static_cast<int>(m_stackPosition - m_stackStart); }

    // Methods

    // Allocates a block of memory for the stack allocator
       of the specified size (_stackSizeBytes).
    // This method sets up the stack and defines its
       boundaries.
    void AllocateStack(unsigned int _stackSizeBytes);

    // Allocates memory from the stack of the requested
       size (_sizeBytes).
    // Returns a pointer to the allocated memory.
    unsigned char* GetMemory(unsigned int _sizeBytes);

    // Marks the current position of the stack so that
       memory can be freed back to this point later.
    void Mark();

    // Frees memory back to the most recently marked
       position, effectively undoing any allocations made
       after that point.
    void FreeToMarker();

    // Clears all memory allocated by the stack, resetting
       the stack to its initial state.
    // This does not free the memory itself but resets the
       internal pointers.
    void ClearMemory();

private:
```


assignmentcompletion

```
    // Members

    // Pointer to the current marker, which represents the
       last marked position in the stack.
    unsigned char* m_marker;

    // Pointer to the start of the memory block used for
       the stack.
    unsigned char* m_stackStart;

    // Pointer to the current position in the stack where
       the next memory allocation will occur.
    unsigned char* m_stackPosition;

    // Pointer to the end of the memory block, used to
       ensure that allocations do not exceed the stack
       size.
    unsigned char* m_stackEnd;
};

#endif // STACK_ALLOCATOR_H
```

Listing 27: C++ and SDL Program - StackAllocator.h

2.28 StandardIncludes.h

```
#ifndef STANDARD_INCLUDES_H
#define STANDARD_INCLUDES_H

#define SDL_MAIN_HANDLED // Ensures that SDL does not
   redefine the main function for compatibility with
   custom entry points.

// Standard library headers
#include <string>      // For handling standard string
   operations.
#include <vector>      // For using dynamic arrays
   (vectors).
#include <iostream>    // For standard input and output
   streams.
```

```cpp
#include <fstream>     // For file stream operations (reading/writing files).
#include <cinttypes>   // For fixed-size integer types (e.g., int32_t, uint8_t).
#include <stdio.h>     // For standard C I/O operations.
#include <thread>      // For multithreading support.
#include <map>         // For using associative containers (maps).
#include <SDL.h>       // SDL library for handling graphics, input, and window management.
#include <SDL_ttf.h>   // SDL extension library for handling TrueType fonts.
#include <SDL_pixels.h>// SDL structure for pixel format definitions.
#include <cstdlib>     // For standard library utilities like memory allocation and process control.

// Project-specific headers
#include "Singleton.h"     // For the Singleton design pattern implementation.
#include "BasicStructs.h"   // For basic structs like Color, Point, and Rect.

// Platform-specific headers and macros
#ifdef _WIN32
#include <Windows.h>    // Windows-specific functions and types.
#include <direct.h>     // For directory handling on Windows.

#define M_ASSERT(_cond, _msg) \
    if (!(_cond)) { \
        OutputDebugStringA(_msg); /* Outputs the error message to the debugger console */ \
        std::abort();             /* Aborts the program if the condition fails */ \
    }

#define GetCurrentDir _getcwd // Define macro to get the current directory on Windows.
```


assignmentcompletion

```

#else
#include <unistd.h>    // POSIX-specific functions for
   Unix-based systems.

#define GetCurrentDir getcwd // Define macro to get the
   current directory on Unix-based systems.

#endif

// Using the standard namespace to avoid needing to prefix
   standard library types with `std::`.
using namespace std;

#endif // STANDARD_INCLUDES_H
```

Listing 28: C++ and SDL Program - StandardIncludes.h

2.29 Texture.cpp

```
#include "Texture.h"

// Initialize the static ObjectPool for Texture instances.
// This pool is used to manage and reuse Texture objects
   efficiently.
ObjectPool<Texture>* Texture::Pool = nullptr;

// Constructor: Initializes the Texture object by setting
   default values.
Texture::Texture()
{
    m_imageInfo = {};        // Initialize the image
       information structure to empty/default values.
    m_texture = nullptr;     // Initialize the texture
       pointer to null (no texture loaded initially).
}

// Destructor: Default destructor for the Texture class.
// No specific cleanup required as assets are managed
   externally.
```

