**TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN**
**– ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH**

Khoa Công nghệ Thông tin

**CS202 - Programming Systems**

# Super Mario OOP Project

Nhóm thực hiện: 11

**Thành viên thực hiện:**
Phan Nhất Thành (23125091)
Nguyễn Thanh Hữu (24125030)
Văn Tuấn Khải (24125032)
Lê Tống Thiện Nhân (24125036)
Nguyễn Đình Thiên Lộc (24125093)

**Giáo viên hướng dẫn:**
Đinh Bá Tiến
Hồ Tuấn Thanh

# Table of Contents

# List of Tables

## Abstract

This project presents a desktop platformer game inspired by Super Mario, written in C++ using the raylib graphics library. The game features classic Mario mechanics such as running, jumping, collecting coins, defeating monsters, and progressing through levels. The codebase is modular and object-oriented, supporting extensibility for new levels, enemies, and features.

## 1. Introduction

Platformer games like Mario are iconic in computer science education and game development. This project aims to recreate the Mario experience, providing a hands-on opportunity to learn about game loops, collision detection, resource management, and real-time rendering in C++. The project demonstrates how classic gameplay can be implemented with modern programming practices.

## 2. Work Division and Implementation Notes

| **Class / Work** | **Assigned To** |
|---|---|
| Enemies (Koopa, Goomba, Banzai Bill, etc.) | Lộc |
| Blocks (Item Block, Stone, etc.) | Hữu |
| Items (Fire Flower, Coin, Super Mushroom, 1Up Mushroom, etc.) | Khải |
| Characters (Mario, Luigi), Core Systems (Sound, Logic, Map, etc.) | Nhân |

Table 1: Work Division by Module

**Important Implementation Notes:**

- **About Inheritance:**
  - Every object class inherits from `Entity`.
- **About Implementation:**
  - The `Draw` function is called separately from `updateStateAndPhysics`.
  - `updateStateAndPhysics` is called using a **DeltaTime** step (see Mario class for logic).
- **About Resources:**
  - All textures and sounds are loaded through `ResourceManager` (Singleton Pattern).
  - All sounds are played through `SoundController` (Singleton Pattern).
- **About File Structure:**
  - Header files: `include/`
  - Source files: `src/`
  - Resource files: `resources/`

## 3. Data Storage

Game data such as levels, tiles, and enemy placements are stored in JSON files and loaded at runtime. In-memory, the game uses C++ classes and STL containers (vectors, maps) to manage entities, resources, and game state. ResourceManager handles textures, sounds, and music, ensuring efficient loading and unloading.

## 4. Project Architecture

- **Language:** C++
- **Graphics:** raylib (cross-platform game graphics library)
- **Key Modules:**
  - `main.cpp` – Main game loop, window and audio management
  - `StateManager/States` – Menu, gameplay, and settings states
  - `Level/Map` – Level logic, map loading, entity placement
  - `PlayableCharacter/Monster/Item/Block` – Entity classes
  - `ResourceManager` – Texture, sound, and music management
  - `CollisionMediator` – Handles all collision logic
- **Development:** Modular C++ project, built using CMake or Visual Studio

## 5. Implementation Details

- **Player:** Supports running, jumping, power-ups, and firing fireballs.
- **Enemies:** Includes Goomba, BanzaiBill, and others, each with unique movement and collision logic.
- **Items:** Coins, mushrooms, and power-ups with collection and effect logic.
- **Blocks:** Question blocks, breakable blocks, and static tiles.
- **Level System:** Loads map and object data from `JSON`, spawns entities, and manages sections for performance.
- **Collision:** All entity interactions (player-enemy, player-item, fireball-enemy, etc.) are handled by `CollisionMediator`.
- **Resource Management:** Centralized loading/unloading of textures, sounds, and music.
- **Menus and UI:** Main menu, settings, and pause implemented as separate states.

