HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

KHOA CÔNG NGHỆ THÔNG TIN

---------- ☙🕮❧ ----------

BÁO CÁO BÀI TẬP LỚN

MÔN HỌC: LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

Tên đề tài : Thiết kế game RPG retro

Nhóm môn học: 16

Nhóm BTL: 14

Thành viên: B23DCCN130 – Lưu Đức Thành Đạt

B23DCCN233 – Vũ Hải Dương

B23DCCN942 – Đỗ Minh Vũ

Hà Nội năm 2025

Thành viên	Nhiệm vụ	Tỉ lệ đóng góp(%)
Lưu Đức Thành Đạt	Code:combat, ai.movement, weapons, WorldObject, monster, Thiết kế GamePlay, xuất file exe, Báo cáo	45% (35% code + 10% báo cáo)
Vũ Hải Dương	Code:interact, gameloop, một vài object, collision, 1 số tool, object data , player, âm thanh, quản lý file map và render map,Entity và thuộc tính, entity manager, ui message. Thiết kế map	25 % (code)
Đỗ Minh Vũ	Code:UI, dialogue, save game, NPC interact, check collision Thiết kế âm nhạc Báo cáo	30%(20 % code +10% báo cáo)

Contents

Danh mục bảng

LỜI CẢM ƠN

Lời đầu tiên, chúng em xin gửi lời cảm ơn đến Học viện Công nghệ Bưu chính Viễn thông đã tạo điều kiện để chúng em được học tập và rèn luyện trong môn Lập trình Hướng đối tượng.

Đặc biệt, chúng em xin trân trọng cảm ơn thầy Nguyễn Mạnh Sơn, giảng viên bộ môn, người đã tận tình giảng dạy, hướng dẫn và truyền đạt cho chúng em những kiến thức bổ ích và thiết thực trong suốt quá trình học tập. Sự nhiệt huyết và phương pháp giảng dạy của thầy giúp chúng em hiểu rõ hơn ý nghĩa cốt lõi của lập trình hướng đối tượng: không chỉ viết mã để chương trình chạy được, mà quan trọng hơn là xây dựng cấu trúc mã nguồn khoa học, có thể mở rộng, dễ duy trì và tái sử dụng lâu dài.

Môn học có lượng kiến thức tương đối lớn và yêu cầu khả năng tư duy logic cao. Trong quá trình tiếp thu và hoàn thiện bài tập lớn, mặc dù đã cố gắng nhưng do kiến thức và kinh nghiệm thực tiễn còn hạn chế, bài báo cáo chắc chắn khó tránh khỏi thiếu sót. Chúng em rất mong nhận được những góp ý của thầy để có thể hoàn thiện hơn trong những lần làm bài sau.

Một lần nữa, chúng em xin chân thành cảm ơn thầy và kính chúc thầy luôn mạnh khỏe, công tác tốt.

CHƯƠNG 1. GIỚI THIỆU BÀI TOÁN

1.1. Giới thiệu chung

Đề tài nhóm em xây dựng một trò chơi 2D theo phong cách RPG phiêu lưu – chiến đấu, được lập trình bằng Java theo mô hình lập trình hướng đối tượng (OOP).
Game gồm các chức năng chính:

- Điều khiển nhân vật theo 4 hướng
- Hệ thống bản đồ nhiều map
- Quái vật với AI di chuyển (Wander, Chase)
- Hệ thống combat, chỉ số nhân vật, sát thương
- Hệ thống vật phẩm, object tương tác
- NPC và hội thoại
- Lưu – tải game
- UI, menu, hiệu ứng

Game mang tính chất học thuật, dùng để thực hành kiến thức OOP: đóng gói – kế thừa – đa hình – trừu tượng.

1.2. Lý do chọn đề tài

- Tính thực tế cao, sát với lập trình game chuyên nghiệp
- Giúp hiểu rõ cách tổ chức project lớn bằng Java
- Ứng dụng nhiều design pattern: Singleton, Factory, Strategy, State Machine
- Dễ mở rộng, cập nhật, sửa lỗi → phù hợp với tiêu chí "OOP sạch, dễ debug" mà thầy yêu cầu

1.3. Mục tiêu của dự án

- Xây dựng game có gameplay hoàn chỉnh
- Áp dụng đúng tư duy OOP trong toàn bộ project
- Tách module rõ ràng, dễ bảo trì
- Tránh viết code spaghetti
- Tạo sản phẩm có thể chạy độc lập (file .exe)

