

**Main Tools used:**

* Raylib
* C++
* Visual Studio Code
* Assistance from LLM
* YouTube Tutorials
* Raylib Cheat sheet

**Classes Summarized:**

**Entity.cpp**

**Purpose**: Defines shared behaviors and attributes for game entities like players or enemies.

**Key Elements**:

* Entity::Entity() – Default constructor.
* operator= – Custom assignment; handles deep copy of texture.
* ~Entity() – Unloads texture on destruction.
* hitbox() – Returns the entity's bounding box.
* get\_attack\_hitbox() – Calculates melee attack area based on facing direction.
* get\_wall\_coll() – Used to detect wall collisions ahead of the entity.
* DrawAtkHitbox() – Debug tool to draw the attack range box.
* gravity\_on() – Applies gravity; snaps to ground if fallen too far.
* get\_jumphitbox() – Small area below the entity used for jump/ground detection.

**In essence**: A base class managing an entity's position, physics, collision, and attack range.

**Player.cpp**

**Purpose:** Handles the player's behavior, rendering, movement, and collision.

**Key Elements:**

* Player::Player() – Loads sprite, sets movement attributes and spawn point.
* Draw() – Draws the player sprite and name above them.
* update() – Handles:
  + Gravity and ground detection.
  + Platform and wall collision.
  + Movement logic (including sprinting and jumping).
  + Attacking breakable platforms (Z key).
* light\_attack() – If in range and cooldown allows, damages breakable platforms.
* setpositon() – Moves the player to a specific location.

**In essence:** The player character's input, physics, animation, and basic combat behavior.

**Platform.cpp**  
**Purpose**: Defines platform and wall objects used in the game world. Handles rendering, collision bounds, and breakability.

**Key Elements**:

* Platform::Platform(...) – Overloaded constructors to create:
  + Static floor or wall
  + Custom-sized platforms
  + Breakable platforms (with different textures)
* Draw() – Renders the platform texture at the correct location.
* GetHitbox() – Returns the platform’s collision rectangle.
* update() – If breakable and health < 0, moves the platform offscreen (simulating destruction).
* ~Platform() – Unloads texture to free memory.

**In essence**: Represents all static or breakable surfaces in the game; handles drawing, collision bounds, and destruction logic.

**Room.cpp**

**Purpose:** Represents a single room in the dungeon level. Handles platform/star setup, rendering, and interaction logic.

**Key Elements:**

* Room::Room(...) – Initializes the room with a background texture, player spawn location, and the exit zone.
* AddPlatform(...) and AddStar(...) – Add platforms and collectible skulls (stars) to the room.
* Draw(Player&) –
  + Draws the background and exit.
  + Updates/draws platforms and skulls.
  + Checks collision between player and stars (collects them).
  + Checks if player is at the exit and has collected all stars → if so, marks level complete.
* get\_playerspawn() – Returns player spawn location.
* ~Room() – Unloads the background texture.

**In essence:** One level's contents and behavior (platforms, collectibles, background, exit logic).

**Dungeon.cpp**

**Purpose:** Manages the sequence of rooms as a queue.

**Key Elements:**

* Dungeon() – Initializes an empty dungeon (no rooms).
* Enqueue(Room\*) – Adds a room to the end of the dungeon queue.
* ProceedToNextRoom() – Moves to the next room.
* GetCurrentRoom() – Returns the currently active room.
* ~Dungeon() – Cleans up dynamically allocated room memory.

**In essence:** The dungeon is a sequence (linked list-like) of rooms that the player progresses through.

**Star.cpp**

**Purpose**: Represents a collectible or game object (like a star or skull) with basic behavior.

**Key Elements**:

* Star::Star(x, y) – Initializes position and loads texture.
* Draw() – Renders the object to the screen.
* update() – Moves the object offscreen (likely after collection).
* GetHitbox() – Returns its bounding box for collision.
* ~Star() – Frees texture memory.

**In essence**: A simple game object with draw, collision, and removal behavior.

**DSA, Algorithms & Advanced C++ Features included:**

• Linked Lists: Rooms serving as nodes

• Queue: Traversal of Nodes via Dungeon class

• Sorting Algorithms: Bubble sort for leaderboard

• Exception Handling: Input Validation for username

• STL Containers: Use of Vectors

**Given More time I would Add:**

* Use Entity class to make enemies
* Add more levels
* Use file handling to store scores
* Add animations
* Add more sfx
* Add Items and Buffs