## 6. Technical Problems and Solutions

- **Consistent Physics:** Implemented a fixed time-step update loop to ensure smooth and predictable movement and collision, regardless of frame rate.
- **Resource Loading:** Designed `ResourceManager` to prevent redundant loading and ensure all assets are available when needed.
- **Collision Complexity:** Developed `CollisionMediator` to handle multiple entity types and resolve interactions efficiently.
- **Level Performance:** Used spatial partitioning (sections) to update and check collisions only for nearby entities.

## 7. Features Demonstration

- Classic Mario gameplay: running, jumping, collecting coins, defeating enemies.
- Multiple levels loaded from `JSON` map files.
- Power-ups and score system.
- Sound effects and background music.
- Pause, settings, and credits menus.
- Modular codebase for easy extension (new levels, enemies, items).

# 8. Implementation Details

This application is designed with a modular, object-oriented architecture. Each core game component (player, enemies, items, blocks, UI, etc.) is implemented as a separate class, with clear separation between logic, rendering, and resource management. The main control flow is managed by a centralized state manager and game loop.

## Main Control Flow

The file `main.cpp` manages the game loop, window and audio lifecycle, and state routing. User input and game state transitions are handled through a `StateManager.cpp`, which switches between menu, gameplay, settings, and credits. Each state is responsible for its own update and draw routines.

## Core Game Modules

- `main.cpp` – Initializes the window, audio, and resources; runs the main loop; delegates to the current state.
- `StateManager.cpp`, `MenuState.cpp`, `GameState.cpp` – Manage game states, menus, and transitions.
- `Level.cpp`, `Map.cpp` – Load and manage level data, including tiles, objects, and entity placement.
- `PlayableCharacter.cpp`, `Mario.cpp`, `Luigi.cpp` – Player logic, movement, power-ups, and actions.
- `Monster.cpp`, `Goomba.cpp`, `BanzaiBill.cpp`, etc. – Enemy behaviors and interactions.
- `Item.cpp`, `Coin.cpp`, `FireFlower.cpp`, etc. – Collectibles and power-up logic.
- `Block.cpp`, `QuestionBlock.cpp`, etc. – Interactive and static blocks.
- `ResourceManager.cpp` – Loads and manages textures, sounds, and music (singleton pattern).
- `SoundController.cpp` – Plays sound effects and music (singleton pattern).
- `CollisionMediator.cpp` – Handles all collision detection and resolution between entities.

**Flow Example:** When the player presses the jump key, `PlayableCharacter.cpp` updates the player's velocity. The main loop in `main.cpp` calls `stateManager.update()`, which triggers `Level::UpdateLevel()`. This updates all entities, applies physics, checks collisions via `CollisionMediator.cpp`, and then `stateManager.draw()` renders the updated game state.

## Level and Entity Management

- `Level.cpp` – Manages all entities in the level, including player, monsters, items, and blocks. Handles section-based updates for performance.
- `Map.cpp` – Loads level layout and object placement from JSON files.
- `MonsterFactory.cpp`, `ItemFactory.cpp`, `BlockFactory.cpp` – Create entities based on map data.

Entities are updated and drawn in a fixed time-step loop to ensure consistent physics and animation. Each entity class implements its own `updateStateAndPhysic()` and `Draw()` methods.

## Collision and Resource Management

- `CollisionMediator.cpp` – Centralizes all collision logic, handling interactions between player, monsters, items, and blocks.
- `ResourceManager.cpp` – Ensures all textures, sounds, and music are loaded once and reused throughout the game.
- `SoundController.cpp` – Handles all sound playback, ensuring no resource leaks and smooth audio transitions.

## Menus, UI, and Sound

- `MenuState.cpp`, `SettingMenuState.cpp`, `CreditState.cpp` – Implement the main menu, settings, and credits screens.
- `HUD.cpp` – Displays score, coins, lives, and other in-game information.
- `Button.cpp`, `Slider.cpp` – UI components for interaction.
- `SoundController.cpp` – Manages background music and sound effects for menus and gameplay.

**Summary:** The Mario project is structured for clarity and extensibility, with each module responsible for a specific aspect of the game. The main loop coordinates updates and rendering, while resource and collision management ensure efficient and correct gameplay. This modular approach allows for easy addition of new features, levels, and entities.

