VNU-HCM

UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

FINAL PROJECT REPORT

Project: Mario Game

Course: CS202 - Programming Systems

Students :

Nguyen Chi Bao - 24125025

Pham Nguyen Minh Quan - 24125041

Phan Anh Khoa - 24125059

Quach Thien Lac - 24125092

Duy, Nguyen Quang - 18125048

August 23, 2025

Contents

1 Abstract:

This report will outline our group's work on the Mario game, including high-level architecture, implementation details, and all features of the game we made; it will show how the project adheres to SOLID principles and puts in use certain Gang-of-Four (GoF) design patterns to make development more robust. This report will also delineate technical difficulties we encountered and improvements we think can be made in the absence of time constraints. We also attach in this report a link to the live demonstration of the game.

2 Introduction:

As part of our CS202 - Programming Systems coursework, we are to create a Mario game clone that utilizes object-oriented (OO) concepts and design patterns we have learned. For this project, we have developed a simple 3-level game with basic blocks, power-ups, and enemy system, with a level editor and collision handler designed from scratch. We make sure that the game's architecture follows SOLID principles and implements around 5 GoF design patterns which we will explore further in this report.

3 Program Features:

3.1 Available features:

The game supports three main features: Play, Load, and Editor. Additionally, the Setting menu provides controls over certain aspect of the game. Below are the detailed explanation to each of these features:

1. **Play:** The ”story mode” gameplay option, where users play the three levels in order. Before starting to play, players can select between two characters: Luigi or Mario. The game then load the selected characters into the first map.
2. **Load:** Players can choose one in the three levels in the ”story mode” to play.
3. **Editor:** In the editor, players can make a map of their own to play. Players can save the map in two ways: to an in-built folder or to a specified path using a file dialog; they can load the same map from the file dialog as well.

4. **Settings:** Players can adjust the volume of sound effects and music. While playing, they also have the option to pause the game or return to the main menu (with or without saving their progress).

3.2 User Interface (UI):

- Using Raylib, a GUI library aimed towards game development, and royalty-free resource packs online, we design a user-friendly interface consisting of consistent and straightforward UI elements like buttons of similar graphic style or color for users to easily navigate the game.
- One notable aspect of the UI system is the map editor scene, which, despite remaining basic, communicates its functionality effectively. The controls are geared towards simplicity: players can place blocks with the left mouse, delete them with the right mouse, and traverse the map using the middle mouse; switching between the map layers and sprite sheets can be done via hotkeys. This design makes the drawing process more convenient and intuitive.

4 Architecture:

4.1 UML:

Below is the UML showing the high-level architecture of the game:

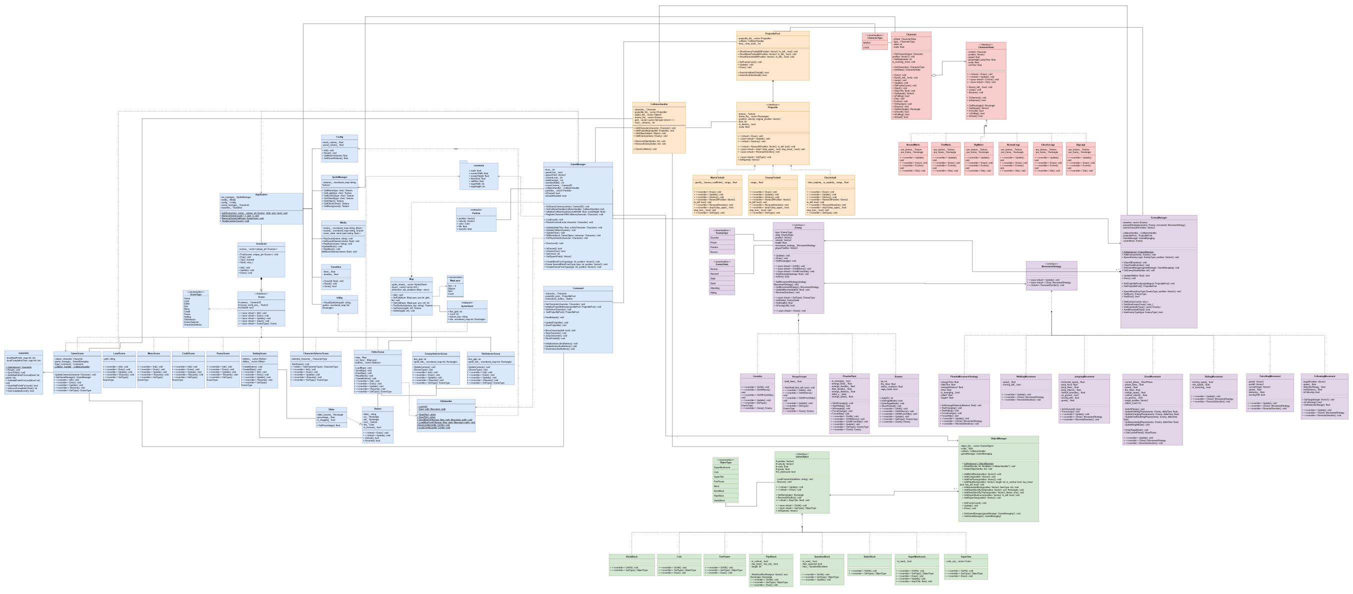


Figure 1: UML Diagram

4.2 High-level organization:

4.2.1 Codebase:

We choose a scene-based approach to the game by making all unique screens inherit from a Scene class with a four-stage lifecycle of Initialize - Draw - Update - Clear and managing their appearance via a scene stack. These scenes form the bulk of our Core system, along with managers for the game state, resources, and I/O. As per requirement of the game, we also design four other systems for major components of the game:

- Collision/Projectile (C/P): define projectile behaviors and how game objects (characters, enemies, objects) interact with each other.
- Character: define behaviors for playable characters (movement, power-ups, .etc.).
- Enemies: define behaviors for enemies (including boss and minions).
- Objects: define behaviors for objects (including blocks and power-ups).

4.2.2 Resources:

We organize all resources according to their type: image and audio. Image resources are mainly sprites, which are a mix between Super Mario 3, Super Mario World, and homegrown sprites; there are also packs of UI elements and backgrounds for certain screen. Sound effects and soundtracks are taken from various Mario games as we see fit. We make sure to only include royalty-free resources and pay attribution where necessary (in the credit section of the game's main menu).

4.2.3 Dependencies:

Apart from raylib, we also use Niels Lohmann's C++ json API for map loading and saving, and tinyfiles-dialogs for specifying save/load paths with a file dialog.

4.3 SOLID and Design Patterns Adherence:

4.3.1 SOLID principles:

Overall, the game adheres quite well to SOLID principles, with detailed examples outlined below:

1. **Single responsibility principle (SRP):** Most classes have a clear focus on what they do. For example, class SpriteManager only loads the textures for the sprites, which other classes can retrieve via its getter methods. However, certain classes, such as GameManaging, are still too clunky as a result of being overloaded with methods that deal with irrelevant functions such as updating the camera and particle effects and thus requires refactoring to better reflect their intended purposes.

2. **Open-closed principle (OCP):** The scene-based architecture allows the effortless inclusion of new scenes to extend capabilities without modifying existing code. Other components also express this principle, such as the serializers/deserializers functions in the Map class, which can readily be extended to handle other data types.

3. **Liskov Substitution principle (LSP):** Subclasses of Scene, the key components of the game, can be exchanged for their base class without breaking behavior. This adherence is also observed in other systems like Character or Object, where subclasses honor base classes' contracts.

4. **Interface Segregation principle (ISP):** Almost all systems, especially Enemy and Object, have a clear segregation between their interface and concrete implementation. Nonetheless, as stated

above, GameManaging class can benefit from further dividing its interface into smaller sub-services to decrease the level of tight coupling.

5. **Dependency Inversion principle (DIP):** Aside from the Scene class, Enemy class also exhibits this principle in practice by holding a reference to MovementStrategy and delegate all movement logic to it; this way, both the high-level modules (Enemy) and low-level ones (Goomba, Koopa) depend on the movement abstraction. This design can be mirrored in GameManaging, where the class is still dependent on a concrete CollisionHandler.

4.3.2 Design patterns:

We integrate numerous designs patterns into the game's architecture to enhance code performance and readability. Below is a table listing the design patterns used and their places in the codebase. We will explore these in-depth in the next section:

Pattern	**Where/How**
Singleton	`ObjectManager::GetInstance()`, `EnemyManager::GetInstance()`
Strategy	`MovementStrategy` injected into `Enemy` for AI behaviours
Template Method	`CollisionHandler` orchestrates collision checking process
State	`Character` delegates to `CharacterState` subclasses
Prototype	`Enemy::Clone()` to duplicate enemy instances
Factory	`GameManaging::CreateEnemyFromType()`, `CreateBlockFromType()` methods
Façade	`SpriteManager` hides texture-loading complexity
Adapter (JSON)	`nlohmann::adl_serializer` adapting `Map` and `Rectangle` to JSON

Table 1: An outline of design patterns in program Architecture

5 Implementation details:

We will divide this section into five parts, with each part exploring the detailed workings of each subsystem in the project that we outlined in the previous section.