CHƯƠNG 2. KHẢO SÁT & PHÂN TÍCH NGHIỆP VỤ

2.1. Lối chơi (Gameplay)

- Người chơi điều khiển nhân vật chính di chuyển trên bản đồ
- Gặp quái: quái phát hiện, đuổi theo, tấn công theo frame timing
- Người chơi có thể né, phản đòn, tấn công
- Trên bản đồ có vật phẩm (chìa khóa, rương, potion), NPC, cổng dịch chuyển
- Qua map → đánh boss → hoàn thành

2.2. Đối tượng trong game

- Player
- Monster (nhiều loại, AI khác nhau)
- WorldObject (vật phẩm, hộp, rương…)
- NPC
- TileMap

2.3. Luồng hoạt động chính

1. Load map
2. Load quái, vật phẩm
3. Game loop chạy update → render
4. Người chơi tương tác → combat → di chuyển map
5. Lưu tiến trình khi thoát game

CHƯƠNG 3. CÔNG NGHỆ VÀ KỸ THUẬT SỬ DỤNG

3.1. Ngôn ngữ – Công cụ

- Java 24
- IDE: IntelliJ IDEA , NetBeans
- Graphics2D (java.awt)
- Gson (save/load dữ liệu)
- Tiled editor (thiết kế map)

3.2. Các design pattern sử dụng

Singleton

- GamePanel
- SoundManager
- GameStateManager

Factory Pattern

- Tạo quái theo loại (MonsterFactory)
- Tạo object (ObjectFactory)

Strategy Pattern

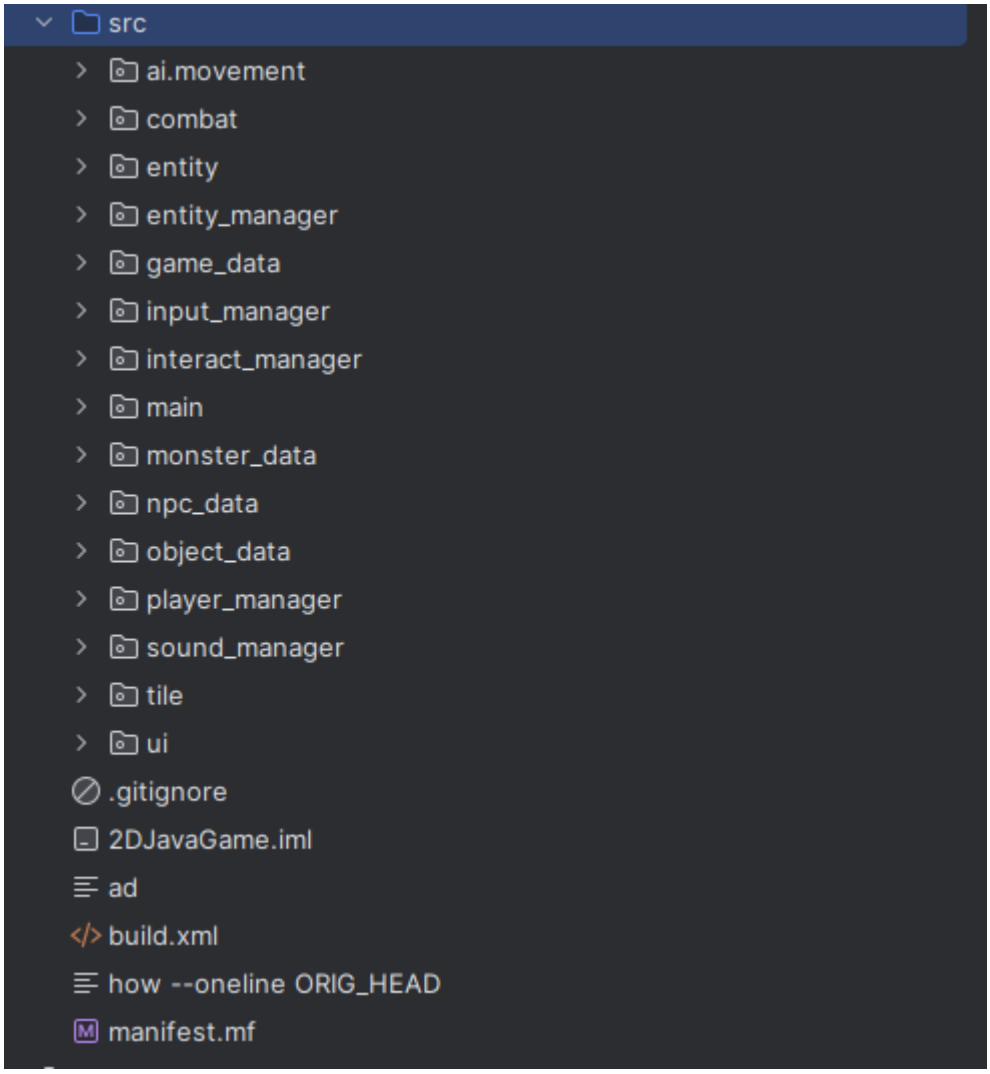
- AI movement:
    - WanderMovement
    - ChaseMovement