# 9. User's Manual

This section provides installation, setup, and usage instructions for the Super Mario OOP Project.

## 9.1. Installation Requirements

- Windows or Linux OS
- C++17 compatible compiler (Visual Studio or g++)
- raylib graphics library (`https://www.raylib.com/`)
- (Optional) vcpkg for Windows or Makefile/CMake for Linux builds

For a detailed guide on setting up raylib with vcpkg on Windows, see:
`https://www.youtube.com/watch?v=UiZGTIYld1M`

## 9.2. Building the Application

- **Windows:** Clone the repository, open the folder in Visual Studio, install raylib via vcpkg, then build and run the project.
- **Linux:** Install raylib (`sudo apt install libraylib-dev`), then run `make run` or use the provided CMake files.

## 9.3. Usage Instructions

1. Launch the application. The main menu will appear.
2. Use the arrow keys or mouse to navigate the menu.
3. Select "Start Game" to begin playing.
4. Choose your character (Mario or Luigi) if prompted.
5. Use the following controls during gameplay:
   - **Arrow keys:** Move left/right, crouch, or climb.
   - **Spacebar:** Jump.
   - **Z:** Run or shoot fireball (if powered up).
   - **Esc:** Pause the game and access the pause menu.
6. Collect coins, power-ups, and defeat enemies to progress through the level.
7. Reach the end of the level to complete it and advance.
8. Use the pause menu to adjust settings, restart, or return to the main menu.

## 9.4. Customization

- Music and sound effect volumes can be adjusted in the settings menu.
- Key bindings and color themes are not yet customizable in this version.

## 9.5. File Structure

- `resources/Map/` contains level map files in JSON format.
- `resources/Entity/` contains sprites for characters, enemies, and items.
- `resources/SFX/` and `resources/Music/` contain sound effects and background music.