5.1 Core system:

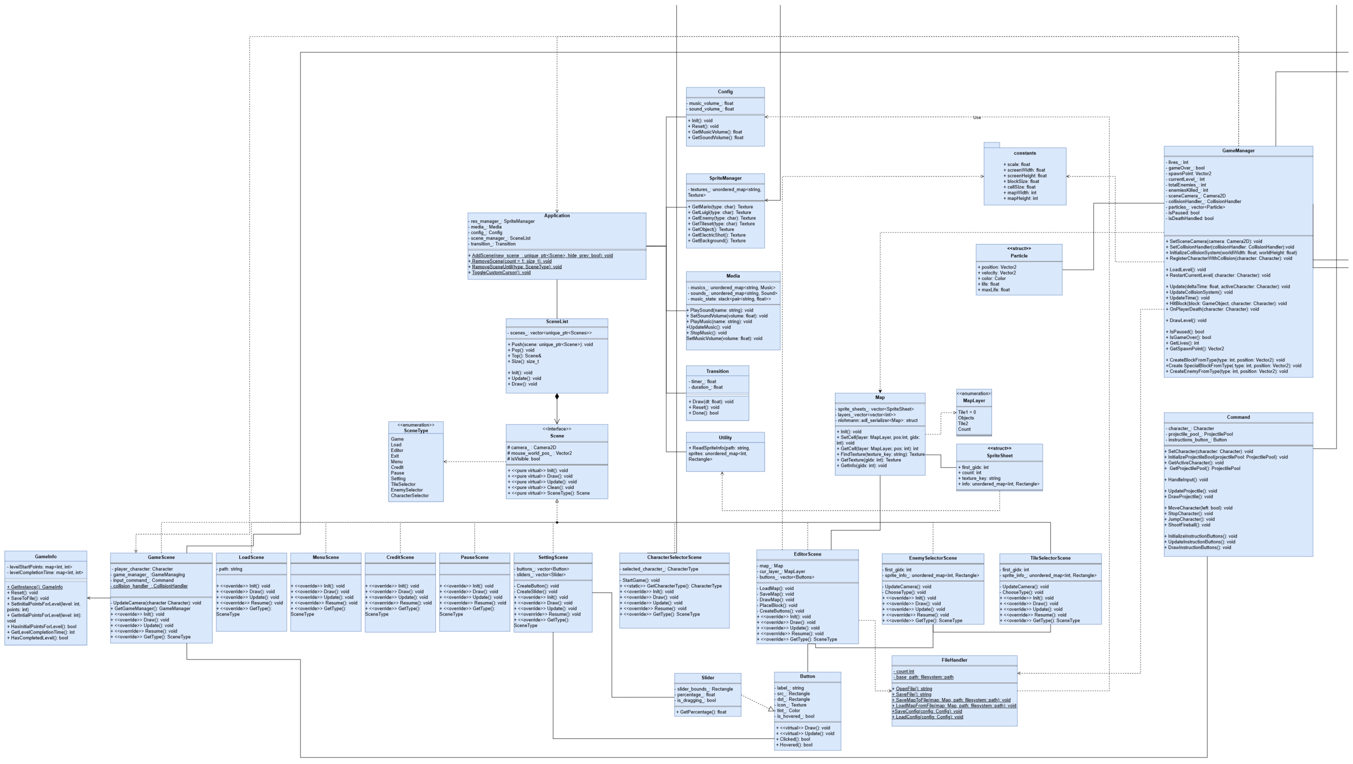


Figure 2: The Core subsystem of the project

1. **Organization:** The Core subsystem handles program state switching and key game logic. Its operation revolves a scene-based approach where a scene stack manages which screen is being shown to the players based on their input. Several other modules supports these "scenes" by supplying them with the required resources or functionalities. Below is a delineation of notable source files in this system and their responsibilities:

   (a) `Scene.cpp`: An interface for scenes. Includes four lifecycle functions: Init - Draw - Update - Clean and a SceneType getter that returns the type of the scene subclass.

(b) `SceneList.cpp`: Manages all scene as a stack. Includes basic stack operations like Push - Pop - Top (get top element) - Size (check size).

(c) `Application.cpp`: A singleton managing the entire game, with access to both the scene list and resources. Can add and remove scenes as well as providing resources via getters.

(d) `GameScene.cpp`: A notable scene which handles running the actual game. This scene coordinates operations of the character, collision handler, and game manager, to ensure everything runs smoothly. This scene also handles level progression, with information saved in another class named `GameInfo.cpp`.

(e) `EditorScene.cpp`: Another notable scene which handles drawing and saving the map. Contains an instance of the `Map` class for I/O purpose, and to other selector scenes for choosing the right block to place in the map. The map is layered, similar to graphical editors like Photoshop, to accommodate for objects with consistent sizes like blocks and inconsistent sizes like enemies.

(f) `Map.cpp, FileHandler.cpp`: Responsible for I/O operations of the game. `Map` deals with serializing/deserializing the map to/from JSON and retrieving textures via blocks' Global ID (GID), while `FileHandler` deals with opening dialog boxes and saving/loading player-made maps/game configurations.

(g) `SpriteManager.cpp, Media.cpp`: Facades managing visual and audio resources of the game, allowing easy retrieval via straightforward getters.

(h) `GameManaging.cpp, Command.cpp`: Handles every aspects of gameplay, including loading, drawing, and updating the level, making calls to the collision handlers and managers of objects and enemies where necessary, and store relevant gameplay data like points, health, and game time. `Command` class maps players input to character and projectile movement and handles also the toggling of the instruction panel.

2. **Adherence to SOLID principles and design patterns implementation:**

(a) **SOLID principles:** The `Core` subsystem follows most SOLID principles, though there is still room for improvement:

- **SRP:** Classes such as `SpriteManager` (resource loading), `Config` (settings), and individual `Scene` subclasses each focus on a single responsibility. However, `GameManaging` is overloaded with unrelated tasks (state, particles, collisions) and would benefit from refactoring.
- **OCP:** The scene-based design allows new `Scene` types to be added without altering existing code. Serialization in `Map` can also be extended easily.
- **LSP:** All `Scene` subclasses and `Command` types respect their base class contracts and are interchangeable.
- **ISP:** Interfaces are concise (e.g., `Scene` only defines `Init`, `Update`, `Draw`, `Resume`, `Type`). Again, `GameManaging` is an exception due to its broad API.
- **DIP:** High-level modules depend on abstractions (`Scene`, `Command`), but `GameManaging` still relies on concrete `CollisionHandler` instead of an interface.

(b) **Design patterns:** The subsystem implements three design patterns: Singleton, Façade, and Factory:

- **Singleton:** `Application::Instance()` follows the singleton pattern to ensure a single global instance. We choose this pattern because at any given point in the game, having more than one Application process will surely lead to conflicting calls and duplicate program states, which will seriously compromise the game's performance.
- **Façade:** `SpriteManager` and `Media` classes encapsulate and simplify access to texture resources, hiding the complexity of loading and managing them. We feel the need to simplify the interface of these managers due to their frequency of use; having to make too many complicated calls to these managers will slow down the development process and make the code looks more messy. The Façade pattern is naturally the best choice we could think of to address this issue.
- **Factory Method:** Methods like `GameManaging::CreateEnemyFromType()` and `CreateBlockFromType()` instantiate objects based on type parameters. We choose this pattern to instantiate new block and enemies because of several benefits: better encapsulation, higher extensibility, and the ease of integrating into bigger systems (including a factory function inside the GameManaging class).