```
16 Texture::~Texture()
17 {
18 }
19
20 // Load method: Loads a TGA texture from a file using the
      TGAReader.
21 // _guid: The file path or identifier for the texture.
22 void Texture::Load(string _guid)
23 {
24     TGAReader r = TGAReader();  // Create an instance of
          the TGAReader to read the texture file.
25
26     // Load the TGA file and store the texture data in
          m_texture and image information in m_imageInfo.
27     m_texture = r.LoadTGAFromFile(_guid, &m_imageInfo);
28 }
29
30 // Serialize method: Serializes the texture's data (asset)
      into the output stream.
31 // This method writes the necessary information to save
      the state of the texture.
32 void Texture::Serialize(std::ostream& _stream)
33 {
34     // Serialize the texture's asset (m_texture) into the
          stream.
35     SerializeAsset(_stream, m_texture);
36 }
37
38 // Deserialize method: Deserializes the texture's data
      (asset) from the input stream.
39 // This method restores the texture state and processes
      the texture data.
40 void Texture::Deserialize(std::istream& _stream)
41 {
42     TGAReader r = TGAReader();  // Create an instance of
          the TGAReader to process the asset.
43
44     // Deserialize the texture's asset (m_texture) from
          the stream.
45     DeserializeAsset(_stream, m_texture);
```


www.assignmentcompletion

```
46
47      // Process the texture asset using TGAReader to update
            m_imageInfo.
48      r.ProcessAsset(m_texture, &m_imageInfo);
49  }
50
51  // ToString method: Outputs information about the texture
        for debugging purposes.
52  // This includes calling the ToString method of the
        texture asset.
53  void Texture::ToString()
54  {
55      cout << "TEXTURE" << endl;  // Print a label
            indicating this is a texture.
56
57      // If the texture exists, call its ToString method to
            output its details.
58      if (m_texture != nullptr)
59      {
60          m_texture->ToString();  // Call the asset's
                ToString method.
61      }
62
63      // Call the base class's ToString method to print
            additional resource information.
64      Resource::ToString();
65  }
```

Listing 29: C++ and SDL Program - Texture.cpp

2.30 Texture.h

```
1  #ifndef TEXTURE_H
2  #define TEXTURE_H
3
4  #include "Resource.h"
5  #include "TGAReader.h"
6
7  // Forward declaration of the Asset class.
```

```
class Asset;

// The Texture class represents an image or texture resource in the game.
// It inherits from the Resource class, allowing it to be serialized, deserialized, and managed by the resource system.
// The texture data is associated with an Asset, and image information is stored in the ImageInfo structure.

class Texture : public Resource
{
public:
    // Constructor: Initializes a new Texture object.
    Texture();

    // Destructor: Cleans up the Texture object and releases associated resources.
    virtual ~Texture();

    // Accessors

    // Returns the texture data as an Asset pointer.
    // The texture data contains the actual image information for rendering.
    Asset* GetData() { return m_texture; }

    // Returns a pointer to the ImageInfo structure, which contains metadata about the image (width, height, etc.).
    ImageInfo* GetImageInfo() { return &m_imageInfo; }

    // Methods

    // Serializes the texture's data to an output stream (e.g., saving to a file).
    void Serialize(std::ostream& _stream) override;

    // Deserializes the texture's data from an input stream (e.g., loading from a file).
```


assignmentcompletion

```
    void Deserialize(std::istream& _stream) override;

    // Converts the texture's details into a string for
       debugging or logging purposes.
    void ToString() override;

    // Loads the texture data from an asset identified by
       its GUID (global unique identifier).
    // The GUID is used to locate and load the
       corresponding texture data.
    void Load(string _guid);

    // Object pool for managing Texture instances,
       optimizing memory usage by reusing objects.
    static ObjectPool<Texture>* Pool;

private:
    // Members

    // Structure that holds metadata about the image, such
       as its dimensions, format, etc.
    ImageInfo m_imageInfo;

    // Pointer to the Asset that holds the actual texture
       data.
    Asset* m_texture;
};

#endif // TEXTURE_H
```

Listing 30: C++ and SDL Program - Texture.h

2.31 TGAReader.cpp

```
#include "TGAReader.h"
#include "AssetController.h"

// Constructor: Initializes the TGAReader object by
   setting the header to default values and data to null.
```


assignmentcompletion

```
TGAReader::TGAReader()
{
    m_header = {};      // Initialize the TGA header to default (empty) values.
    m_data = nullptr;   // Initialize the data pointer to null (no data loaded initially).
}