State Machine

- Menu → Play → Pause → Dialogue → GameOver

3.3. Mô hình tổ chức project (Packages)

- **ai.movement** : xử lý AI di chuyển (Wander, Chase)
- **combat** : hệ thống chiến đấu, sát thương
- **entity** : lớp cơ sở của mọi đối tượng trong game
- **entity_manager**: quản lý các entity theo map
- **game_data**: lưu/tải game, SaveManager
- **input_manager**: nhận input từ người chơi
- **interact_manager**: xử lý tương tác với object, NPC
- **monster_data**: dữ liệu quái
- **npc_data**: dữ liệu NPC, hội thoại
- **object_data**: item, vật phẩm, vũ khí
- **player_manager**: người chơi, chỉ số, UI player
- **sound_manager**: âm thanh
- **tile**: bản đồ, tile, chunk
- **ui**: giao diện, HUD, menu



Hình 1: Mô hình tổ chức project

CHƯƠNG 4. KIẾN TRÚC HỆ THỐNG VÀ GIẢI PHÁP CÀI ĐẶT

4.1. Game loop và State Machine

Trong trò chơi, toàn bộ hoạt động được điều khiển bởi một "vòng lặp game" chạy trên thread riêng, gọi tuần tự hàm update() và repaint() của GamePanel. Luồng cập nhật này lại phụ thuộc vào trạng thái game hiện tại (START, PLAY, PAUSE, DIALOGUE, GAME_OVER…) do GameStateManager quản lý. Nhờ đó, logic game, vẽ màn hình và chuyển cảnh được tổ chức rõ ràng, tránh if–else rối.

4.1.1. GameLoop (thread game)

Lớp: main.GameLoop implements Runnable

Vai trò:

- Đóng vai trò là vòng lặp game chính (game loop) chạy trong một thread riêng.
- Đảm bảo game cập nhật và vẽ với tốc độ cố định 60 FPS nhờ biến FPS = 60.

Cách hoạt động:

- Khi GamePanel.startGameThread() được gọi, chương trình tạo một thread mới:

```
gameThread = new Thread(new GameLoop(this));
gameThread.start();
```

- Trong phương thức run():
    - Tính toán drawInterval = 1000000000 / FPS (đơn vị nano giây) để biết mỗi khung hình cách nhau ~16.67ms.
    - Biến nextDrawTime lưu thời điểm cần vẽ khung hình tiếp theo.
    - Vòng lặp while(true) thực hiện liên tục:
        1. gp.update(); – cập nhật logic game (entity, map, UI…).
        2. gp.repaint(); – yêu cầu Swing vẽ lại màn hình.
        3. Tính remainingTime = nextDrawTime - System.nanoTime() để biết cần ngủ bao lâu.
        4. Thread.sleep(remainingTime / 1_000_000); để **giữ FPS ổn định**, tránh game chạy quá nhanh.
        5. Cập nhật nextDrawTime += drawInterval;.

4.1.2. GamePanel – vòng update/render

Lớp: main.GamePanel extends JPanel
Vai trò:

- Là lõi chính của game – vừa là component Swing để vẽ, vừa chứa toàn bộ hệ thống con:
    - Quản lý tile, chunk: TileManager, ChunkManager
    - Quản lý entity & object: EntityManager, ObjectManager
    - Hệ thống va chạm: CollisionChecker, Interact
    - Hệ thống UI: UIManager và các UI cụ thể (MainMenuUI, PauseOverlay, HealthUI, MessageUI, GameOverUI, DialogueUI…)
    - Lưu game: SaveManager
    - Trạng thái game: GameStateManager gsm

- Cài đặt vòng cập nhật (update) và vòng vẽ (render) tương ứng với update() và paintComponent().

a) Vòng cập nhật – update()

Phương thức update() được GameLoop gọi mỗi frame. Logic bên trong phụ thuộc vào gsm.getState():

```
public void update() {
    switch (gsm.getState()) {
        case START, GAME_OVER -> uiManager.update(gsm.getState());

        case PLAY -> {
            chunkM.updateChunks(em.getPlayer().worldX, em.getPlayer().worldY);
            currentMap = uTool.mapNameToIndex(chunkM.pathMap);
            em.update(currentMap);
            if (em.getPlayer() != null && em.getPlayer().isDead()) {
                gsm.setState(GameState.GAME_OVER);
                return;
            }
            uiManager.update(GameState.PLAY);
        }
        case DIALOGUE -> uiManager.update(gsm.getState());
        case PAUSE    -> uiManager.update(GameState.PAUSE);
    }
}
```