5.2 Collision/Projectile (C/P):

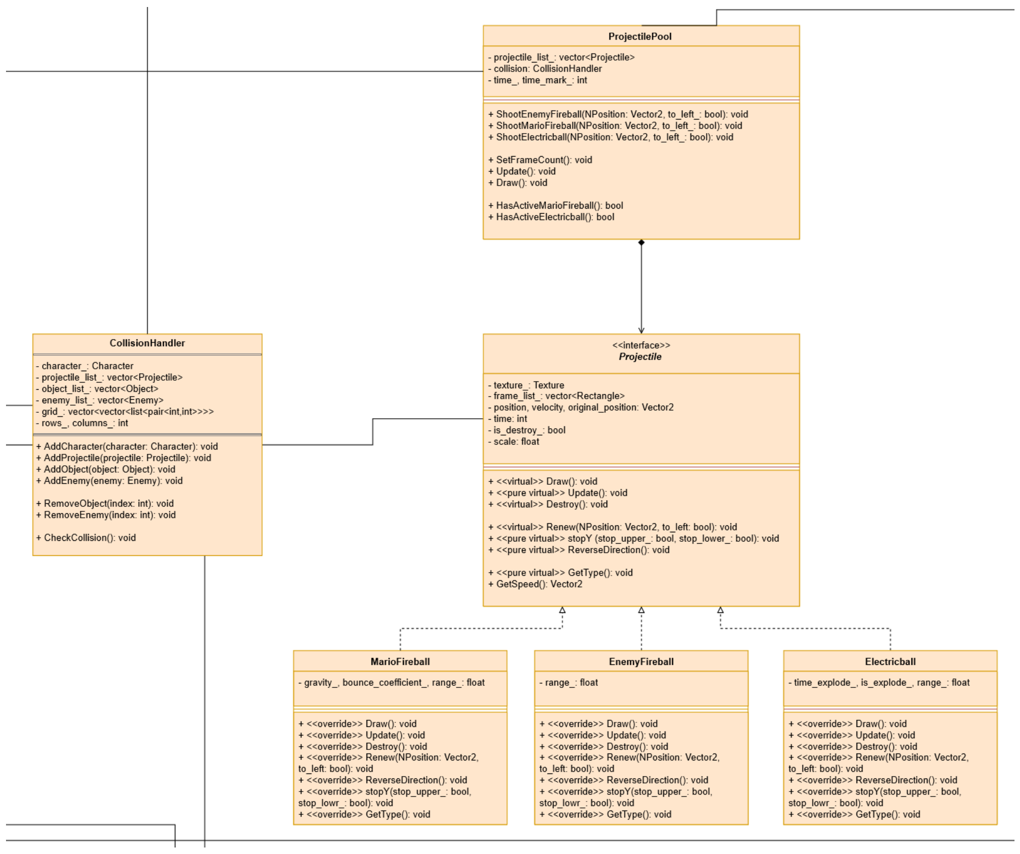

ProjectilePool
- projectile_list_: vector<Projectile>
- collision: CollisionHandler
- time_, time_mark_: int
+ ShootEnemyFireball(NPosition: Vector2, to_left_: bool): void
+ ShootMarioFireball(NPosition: Vector2, to_left_: bool): void
+ ShootElectricball(NPosition: Vector2, to_left_: bool): void
+ SetFrameCount(): void
+ Update(): void
+ Draw(): void
+ HasActiveMarioFireball(): bool
+ HasActiveElectricball(): bool
CollisionHandler
- character_: Character
- projectile_list_: vector<Projectile>
- object_list_: vector<Object>
- enemy_list_: vector<Enemy>
- grid_: vector<vector<list<pair<int,int>>>>
- rows_, columns_: int
+ AddCharacter(character: Character): void
+ AddProjectile(projectile: Projectile): void
+ AddObject(object: Object): void
+ AddEnemy(enemy: Enemy): void
+ RemoveObject(index: int): void
+ RemoveEnemy(index: int): void
+ CheckCollision(): void
<<interface>>
Projectile
- texture_: Texture
- frame_list_: vector<Rectangle>
- position, velocity, original_position: Vector2
- time: int
- is_destroy_: bool
- scale: float
+ <<virtual>> Draw(): void
+ <<pure virtual>> Update(): void
+ <<virtual>> Destroy(): void
+ <<virtual>> Renew(NPosition: Vector2, to_left: bool): void
+ <<pure virtual>> stopY (stop_upper_: bool, stop_lower_: bool): void
+ <<pure virtual>> ReverseDirection(): void
+ <<pure virtual>> GetType(): void
+ GetSpeed(): Vector2
MarioFireball
- gravity_, bounce_coefficient_, range_: float
+ <<override>> Draw(): void
+ <<override>> Update(): void
+ <<override>> Destroy(): void
+ <<override>> Renew(NPosition: Vector2, to_left: bool): void
+ <<override>> ReverseDirection(): void
+ <<override>> stopY(stop_upper_: bool, stop_lowr_: bool): void
+ <<override>> GetType(): void
EnemyFireball
- range_: float
+ <<override>> Draw(): void
+ <<override>> Update(): void
+ <<override>> Destroy(): void
+ <<override>> Renew(NPosition: Vector2, to_left: bool): void
+ <<override>> ReverseDirection(): void
+ <<override>> stopY(stop_upper_: bool, stop_lowr_: bool): void
+ <<override>> GetType(): void
Electricball
- time_explode_, is_explode_, range_: float
+ <<override>> Draw(): void
+ <<override>> Update(): void
+ <<override>> Destroy(): void
+ <<override>> Renew(NPosition: Vector2, to_left: bool): void
+ <<override>> ReverseDirection(): void
+ <<override>> stopY(stop_upper_: bool, stop_lowr_: bool): void
+ <<override>> GetType(): void

Figure 3: The C/P system of the project

1. **Organization:** This subsystem contains two modules: `CollisionHandler` and `Projectile`. Below is a brief overview of their responsibilities, as well as our design choice for keeping them separate from the `Core` subsystem:

   - **Responsibilities:**

     The `Collision` module manages collision detection between the player, projectiles, objects, and enemies via a spatial grid with buckets, i.e. a 2D vector whose entries are lists of pairs. Each pair contains a identifier for the type of object currently in that space and its index within the grid. Entities register themselves, and with each frame the handler updates their grid positions, then performs targeted checks. This design minimizes unnecessary collision checks and integrates tightly with other managers while keeping the collision logic centralized.

The `Projectile` module consists of an interface for the projectiles, implemented by three subclasses for Mario's fireball, Luigi's electric ball (our invention), and Bowser's fireball. These projectiles are managed in an array by the `ProjectilePool` class. Projectiles have three lifecycle methods: Draw - Update - Destroy, along with two utility methods to reverse direction and check y-axis position to let the projectiles bounce when they hit the ground.

- **Design choice:** We model `Collision/Projectile` (C/P) as its own subsystem rather than part of `Core`. This separation is to follow SRP: Core focuses on scenes/state/resources; C/P, on the other hand, works with collision and projectiles, aspects that mainly concern the character/object side of the game but do not belong to any of them. This separation also allows independent testing and easier extension of the C/P subsystem if needed.

2. **Adherence to SOLID principles and design patterns implementation:**

   (a) **SOLID principles:** Overall, the subsystem adheres quite well to SOLID principles, with the exception of some aspects:

   - **SRP:** Projectiles encapsulate their own motion/expiry/rendering; the pool manages lifecycle. However, `CollisionHandler` blends grid management, collision checking, and collision resolution into one class. As improvement, this class would benefit from being refactored into something like: *SpatialGrid* (management), *CollisionDetector* (collision checking), and *InteractionResolvers* (collision resolution).
   - **OCP:** New projectile types can easily be plugged in via `Projectile` polymorphism, but `ProjectilePool` exposes type-specific `Shoot*` methods and `CollisionHandler` still uses large conditionals block which makes addition difficult. Replace with a factory/registry for the `Shoot` methods and collision strategy classes to add behaviors without editing core code.
   - **LSP:** `Projectile` interface ensures all calls go through declared virtuals (avoid relying on subclass-only members).
   - **ISP:** The API of both modules are sufficiently well-designed for a game this scale; however, as the game grows in size, more refactoring needs to be done to separate utility functions from rendering ones (in Projectile) and bookkeeping from resolution (as outlined above in SRP).

- **DIP:** The pool stores `Projectile*`, which is good, but it still depends on concrete `CollisionHandler`; `CollisionHandler` in turn depends on concrete `Character/Enemy/GameObject`. To better follow this principle, we would introduce more abstraction layers,
  e.g. (`ICollisionWorld`, `ICollider` to invert dependencies.

(b) **Design patterns:** The subsystem implements two design patterns: Object Pool and Template Method.

- The ProjectilePool implements the Object Pool pattern, a performance-oriented creational pattern related to but not formally part of the original Gang of Four catalog. It can be viewed as a specialized factory that reuses instances instead of instantiating them afresh. We chose this pattern over Flyweight because the projectiles in our game has too little common attributes to be shared across all instances; instead, each projectile has a determined lifetime, and from what we have read, the Object Pool pattern, with clear acquire/release semantics (`Renew()/Destroy()` in our code), handles this aspect better.
- The Template Method pattern lies inside the `CollisionChecking()` function, in which it allows the function to orchestrate the collision-checking process by calling relevant methods in a specific order (although in the original pattern, these methods would be subclasses that override the function calls). We feel this is a rather straightforward choice given the nature of the function call order and existing methods.
- As possible improvement, after refactoring this class, we would like to use the Observer pattern to notify relevant classes in Character and Enemy/Object subsystems when a collision event happens, in order to let them process those events themselves, instead of letting the Collision subsystem implementing these side-effects.