// ProcessAsset method: Processes the raw TGA data to extract image information such as width, height, and pixel depth.
// _rawTGA: The asset containing the raw TGA data.
// _imageInfo: A structure to store the image information extracted from the TGA header.
void TGAReader::ProcessAsset(Asset* _rawTGA, ImageInfo* _imageInfo)
{
    // Copy the TGA header from the raw data into the m_header structure.
    memcpy(&m_header, _rawTGA->GetData(), sizeof(TGAHeader));

    // Check if the image format is supported.
    M_ASSERT((m_header.DataTypeCode == 2), "Can only handle image type 2 (uncompressed true-color images).");
    M_ASSERT((m_header.BitsPerPixel == 24 || m_header.BitsPerPixel == 32), "Can only handle pixel depths of 24 and 32.");
    M_ASSERT((m_header.ColourMapType == 0), "Can only handle color map type 0 (no color map).");

    // Calculate the data offset within the TGA file buffer.
    int dataOffset = sizeof(TGAHeader);              // Offset starts after the TGA header.
    dataOffset += m_header.IDLength;                 // Add the ID length to the offset.
    dataOffset += m_header.ColourMapType * m_header.ColourMapLength;  // Add the length of the
```


assignmentcompletion

```
        color map (if any).
28
29      // Store the image properties in the ImageInfo
            structure.
30      _imageInfo->Width = m_header.Width;              //
            Set the image width.
31      _imageInfo->Height = m_header.Height;            //
            Set the image height.
32      _imageInfo->BitsPerPixel = m_header.BitsPerPixel; //
            Set the bits per pixel (color depth).
33      _imageInfo->DataOffset = dataOffset;             //
            Set the offset where the pixel data starts.
34  }
35
36  // LoadTGAFromFile method: Loads a TGA file into an Asset
        object and processes it to extract image information.
37  // _file: The file path or identifier of the TGA file.
38  // _imageInfo: A structure to store the image information
        extracted from the TGA header.
39  // Returns the asset containing the TGA image data.
40  Asset* TGAReader::LoadTGAFromFile(string _file, ImageInfo*
        _imageInfo)
41  {
42      // Read the TGA file into an Asset object using the
            AssetController.
43      m_data = AssetController::Instance().GetAsset(_file);
44
45      // Process the asset to extract the image information
            (e.g., width, height, pixel depth).
46      ProcessAsset(m_data, _imageInfo);
47
48      // Return the loaded asset containing the TGA image
            data.
49      return m_data;
50  }
```

Listing 31: C++ and SDL Program - TGAReader.cpp

2.32 TGAReader.h

www.assignmentcompletion

```h
#ifndef TGA_READER_H
#define TGA_READER_H

#include "StandardIncludes.h"

// Forward declaration of the Asset class.
class Asset;

// Struct representing the header of a TGA (Truevision TGA) image file.
// This header contains metadata about the image such as its size, color depth, and format.
// The pragma directive ensures that the struct is packed tightly in memory without padding.
#pragma pack(push,1)
typedef struct
{
    char IDLength;            // Size of the ID field that follows the 18-byte header (usually 0).
    char ColourMapType;       // Type of color map: 0 = none, 1 = has a color palette.
    char DataTypeCode;        // Image type: 0 = none, 1 = indexed, 2 = RGB, 3 = greyscale, +8 = RLE packed.
    short ColourMapStart;     // First entry in the color map (palette).
    short ColourMapLength;    // Number of colors in the color map.
    char ColourMapDepth;      // Number of bits per palette entry (15, 16, 24, or 32 bits per pixel).
    short x_Origin;           // X origin of the image.
    short y_Origin;           // Y origin of the image.
    short Width;              // Width of the image in pixels.
    short Height;             // Height of the image in pixels.
    char BitsPerPixel;        // Bits per pixel: 8, 16, 24, or 32 bits.
    char ImageDescriptor;     // Image descriptor byte that provides additional info (e.g., alpha channel
```


assignmentcompletion

```
        depth).
} TGAHeader;
#pragma pack(pop)

// Struct representing basic information about an image
   such as its width, height, and bit depth.
// Also includes the offset where the actual image data
   starts.
typedef struct
{
    short Width;          // Width of the image in pixels.
    short Height;         // Height of the image in pixels.
    char BitsPerPixel;    // Number of bits per pixel
       (e.g., 24 for RGB, 32 for RGBA).
    short DataOffset;     // Offset in the file where the
       image data begins.
} ImageInfo;