## 9.6. Level Format Example

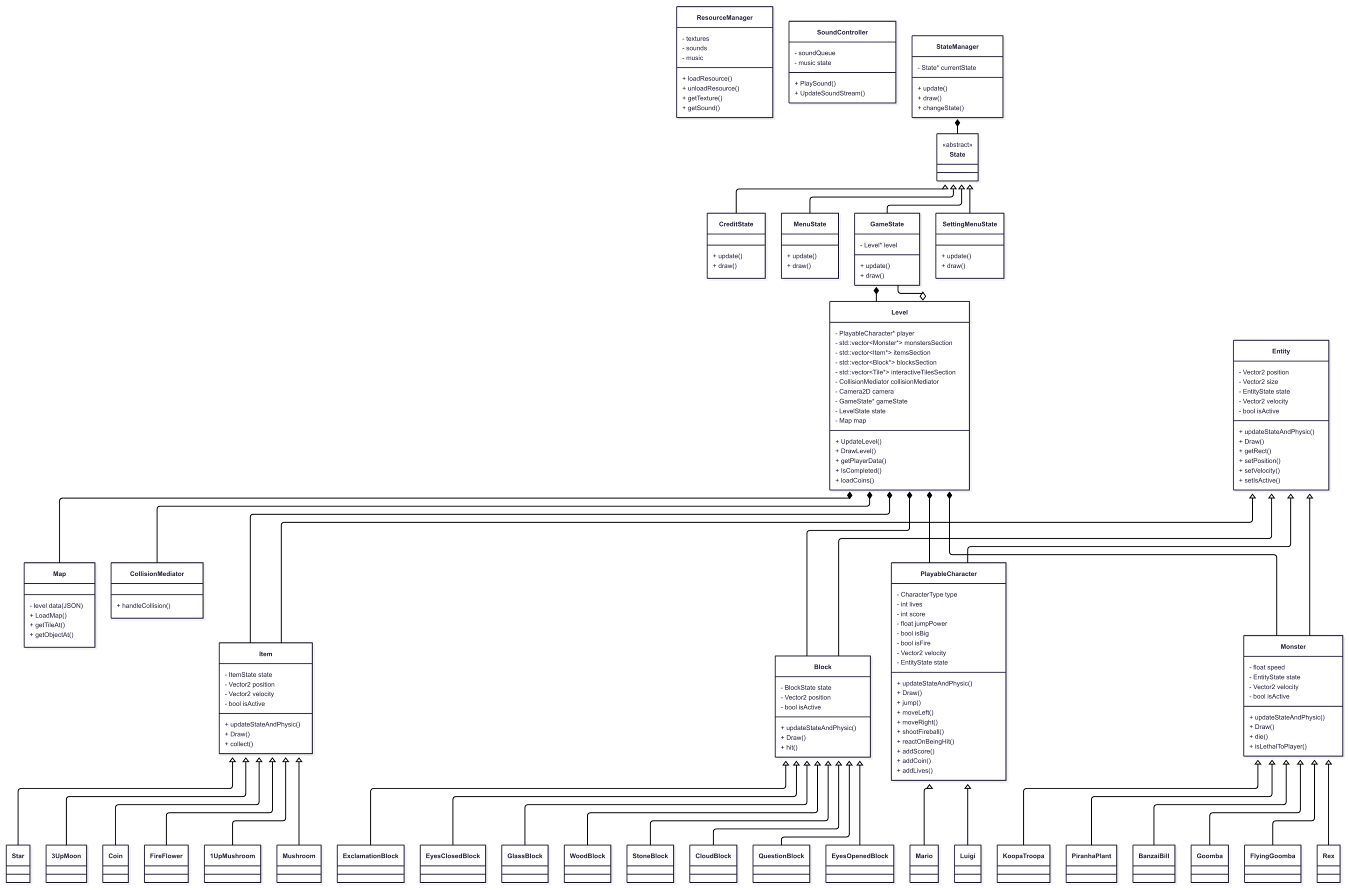
A level file (e.g., `map1.json`) contains tile and object layers. Example snippet:

```
{
  "layers": [
    {
      "type": "tilelayer",
      "data": [ ... ]
    },
    {
      "type": "objectgroup",
      "name": "Monsters",
      "objects": [
        { "type": "Goomba", "x": 500, "y": 700 },
        { "type": "BanzaiBill", "x": 1200, "y": 650 }
      ]
    }
  ]
}
```

## 9.7. Troubleshooting

- If the game does not launch, ensure raylib is installed and your compiler supports C++17.
- For missing textures or sounds, verify the `resources/` folder is present and complete.
- If controls do not respond, check your keyboard layout and focus on the game window.

## 10. UML Class Diagram



Hình 1: UML Class Diagram of Main Project Architecture

# 11. Code Snippets

This section presents representative code snippets for core mechanics and systems in the Super Mario OOP Project, aligned with the actual implementation logic from the source files.

### Player Jump and Power-Up Collection

```cpp
void PlayableCharacter::jump() {
    if (onGround) {
        velocity.y = -jumpPower;
        onGround = false;
        SoundController::getInstance().PlaySound("smw_jump.wav");
    }
}
void CollisionMediator::HandleMarioWithItem(Mario*& mario, Item*& item,
    CollisionInfo AtoB) {
    if (AtoB == COLLISION_NONE)
        return;

    if (item && CheckCollisionRecs(mario->getRect(), item->getRect())) {
        if (auto* mushroom = dynamic_cast<Mushroom*>(item)) {
            mushroom->collect();
            mario->setBig(true);
            mario->addScore(1000);
        }
        else if (auto* fireFlower = dynamic_cast<FireFlower*>(item)) {
            fireFlower->collect();
            mario->setFire(true);
            mario->addScore(1000);
        }
    }
}
```