5.3 Character:

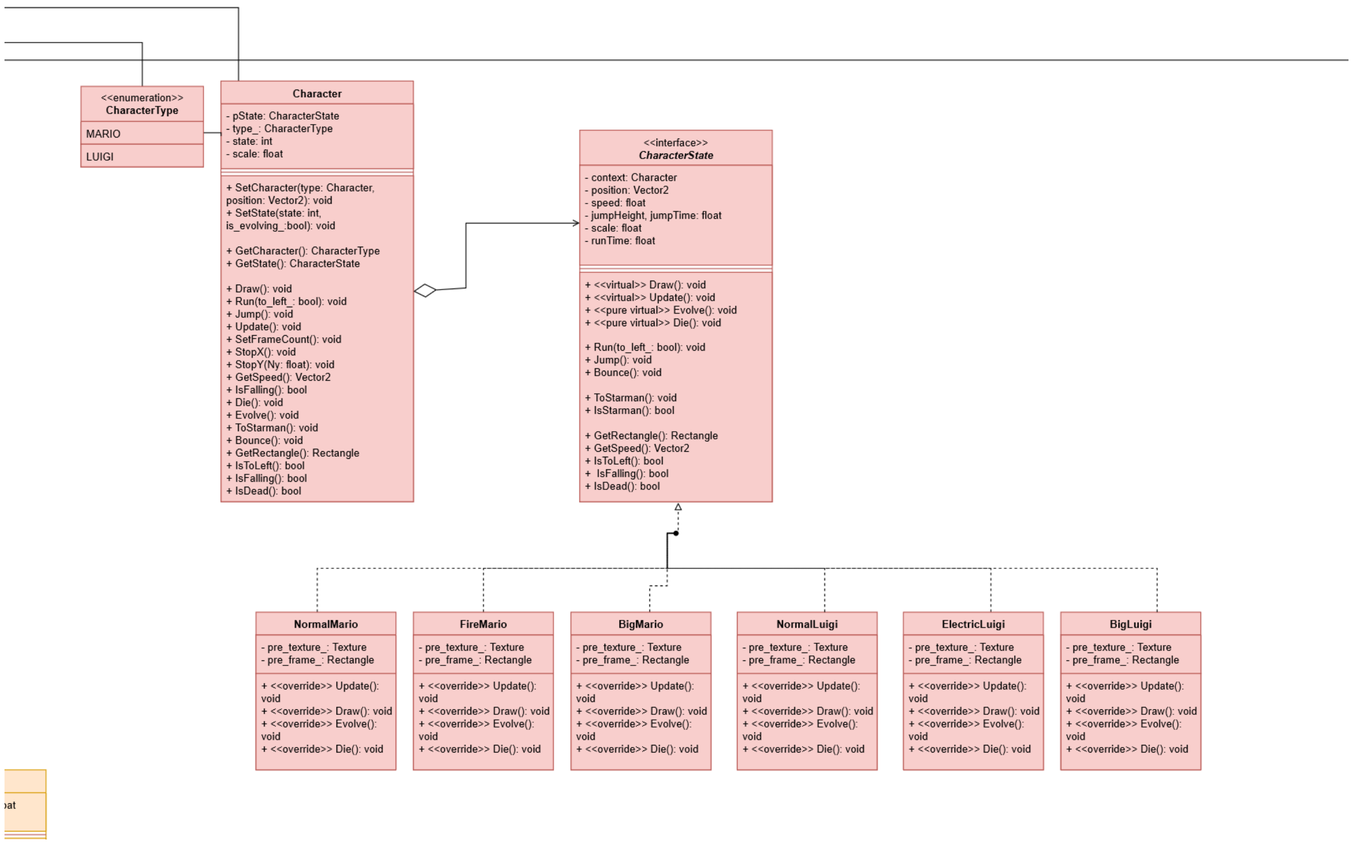


Figure 4: Character subsystem in the project

1. **Organization:** The `Character` subsystem consists of two modules. The first is a character class that creates and manages instances of playable characters Mario and Luigi. This module serves as the context and stores a reference to another interface named `CharacterState` for switching character states when a power-up is consumed (Normal, Fire/Electric, Starman). The six states (three for each character) are implemented by six subclasses.

2. **Adherence SOLID principles and design patterns implementation:**

   (a) **SOLID principles:** Overall, this subsystem adheres quite well to SOLID principles:

   - **SRP:** `Character` delegates gameplay logic to `CharacterState` subclasses, which encapsulate movement, physics, and state-specific rendering; this is desired behavior. However, `CharacterState` mixes physics, animation, and some game-rule logic, which could be refactored as program size grows.

- **OCP:** Well supported via the State pattern — adding a new form (e.g., IceMario) requires only a new `CharacterState` subclass without modifying `Character`. Similarly, `EnemyManager` and `ObjectManager` can interact with `Character` polymorphically.
- **LSP:** All `CharacterState` subclasses honor the base contract, so `Character` can swap them at runtime without breaking client code.
- **ISP:** The `CharacterState` API is reasonably focused, though it contains operations (`Evolve`, `ToStarman`) that not all states may use; minor refactoring could split pure movement/physics from power-up logic.
- **DIP:** `Character` class depends on the abstraction `CharacterState`, as part of the State pattern.

(b) **Design patterns:**

The subsystem implements the State pattern, by letting the base `Character` class be the context and delegate state-switching logic and state-specific behavior to `CharacterState` and its subclasses instead. Given the number of states of both characters, we feel that this is a natural choice.

5.4 Enemy:

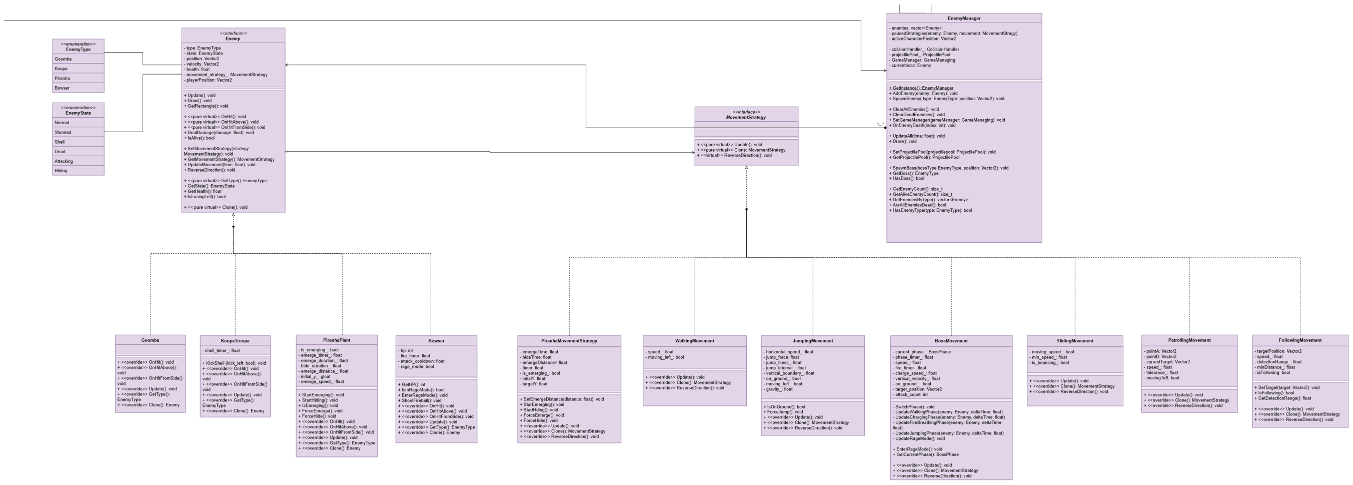


Figure 5: The Enemy subsystem in the project

1. **Organization:** This subsystem consists of one interface serving as the context, which relies on a enum class `EnemyState` that declares state names to handle state-switching and implement state-

specific behaviors in subclasses. Movement strategies are processed separately in another class through the Strategy pattern. All enemies are managed in an array by a `EnemyManager` class.

2. **SOLID principle adherence and design patterns implementation:**

   (a) **SOLID principles:**

   - **SRP:** separation of concerns (instantiation from management) is desired behavior. However, `EnemyManager` and `Enemy` have multiple responsibilities (e.g. `EnemyManager` has functions to specificall spawn the boss even though it is supposed to handle all enemy types); split into focused components.
   - **OCP:** Addition and modification through the enum class remains unintuitive; it is better if the State pattern is followed, which involves turn the enum class into an interface and let the subclasses of `Enemy` depend on it for state logic.
   - **LSP:** Inheritance chains (`Enemy` subclasses) respects this principle.
   - **ISP:** Enemy interface is fairly broad, including physics, combat, and rendering at the same time. This class can benefit from being refactored into smaller interfaces.

   DIP Classes in the subsystem should depend more on abstractions, instead of being locked in concrete dependencies or holding raw pointers.

   (b) **Design pattern:**

   - Class `EnemyManager` implements the Singleton patterns and acts as a process-wide singleton to centralize enemy. The uniqueness of managers is necessary in making the program run smoothly.
   - Through the Façade pattern, `EnemyManager` provides a simplified façade over many object classes (add/update/draw); keep the façade thin after splitting responsibilities. Given the frequency of use of manager classes, we feel using the Façade pattern would help make function calls more intuitive, thus speeding up development.
   - The `Enemy` class follows a weak version of the State pattern, where the state interface is instead an enum class, and the context's subclasses directly handles the state logic. We chose this approach at first since we think delegating all state logic, which is not much,

for the context's subclasses to implement would simplify the code. This class, however, does implement quite well the Prototype pattern through its pure virtual `Clone()` method, which we feel is suitable as it nicely supports duplication and pooling of enemies without costing performance.

- The `Movement Strategy` class implements the Strategy pattern, by delegating strategies through an interface and its set of subclasses. Since there are many strategies that only concern one aspect of the enemy - movement (hence rendering State pattern unsuitable), and is subject to multiple changes during runtime, we feel the choice of the Strategy pattern is logical.