// The TGAReader class is responsible for reading and
   processing TGA (Truevision TGA) image files.
// It extracts image data and metadata from TGA files and
   provides methods for loading and processing
// the TGA assets.

class TGAReader
{
public:
    // Constructor: Initializes the TGAReader object.
    TGAReader();

    // Destructor: Default virtual destructor.
    virtual ~TGAReader() {}

    // Processes a raw TGA asset by reading its header and
       extracting image metadata.
    // The metadata is stored in the provided ImageInfo
       structure.
    void ProcessAsset(Asset* _rawTGA, ImageInfo*
       _imageInfo);

```

```
57      // Loads a TGA image from the specified file and
            stores the image metadata in the provided ImageInfo
            structure.
58      // Returns an Asset containing the image data.
59      Asset* LoadTGAFromFile(string _file, ImageInfo*
            _imageInfo);
60
61  private:
62      // TGA header structure that holds metadata about the
            TGA image.
63      TGAHeader m_header;
64
65      // Pointer to the raw image data loaded from the TGA
            file.
66      Asset* m_data;
67  };
68
69  #endif //TGA_READER_H
```

Listing 32: C++ and SDL Program - TGAReader.h

2.33 Timing.cpp

```
1   #include "Timing.h"
2
3   // Constructor: Initializes the Timing object by setting
        the initial values for FPS and time tracking.
4   Timing::Timing()
5   {
6       m_fpsCount = 0;                 // Initialize FPS count
            to 0 (frames processed in the current second).
7       m_fpsLast = 0;                  // Initialize last
            recorded FPS to 0.
8       m_deltaTime = 0;                // Initialize delta time
            (time between frames) to 0.
9       m_currentTime = SDL_GetTicks();  // Get the current
            time in milliseconds since SDL started.
10      m_lastTime = m_currentTime;      // Set the last time
            to the current time.
```

```
    m_fpsStart = m_currentTime;      // Set the start time for FPS calculation to the current time.
}

// Tick method: Updates the timing information for the current frame.
// This includes calculating the time difference between frames (delta time) and updating the FPS counter.
void Timing::Tick()
{
    // Get the current time in milliseconds since SDL started.
    m_currentTime = SDL_GetTicks();

    // Calculate the delta time (time between the last frame and the current frame) in seconds.
    m_deltaTime = (float)(m_currentTime - m_lastTime) / 1000.0f;

    // Check if one second has passed since the FPS count started.
    if (m_fpsStart + 1000 <= m_currentTime)
    {
        // If one second has passed, update the last recorded FPS with the current frame count.
        m_fpsLast = m_fpsCount;

        // Reset the FPS count for the next second.
        m_fpsCount = 0;

        // Reset the start time for FPS counting.
        m_fpsStart = m_currentTime;
    }
    else
    {
        // Increment the FPS count if less than a second has passed.
        m_fpsCount++;
    }

```

```
    // Update the last time to the current time (for the
       next frame's delta time calculation).
    m_lastTime = m_currentTime;
}
```

Listing 33: C++ and SDL Program - Timing.cpp

2.34 Timing.h

```
#ifndef TIMING_H
#define TIMING_H

#include "StandardIncludes.h"

// The Timing class is responsible for managing the timing
   and frame rate (FPS) in the game.
// It follows the Singleton design pattern, ensuring that
   only one instance of the Timing system
// is used throughout the application. This class handles
   calculating delta time between frames
// and measuring the game's frame rate.

class Timing : public Singleton<Timing>
{
public:
    // Constructor: Initializes the timing system and sets
       up initial time values.
    Timing();

    // Destructor: Default virtual destructor.
    virtual ~Timing() {}

    // Accessors

    // Returns the last calculated frames per second (FPS).
    unsigned int GetFPS() { return m_fpsLast; }

    // Returns the delta time, which is the time elapsed
       between the current frame and the previous frame.
```

v.assignmentcompletion.