Listing 1: Mario Jump and Power-Up Collection

### Monster Movement and Defeat

```cpp
void Goomba::updateStateAndPhysic() {
    if (!isActive) return;
    velocity.x = -speed;
    position.x += velocity.x * GameClock::getInstance().DeltaTime;
}
void CollisionMediator::HandleMarioWithMonster(Mario*& mario, Monster*& monster,
     CollisionInfo AtoB) {
    if (AtoB == COLLISION_SOUTH && mario->getVelocity().y > 0) {
        mario->addScore(400);
        monster->die();
        mario->setVelocity(Vector2{mario->getVelocity().x, -600}); // Bounce
        SoundController::getInstance().PlaySound("smw_stomp.wav");
    } else {
        mario->reactOnBeingHit();
    }
}
```

Listing 2: Goomba Movement and Stomp Defeat

## Block Hit and Item Spawn

```
void QuestionBlock::hit(Mario* mario) {
    if (state == BlockState::UNUSED) {
        state = BlockState::USED;
        spawnItem();
        SoundController::getInstance().PlaySound("smw_power-up_appears.wav");
    }
}
void QuestionBlock::spawnItem() {
    if (containsMushroom) {
        auto* mushroom = new Mushroom(position + Vector2{0, -size.y});
        Level::getInstance().addItem(mushroom);
    }
}
```

Listing 3: Question Block Hit and Item Spawn

## Level Update Loop

```
void Level::UpdateLevel() {
    player->updateStateAndPhysic();
    for (auto& section : monstersSection)
        for (auto* monster : section)
            monster->updateStateAndPhysic();
    for (auto& section : itemsSection)
        for (auto* item : section)
            item->updateStateAndPhysic();
    for (auto& section : blocksSection)
        for (auto* block : section)
            block->updateStateAndPhysic();

    // Collision checks
    collisionMediator.handleCollisions(player, monstersSection, itemsSection,
   blocksSection);
}
```

Listing 4: Level Update and Collision Handling

## Resource Management (Singleton Pattern)

```
Texture2D& ResourceManager::getTexture(const std::string& name) {
    if (textures.count(name) == 0) {
        textures[name] = LoadTexture(("resources/Entity/" + name).c_str());
    }
    return textures[name];
}
void ResourceManager::unloadResource() {
    for (auto& pair : textures) {
        UnloadTexture(pair.second);
    }
    textures.clear();
}
```

Listing 5: ResourceManager Singleton Usage

## References

- Visualization Demo. (2024, Apr 1). *Interactive Data Structures Visualizer using C++ and raylib* [Video]. YouTube. https://www.youtube.com/watch?v=UiZGTIYld1M
- raylib Setup with vcpkg for Visual Studio. (n.d.). GitHub Wiki. https://github.com/raysan5/raylib/wiki/Working-on-Windows#visual-studio-vcpkg
- RayMario Demo. (2020, Sep 20). *Mario-like Game implemented with raylib in C++* [Video]. YouTube. https://youtu.be/pq5NuXYhXYo
- Buzatto, D. (n.d.). *RayMario – A Super Mario-like clone using raylib and C++*. GitHub repository. https://github.com/davidbuzatto/RayMario
- Refactoring.Guru. (n.d.). *Design Patterns*. https://refactoring.guru/design-patterns
- Viblo. (n.d.). *Design Patterns Overview*. Viblo Community. https://viblo.asia/s/design-patterns-68Z00n2NZkG
- Wikipedia contributors. (n.d.). *Behavior tree (artificial intelligence, robotics and control)*. In Wikipedia. https://en.wikipedia.org/wiki/Behavior_tree_(artificial_intelligence,_robotics_and_control)

## 12. Conclusion

The Super Mario OOP Project demonstrates how object-oriented programming and modular design can be used to recreate a classic platformer game in C++. By separating game logic into clear classes and using patterns like singletons and state management, we achieved both functional gameplay and maintainable code.

Despite challenges with timing, collision, and resource handling, our team collaborated effectively to solve problems and deliver a smooth, feature-rich experience. The project structure also makes it easy to add new features or expand the game in the future.

Overall, this project provided valuable experience in C++ development, teamwork, and game architecture, and serves as a strong foundation for future enhancements

## Appendix: Repository and Video

- GitHub Repository: `https://github.com/letongthiennha/Mario`
- Demo Video (YouTube):