5.5 Object:

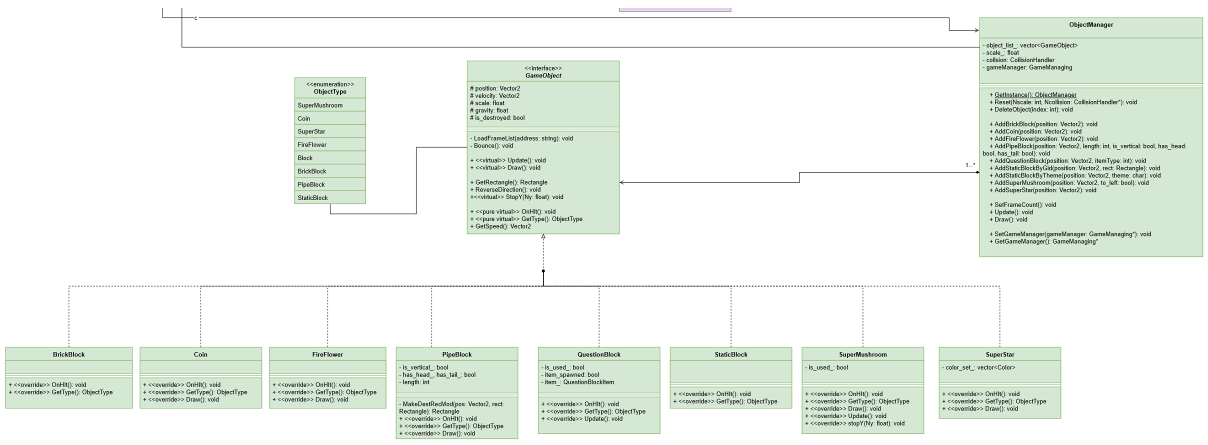

ObjectManager
- object_list_: vector<GameObject>
- scale_: float
- collsion: CollisionHandler
- gameManager: GameManaging
+ GetInstance(): ObjectManager
+ Reset(Nscale: int, Ncollision: CollisionHandler*): void
+ DeleteObject(index: int): void
+ AddBrickBlock(position: Vector2): void
+ AddCoin(position: Vector2): void
+ AddFireFlower(position: Vector2): void
+ AddPipeBlock(position: Vector2, length: int, is_vertical: bool, has_head: bool, has_tail: bool): void
+ AddQuestionBlock(position: Vector2, itemType: int): void
+ AddStaticBlockByGid(position: Vector2, rect: Rectangle): void
+ AddStaticBlockByTheme(position: Vector2, theme: char): void
+ AddSuperMushroom(position: Vector2, to_left: bool): void
+ AddSuperStar(position: Vector2): void
+ SetFrameCount(): void
+ Update(): void
+ Draw(): void
+ SetGameManager(gameManager: GameManaging*): void
+ GetGameManager(): GameManaging*
1...*
<<enumeration>>
ObjectType
SuperMushroom
Coin
SuperStar
FireFlower
Block
BrickBlock
PipeBlock
StaticBlock
<<interface>>
GameObject
# position: Vector2
# velocity: Vector2
# scale: float
# gravity: float
# is_destroyed: bool
- LoadFrameList(address: string): void
- Bounce(): void
+ <<virtual>> Update(): void
+ <<virtual>> Draw(): void
+ GetRectangle(): Rectangle
+ ReverseDirection(): void
+<<virtual>> StopY(Ny: float): void
+ <<pure virtual>> OnHit(): void
+ <<pure virtual>> GetType(): ObjectType
+ GetSpeed(): Vector2
BrickBlock
+ <<override>> OnHIt(): void
+ <<override>> GetType(): ObjectType
Coin
+ <<override>> OnHIt(): void
+ <<override>> GetType(): ObjectType
+ <<override>> Draw(): void
FireFlower
+ <<override>> OnHIt(): void
+ <<override>> GetType(): ObjectType
+ <<override>> Draw(): void
PipeBlock
- is_vertical_: bool
- has_head_, has_tail_: bool
- length: int
- MakeDestRecMod(pos: Vector2, rect: Rectangle): Rectangle
+ <<override>> OnHIt(): void
+ <<override>> GetType(): ObjectType
+ <<override>> Draw(): void
QuestionBlock
- is_used_: bool
- item_spawned: bool
- item_: QuestionBlockItem
+ <<override>> OnHIt(): void
+ <<override>> GetType(): ObjectType
+ <<override>> Update(): void
StaticBlock
+ <<override>> OnHIt(): void
+ <<override>> GetType(): ObjectType
SuperMushroom
- is_used_: bool
+ <<override>> OnHIt(): void
+ <<override>> GetType(): ObjectType
+ <<override>> Draw(): void
+ <<override>> Update(): void
+ <<override>> stopY(Ny: float): void
SuperStar
- color_set_: vector<Color>
+ <<override>> OnHIt(): void
+ <<override>> GetType(): ObjectType
+ <<override>> Draw(): void

Figure 6: The Object subsystem in the project

1. **Organization:** The Object subsystem consists of an interface declaring general properties of game objects, including items and blocks, whose specific implementation are done by 8 subclasses (4 items and 4 types of block); all objects are managed in an array by another class `ObjectManager`.

2. **SOLID principle adherence and design patterns implementation:**

   (a) **SOLID principles:**

   - **SRP:** Again, `ObjectManager` class mixes creation, ownership, update/draw, collision wiring, and game callbacks logic. It would benefit from being split into smaller classes: `ObjectFactory` (instantiation using Factory pattern), `ObjectSystem` (update/draw).

- **OCP:** `Add` methods are hardcoded, while Factory logic is deferred upstream to `GameManaging` class. Although the presence of a Factory is good, it would be better to move its place to live inside the `Object` subsystem instead.
- **LSP:** Subclasses (`BrickBlock`, `Coin`, `FireFlower`, ...) override base `GameObject` operations consistently (e.g., `OnHit`, `Draw/Update` when needed), so substitutability holds.
- **ISP:** `GameObject` interface is broad (`GetSpeed`, `ReverseDirection`, `StopY`), just like that in the Enemy subsystem, and forces static objects (e.g., `StaticBlock`, `QuestionBlock`) to depend on irrelevant members (blocks cannot move or change direction).
- **DIP:** `ObjectManager` depends on concretes (`CollisionHandler`, `GameManaging`); invert dependency via interfaces so high-level logic depends on abstractions.

(b) **Design pattern:**

- Class `ObjectManager` implements the Singleton patterns and acts as a process-wide singleton to centralize objects. Again, the uniqueness of managers is necessary in making the program run smoothly.
- Through the Façade pattern, `ObjectManager` provides a simplified façade over many object classes (add/update/draw); keep the façade thin after splitting responsibilities. Given the frequency of use of manager classes, we feel using the Façade pattern would help make function calls more intuitive, thus speeding up development.
- As improvement, we would like to implement the Observer pattern to emit events (e.g., “object collected”, “block hit”) to `IGameEvents` instead of calling game systems directly; this decouples objects from scoring/timers.

6 Technical difficulties:

6.1 Handling I/O of sprites and maps

- Description: Choosing the suitable format for saving and loading resources is a critical part of the game's design. Additionally, bridging the processes of reading and drawing the image resources poses serious challenges regarding correctness and efficiency, requiring multiple substantial revision before an desirable solution can be found. Storing and managing the resources in one centralized system is similarly an onerous task and requires numerous iterations.
- Approach: Spritesheets and icons are accessed using a separate text file which contains the positions of each sprite/icon. Maps are stored as json files, read and written using custom serializers and deserializers. Textures are pre-loaded into a manager then retrieved using getters.

6.2 Designing the collision handler

- Description: Collision handler is a central components to a 2D platformer whose implementation details are also novel to us novice programmers, therefore requiring great care and effort towards developing a functional prototype.
- Impact: Not being able to develop these system means the game is unplayable, as objects and characters simply do not interact with each other.
- Approach: We chose a grid-based approach to implement the collision handler. Collision handler partitions entities into grid cells each frame, updating positions and detecting overlaps for characters, enemies, and projectiles against blocks and hazards. It then performs conflict resolution, calling relevant methods to add/update/remove objects it currently controls.

6.3 Designing the map editor

- Description: several key aspects of the map editor scene, such as choosing the correct layers, drawing and erasing blocks/objects, or traversing the map are completely new and had to be designed from scratch.

- Impact: Being able to produce a functional map editor adds an sizable bonus to the game's functionality score. Furthermore, it can also be used to make stages for the game, reducing external dependencies.

- Approach: We chose a layer-based approach to the map editor, modeling graphical editors such as Photoshop. This allows the separation of constant- and non=constant-sized objects into different layers, and ensure correct order of appearance when drawing (e.g. the piranha plant must be drawn behind a pipe). Map traversing and layer-switching is done via hotkeys and mouse buttons, which makes controls more intuitive.