```
26      // Delta time is used for smooth movement and
           animations regardless of frame rate fluctuations.
27      float GetDeltaTime() { return m_deltaTime; }
28
29      // Methods
30
31      // Tick method is called once per frame to update the
           timing information.
32      // It calculates the time difference between frames,
           updates the delta time, and tracks the FPS.
33      void Tick();
34
35  private:
36      // Members
37
38      // Stores the current time (in milliseconds) when the
           frame starts.
39      unsigned int m_currentTime;
40
41      // Stores the time of the previous frame.
42      unsigned int m_lastTime;
43
44      // Stores the time at which FPS counting started (used
           to measure the frame rate over time).
45      unsigned int m_fpsStart;
46
47      // Counts how many frames have been rendered since the
           last FPS calculation.
48      unsigned int m_fpsCount;
49
50      // Stores the last calculated FPS value.
51      unsigned int m_fpsLast;
52
53      // Delta time (time elapsed between the current and
           last frame).
54      // This is essential for keeping movement and
           animations frame-rate independent.
55      float m_deltaTime;
56  };
57
```


assignmentcompletion

```
#endif // TIMING_H
```

Listing 34: C++ and SDL Program - Timing.h

2.35 TTFont.cpp

```
#include "TTFont.h"
#include "Renderer.h"

// Constructor: Initializes the TTFont object by setting the font to nullptr and the destination rectangle to default values.
TTFont::TTFont()
{
    m_font = nullptr;          // Initialize the font pointer to null (no font loaded initially).
    destRect = {};             // Initialize the destination rectangle to empty/default values.
}

// Destructor: Default destructor for TTFont class.
// No specific cleanup required as font is managed externally.
TTFont::~TTFont()
{
}

// Initialize method: Initializes the TTF font system and loads a font from a file.
// _pointSize: The size of the font to load.
void TTFont::Initialize(int _pointSize)
{
    // Initialize the SDL_ttf library for font rendering.
    M_ASSERT((TTF_Init() >= 0), "Unable to initialize SDL TTF.");

    // Load the font from the specified file (arial.ttf) with the given point size.
    M_ASSERT((m_font = TTF_OpenFont("../Assets/Fonts/arial.ttf",
```

```
        _pointSize)) != nullptr, "Failed to load font.");
}

// Shutdown method: Shuts down the SDL_ttf library.
void TTFont::Shutdown()
{
    // Quit the SDL_ttf library.
    TTF_Quit();
}

// Write method: Renders a text string to the screen using the loaded font.
// _renderer: The SDL renderer used to draw the text.
// _text: The text string to render.
// _color: The color of the text.
// _pos: The position where the text will be rendered.
void TTFont::Write(SDL_Renderer* _renderer, const char* _text, SDL_Color _color, SDL_Point _pos)
{
    // Create an SDL surface for the rendered text using the UTF8-blended method (smooth text rendering).
    SDL_Surface* surface;
    surface = TTF_RenderUTF8_Blended(m_font, _text, _color);

    // Create a texture from the surface, which can then be rendered to the screen.
    SDL_Texture* texture;
    texture = SDL_CreateTextureFromSurface(_renderer, surface);

    // Define the destination rectangle where the text will be rendered on the screen.
    SDL_Rect destRect{ _pos.x, _pos.y, surface->w, surface->h };

    // Render the texture (containing the text) to the screen using SDL_RenderCopyEx.
    M_ASSERT(((SDL_RenderCopyEx(_renderer, texture, nullptr, &destRect, 0, nullptr, SDL_FLIP_NONE)) >=
```


assignmentcompletion

```
        0), "Could not render texture");

    // Free the surface and texture to avoid memory leaks.
    SDL_FreeSurface(surface);
    SDL_DestroyTexture(texture);
}
```

Listing 35: C++ and SDL Program - TTFont.cpp

2.36 TTFont.h

```
#ifndef TTFONT_H
#define TTFONT_H

#include "StandardIncludes.h"

// Forward declaration of the Renderer class.
class Renderer;

// The TTFont class handles the loading, rendering, and
   management of TrueType fonts (TTF) in the game.
// It provides methods for initializing the font, writing
   text to the screen, and cleaning up resources.
// The class relies on the SDL_ttf library for font
   rendering.

class TTFont
{
public:
    // Constructor: Initializes a TTFont object.
    TTFont();

    // Destructor: Cleans up the font and releases
       resources when the TTFont object is destroyed.
    virtual ~TTFont();

    // Methods

    // Initializes the TrueType font with the specified
       point size (_pointSize).
```


www.assignmentcompletion