- **START, GAME_OVER**: chỉ cập nhật UI tương ứng (menu chính, màn hình game over…). Không chạy logic map và entity.
- **PLAY**:
    - Cập nhật chunk theo vị trí player: chunkM.updateChunks(...) → chỉ load/vẽ những phần map cần thiết.
    - Cập nhật map hiện tại: currentMap = uTool.mapNameToIndex(chunkM.pathMap);
    - Gọi em.update(currentMap); để cập nhật player, quái, NPC… của map đó.
    - Kiểm tra nếu player chết → chuyển state sang GAME_OVER.
    - Cập nhật các UI trong trạng thái PLAY (thanh máu, status, message…).
- **DIALOGUE**: chỉ cập nhật UI hội thoại (DialogueUI), game "đứng hình" nhưng UI vẫn chạy hiệu ứng chữ.

- **PAUSE**: game logic dừng, chỉ update UI pause.

Nhờ cách chia update theo GameState, hệ thống tránh được việc entity vẫn di chuyển khi game đang pause hoặc ở menu.

b) Vòng vẽ – paintComponent(Graphics g)

GamePanel override paintComponent() để vẽ:

```
@Override
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D) g;

    // Vẽ nền/map nếu không ở màn hình START
    if (gsm.getState() != GameState.START)
        tileM.draw(g2, chunkM);

    // Vẽ entity, object
    em.draw(g2, currentMap);
    om.draw(g2, currentMap, em.getPlayer());

    // Vẽ UI tương ứng với state hiện tại
    uiManager.draw(g2, gsm.getState());

    frameCounter++;
    if (frameCounter >= 1_000_000) frameCounter = 0;
    g2.dispose();
}
```

Thứ tự vẽ:

1. Tile/map (background)
2. Entity & object (player, quái, vật phẩm…)
3. UI (máu, menu, pause, game over, dialogue…)

Thiết kế này đảm bảo UI luôn nằm trên cùng, map không che entity, và entity không che HUD.

4.1.3. GameState & GameStateManager

Enum: main.GameState (không show trong code trên nhưng được sử dụng)
Lớp quản lý: main.GameStateManager

```
public class GameStateManager {
    public GameState currentState = GameState.START;
```

```
    public GameState getState() {
        return currentState;
    }

    public void setState(GameState state) {
        currentState = state;
    }
}
```

Vai trò:

- Đóng gói trạng thái hiện tại của game vào một chỗ duy nhất (currentState).
- Các trạng thái điển hình:
    - START – màn hình menu chính
    - PLAY – đang chơi bình thường
    - PAUSE – tạm dừng
    - DIALOGUE – đang hiển thị hội thoại
    - GAME_OVER – màn hình thua cuộc
- GamePanel, UI và các hệ thống khác chỉ cần gọi:
    - gsm.getState() để biết đang ở mode nào.
    - gsm.setState(GameState.PLAY); để chuyển màn hình (ví dụ sau khi nhấn "Start Game" ở menu, hoặc sau khi restart).

Lợi ích của State Machine:

- Tách rõ logic theo trạng thái, tránh nhiều if–else lồng nhau khắp nơi.
- Dễ mở rộng: muốn thêm trạng thái mới (ví dụ SETTINGS, INVENTORY) chỉ cần bổ sung vào GameState và xử lý thêm trong update()/draw().
- Dễ kiểm soát: đảm bảo ở mỗi thời điểm game chỉ có một state chính, giúp vòng game loop gọn gàng, dễ debug.

4.2. Entity System

Hệ thống Entity là lõi để biểu diễn mọi thực thể trong thế giới game: người chơi, quái, NPC, vật phẩm, cổng dịch chuyển… Toàn bộ đều dựa trên lớp cơ sở Entity (hoặc WorldObject) và được tách nhỏ thành các module chuyên trách: di chuyển

(EntityMovement), vẽ sprite (EntityDraw, EntitySpriteManager), AI (MovementController) và chiến đấu (CombatComponent, CombatSystem).

Hệ thống này giúp:

- Tái sử dụng code tối đa giữa Player, Monster, NPC.
- Dễ mở rộng thêm loại quái, NPC, vũ khí mới mà không phải sửa GamePanel.
- Gắn AI di chuyển theo dạng "chiến lược" (Strategy) thay đổi được runtime.