6.4 Debugging core gameplay functions

- Description: In our `GameManaging` class, several key functions such as `LoadLevel()`, `DrawLevel()`, and `Update()` are very long and complex pieces of codes that requires calling multiple subjects in the correct order and checking appropriate conditions to avoid throwing exceptions (e.g. if player dies, check player's remainging health before calling reset). We also encountered issues where the game threw after doing certain things (dying, shooting too many projectiless).

- Impact: Solving these problems means making the game playable and enjoyable, so it was naturally a top priority for us.

- Approach: We resolved the complexity of the large functions in `GameManaging` by modularizing complex logic into smaller helper methods and orchestrating them in the correct order, though this came at the cost of overpopulating the class with irrelevant methods. For debugging, we tested extensively by checking the call stack and using try-throw-catch to throw detailed messages of the instance of error (e.g. "object no. 1334 is out of range."). For the random throw issues, we decided to completely remove the current game scene upon restarting the game then add a new one.

7 Limitations and future improvements:

7.1 Limitations:

- Gameplay is still basic with few variation in playable characters, environments, power-ups, and enemies.
- Map editor can benefit from a redesign in UI and controls.
- Overall codebase is clunky and requires refactoring for more robustness.

7.2 Future improvements:

- Extend gameplay by adding more characters, environments, powerup, and enemies. Support for two players and other game modes is also desired.
- Redesign the UI for key components in the game to be intuitive and responsive to user input. Customizing a unique set of fonts, icons, and buttons for the game is also desired.
- Refactor the code, especially GameManaging class, and reorganize or modify resources to improve organization and consistency in quality.

8 Demonstration:

Below are some screenshots of the game:

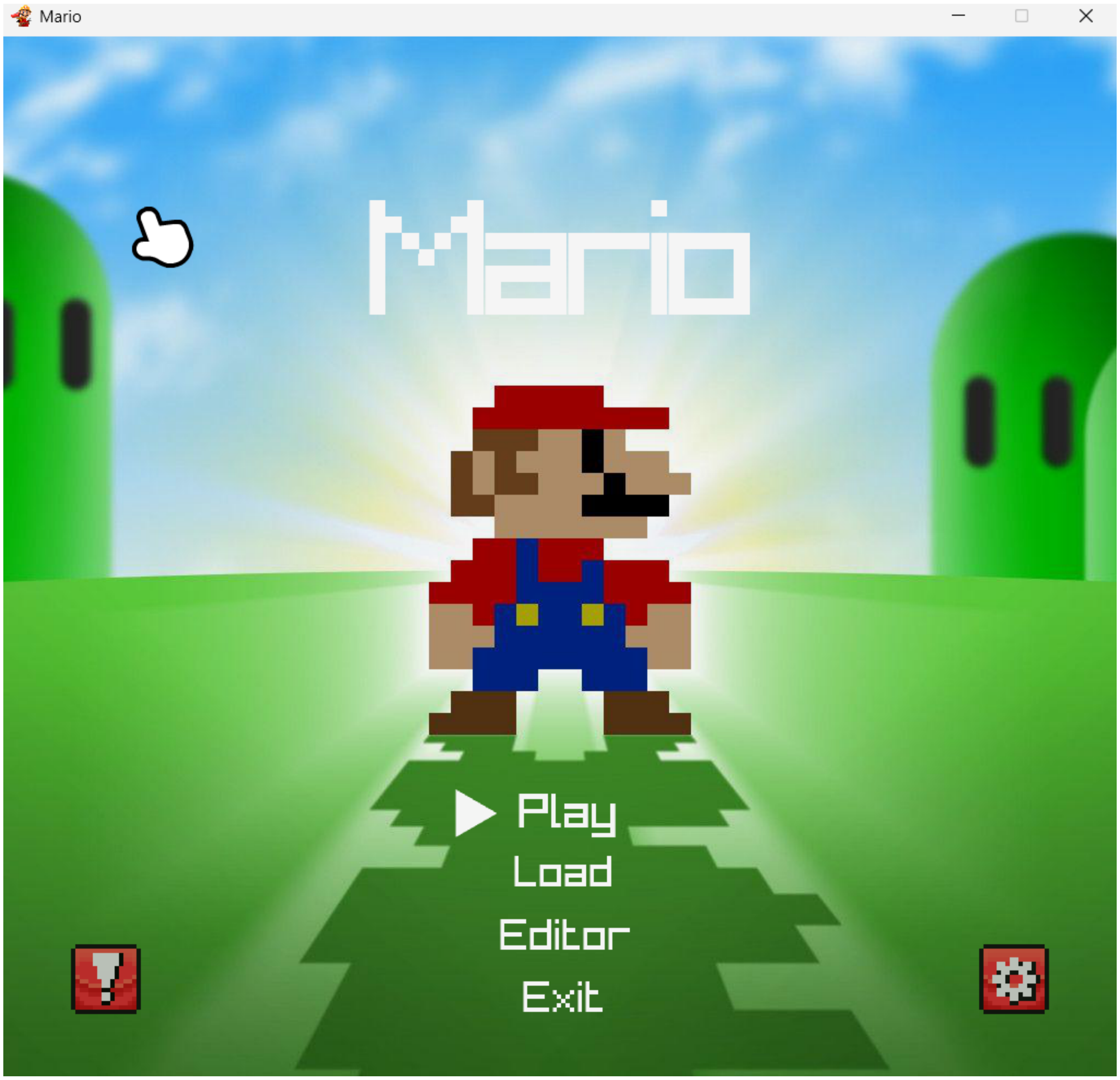


Figure 7: The game's main menu

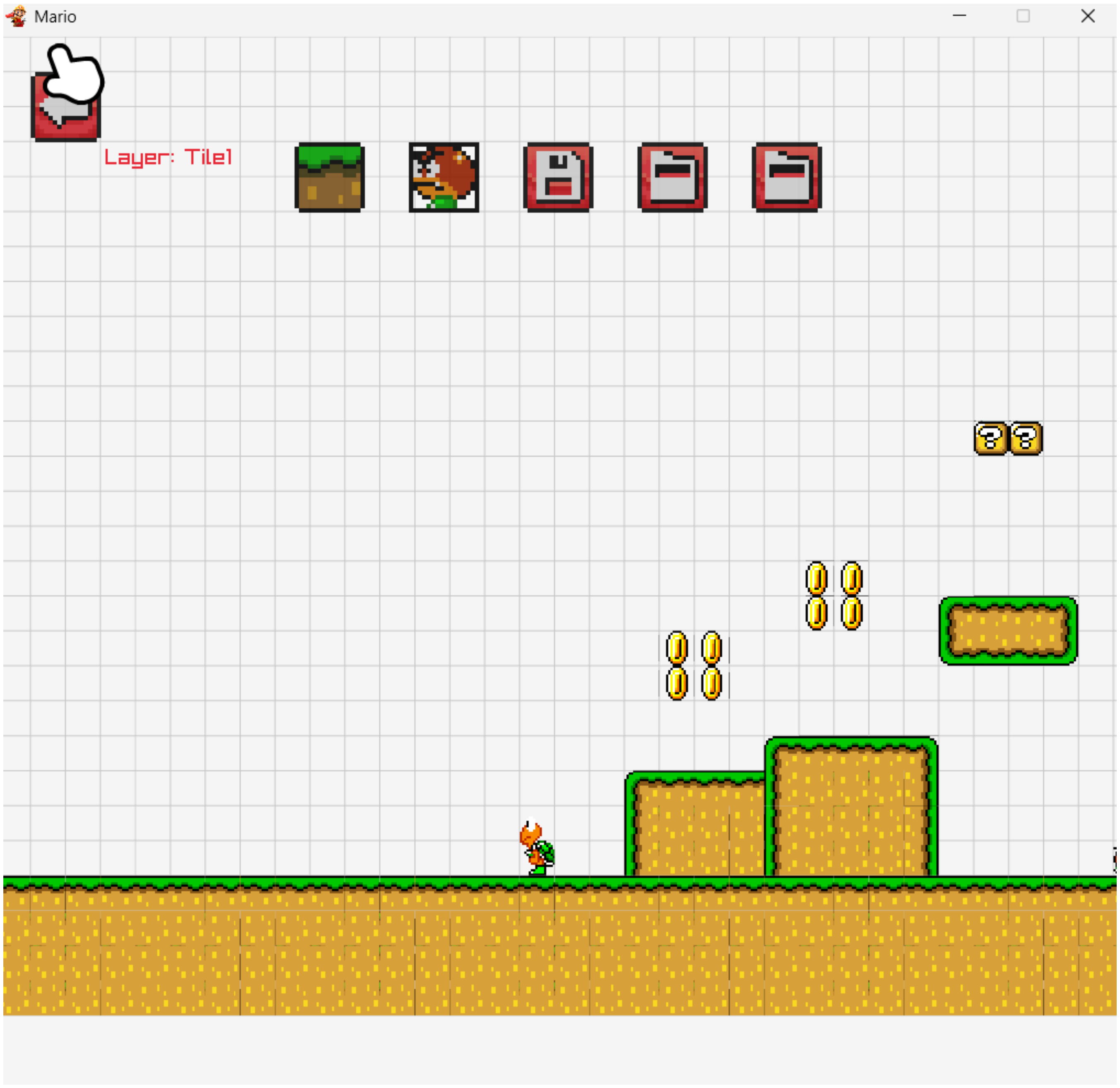


Figure 8: The game's editor scene, with a premade map

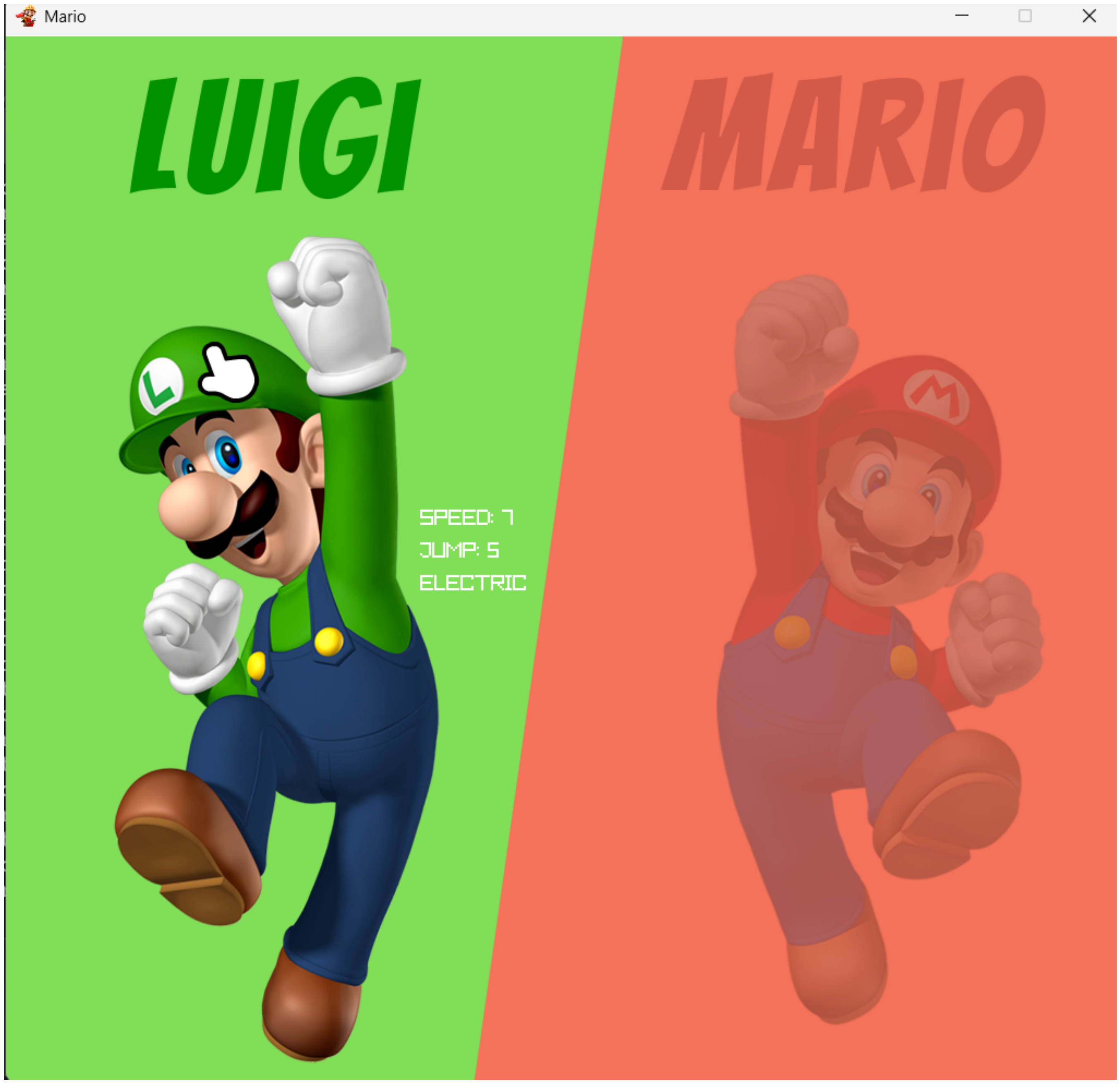


Figure 9: The game's character selection screen, before the level starts

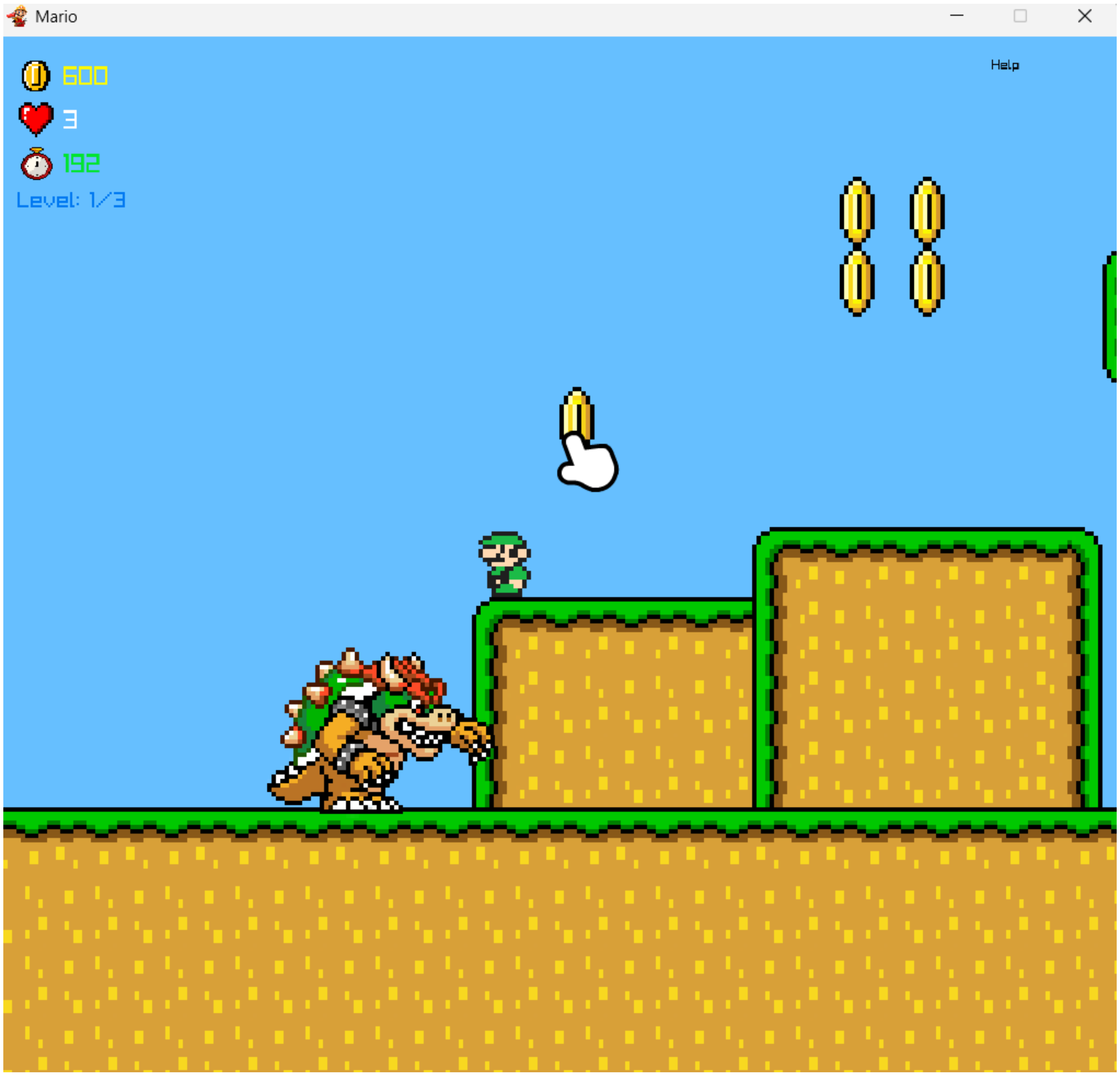


Figure 10: The game's first level, with Bowser for demonstration purpose

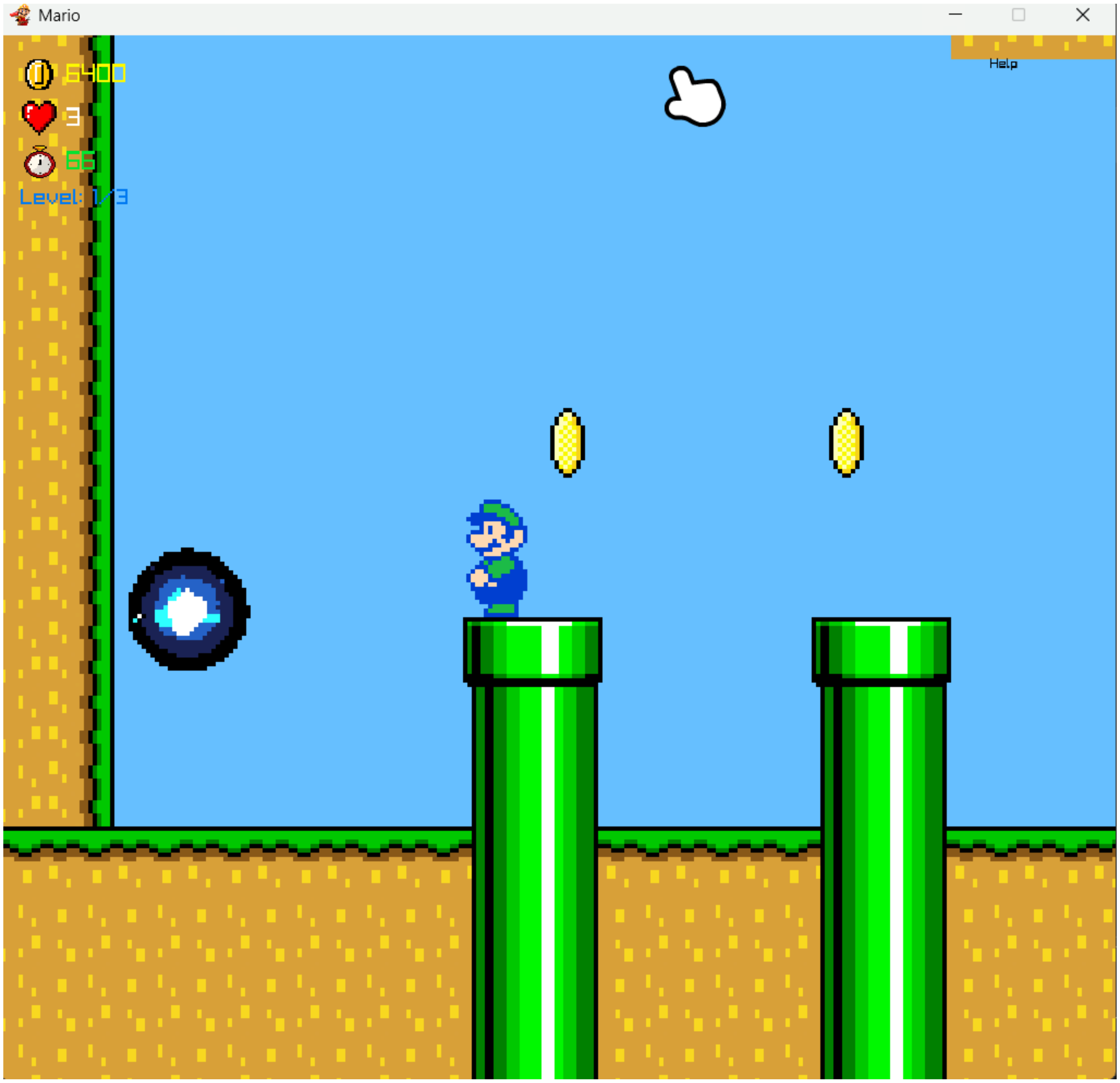
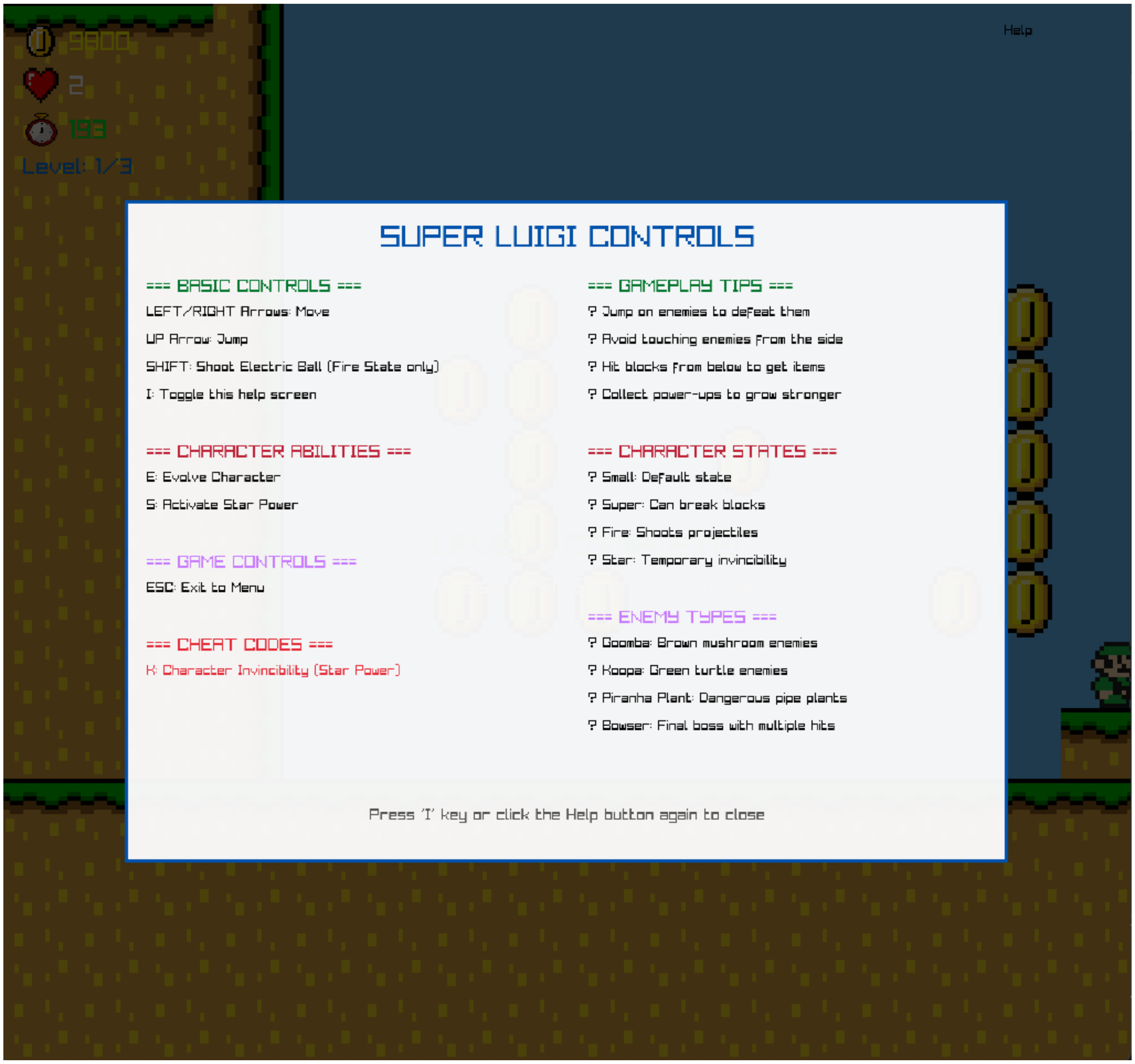


Figure 11: Electric ball fired by Luigi. Both Electric Luigi and the electric ball are our invention.



SUPER LUIGI CONTROLS

=== BASIC CONTROLS ===
LEFT/RIGHT Arrows: Move
UP Arrow: Jump
SHIFT: Shoot Electric Ball (Fire State only)
I: Toggle this help screen

=== CHARACTER ABILITIES ===
E: Evolve Character
S: Activate Star Power

=== GAME CONTROLS ===
ESC: Exit to Menu

=== CHEAT CODES ===
K: Character Invincibility (Star Power)

=== GAMEPLAY TIPS ===
? Jump on enemies to defeat them
? Avoid touching enemies from the side
? Hit blocks from below to get items
? Collect power-ups to grow stronger

=== CHARACTER STATES ===
? Small: Default state
? Super: Can break blocks
? Fire: Shoots projectiles
? Star: Temporary invincibility

=== ENEMY TYPES ===