```
25      // The point size determines the size of the rendered
            text.
26      void Initialize(int _pointSize);
27
28      // Renders the provided text (_text) to the screen
            using the specified SDL_Renderer (_renderer),
29      // with the given font color (_color) and position
            (_pos).
30      void Write(SDL_Renderer* _renderer, const char* _text,
            SDL_Color _color, SDL_Point _pos);
31
32      // Shuts down the TTFont system, releasing any
            resources associated with the font.
33      void Shutdown();
34
35  private:
36      // Members
37
38      // Pointer to the loaded TTF font structure.
39      TTF_Font* m_font;
40
41      // SDL rectangle used to define the destination area
            for rendering the text.
42      SDL_Rect destRect;
43  };
44
45  #endif // TTFONT_H
```

Listing 36: C++ and SDL Program - TTFont.h

2.37 Unit.cpp

```
1  #include "Unit.h"
2
3  // Initialize the static ObjectPool for Unit instances.
4  // This pool is used to manage and reuse Unit objects
       efficiently.
5  ObjectPool<Unit>* Unit::Pool = nullptr;
6
```

```cpp
// Constructor: Initializes the Unit object by setting the sound effect pointer to null.
Unit::Unit()
{
    m_soundEffect = nullptr;  // Initialize the sound effect pointer to null (no sound effect loaded initially).
}

// Destructor: Default destructor for Unit class.
Unit::~Unit()
{
}

// AssignNonDefaultValues method: Assigns default values to the Unit's members for testing or specific use cases.
void Unit::AssignNonDefaultValues()
{
    // Retrieve a sound effect from the object pool and assign non-default values to it.
    m_soundEffect = SoundEffect::Pool->GetResource();
    m_soundEffect->AssignNonDefaultValues();

    // Call the base class method to assign non-default values to the Resource members.
    Resource::AssignNonDefaultValues();
}

// Serialize method: Serializes the Unit's data, including the sound effect, into the output stream.
// This method writes the necessary information to save the state of the Unit.
void Unit::Serialize(std::ostream& _stream)
{
    // Serialize the sound effect pointer.
    SerializePointer(_stream, m_soundEffect);

    // Serialize the base class (Resource) data.
    Resource::Serialize(_stream);
}
```


assignmentcompletion

```cpp

// Deserialize method: Deserializes the Unit's data, including the sound effect, from the input stream.
// This method restores the state of the Unit object.
void Unit::Deserialize(std::istream& _stream)
{
    // Deserialize the sound effect pointer.
    DeserializePointer(_stream, m_soundEffect);

    // Deserialize the base class (Resource) data.
    Resource::Deserialize(_stream);
}

// ToString method: Outputs information about the Unit object for debugging purposes.
// This includes details about the sound effect and calls the base class ToString method.
void Unit::ToString()
{
    // Print "UNIT" to indicate this is a Unit object.
    cout << "UNIT" << endl;

    // If the sound effect exists, call its ToString method to output its details.
    if (m_soundEffect != nullptr)
    {
        m_soundEffect->ToString();
    }

    // Call the base class's ToString method to print additional resource information.
    Resource::ToString();
}
```

Listing 37: C++ and SDL Program - Unit.cpp

2.38 Unit.h

```cpp
#ifndef UNIT_H
```


www.assignmentcompletion

```cpp
#define UNIT_H

#include "Resource.h"
#include "SoundEffect.h"

// The Unit class represents a game unit or entity that is part of the game world.
// It inherits from the Resource class, allowing it to be serialized, deserialized, and managed efficiently.
// Each unit can have associated resources, such as sound effects, which are handled by the SoundEffect class.

class Unit : public Resource
{
public:
    // Constructor: Initializes a new Unit object.
    Unit();

    // Destructor: Cleans up resources when the Unit object is destroyed.
    virtual ~Unit();

    // Serializes the unit's data to an output stream (e.g., saving the unit's state to a file).
    void Serialize(std::ostream& _stream) override;

    // Deserializes the unit's data from an input stream (e.g., loading the unit's state from a file).
    void Deserialize(std::istream& _stream) override;

    // Converts the unit's details into a string for debugging or logging purposes.
    void ToString() override;

    // Assigns non-default values to the unit, allowing customization or initialization of specific attributes.
    void AssignNonDefaultValues() override;

```


www.assignmentcompletion

```
32      // Object pool for managing Unit instances, optimizing
           memory usage by reusing objects.
33      static ObjectPool<Unit>* Pool;
34
35  private:
36      // Pointer to a SoundEffect object associated with the
           unit.
37      // This sound effect could be played during specific
           unit actions (e.g., attack, death, etc.).
38      SoundEffect* m_soundEffect;
39  };
40
41  #endif // UNIT_H
```

Listing 38: C++ and SDL Program - Unit.h

www.assignmentcompletion.com