? Goomba: Brown mushroom enemies
? Koopa: Green turtle enemies
? Piranha Plant: Dangerous pipe plants
? Bowser: Final boss with multiple hits

Press 'I' key or click the Help button again to close

Figure 12: A table containing instructions for the game, toggle during gameplay by pressing ”I”

9 Work division:

9.1 Team members:

1. Bao, Nguyen Chi - 24125025
2. Quan, Pham Nguyen Minh - 24125041
3. Khoa, Phan Anh - 24125092
4. Lac, Quach Thien - 24125092
5. Duy, Nguyen Quang - 18125048

9.2 Responsibilities

1. Bao: Implement the following parts: Character, Enemy (Bowser), Items, Projectiles, Collision Handler.
2. Quan: Implement the following parts: Menu/Editor/Character-Enemy-Tile Selector Scenes, Map Saving/Loading. Also prepare audio resources and manage the Github repository.
3. Khoa: Implement the following parts: Game Manager, Commands, Enemy (Minions).
4. Lac: Implement the following parts: File Handler, Load Scene, Level Design. Also prepare image resources and write this report.
5. Duy: Implement the following parts: Enemy, Objects.

9.3 Collaboration tools:

Messenger, Github.

10 Demo Video

Youtube video

11 References:

- LaTeX by Quan, Tran Hoang
- Refactoring Guru
- SOLID Principle article on Wikipedia