4.2.1. Lớp Entity – lớp cơ sở cho mọi nhân vật

Lớp: entity.Entity implements CombatContext

Vai trò:

- Là lớp cha cho Player, Monster, NPC.
- Chứa toàn bộ dữ liệu chung: vị trí, kích thước, animation, va chạm, máu/atk/def, knockback, i-frame, controller AI, combat component…

Các nhóm thuộc tính chính:

1. Vị trí & kích thước
    - worldX, worldY: tọa độ trong world (pixel).
    - width, height: kích thước sprite/hitbox.
    - screenX, screenY: vị trí tương đối so với camera (tâm màn hình).
2. Animation & hướng
    - Các frame đi bộ: up1, up2, down1, down2, left1, left2, right1, right2.
    - Các frame tấn công: atkUp1, atkUp2, ...
    - direction: hướng di chuyển hiện tại.
    - attackDir: hướng "lock" khi bắt đầu đòn đánh, dùng cho animation combat.
    - spriteCounter, spriteNum, animationON: điều khiển đổi frame.

3. Va chạm & knockback
    - solidArea: hitbox va chạm.
    - Cờ collisionOn, collisionXOn, collisionYOn…
    - Vận tốc & knockback: velX, velY, knockbackFrames, knockbackCounter, clearVelocity()…
4. Chỉ số & trạng thái chiến đấu
    - hp, maxHp, atk, def + các getter/setter (setStats, getHP, reduceHP…).
    - I-frame: invulnerable, invulnFrames, invulnCounter.
    - Field combat: CombatComponent combat, attackBox dùng chung với hệ ECS combat.
5. Liên kết hệ thống
    - Tham chiếu gp: GamePanel để truy cập tile, collision, UI…
    - Ba "module con":
        - EntityMovement emo: xử lý di chuyển/knockback.
        - EntitySpriteManager esm: xử lý đổi sprite, load sprite.
        - EntityDraw ed: xử lý vẽ entity lên màn hình.
6. AI & Dialogue
    - MovementController controller: chiến lược di chuyển (wander, chase…).
    - dialogues, dialogueSet, facePlayer(), speak() – hỗ trợ hệ thống hội thoại cho NPC.

Hàm update() của Entity:

```
public void update() {
    if (isKnockbackActive()) {
```

```
            emo.applyKnockback(this);
        } else {
            if (controller != null) {
                controller.decide(this);
                emo.moveByDirection(this);
            } else {
                int[] d = computeDelta();
                emo.moveWithDelta(this, d[0], d[1]);
            }
        }

        CombatSystem.tick(this);
        StatusSystem.update(this);   // i-frame, timer
        esm.updateSprite(this);
    }
```

- Nếu đang bị knockback → chỉ áp lực knockback, bỏ qua input/AI.
- Nếu có MovementController → giao cho AI quyết định hướng (decide) rồi di chuyển.
- Nếu không có AI → dùng direction + computeDelta.
- Cuối cùng tick hệ thống combat & status, sau đó cập nhật sprite.

4.2.2. Hệ thống vẽ & di chuyển: EntityDraw, EntityMovement, EntitySpriteManager

a) EntityMovement – xử lý di chuyển & knockback

Lớp: entity.EntityMovement

- Hàm moveByDirection(Entity e): di chuyển theo e.direction với actualSpeed.
- Hàm moveWithDelta(Entity e, int dx, int dy):
    - Tính nextX, nextY.
    - Gọi gp.cChecker.checkTile, checkWorldObject, checkPlayer để xác định va chạm.
    - Nếu không va chạm → cập nhật worldX, worldY.
- Hàm applyKnockback(Entity e):

- Di chuyển từng bước 1 pixel theo velX, velY, mỗi bước đều check va chạm.
- Giảm knockbackCounter, khi hết hoặc bị kẹt thì reset velocity.

Nhờ tách EntityMovement, toàn bộ logic dịch chuyển (kể cả boss, slime, NPC) dùng chung, chỉ khác cách set direction/dx, dy.

b) EntityDraw – vẽ Entity theo camera & combat

Lớp: entity.EntityDraw

Trong draw(Graphics2D g2, Entity e):

- Tính camera origin dựa trên vị trí player ⇒ chuyển world → screen.
- Culling: nếu entity nằm ngoài vùng camera thì bỏ qua, giảm load vẽ.
- Nếu Entity đang i-frame: cho sprite nháy (ẩn/hiện) theo frameCounter.
- Lấy trạng thái combat từ CombatSystem:
  - Nếu đang tấn công → ưu tiên dùng attackDir cho animation.
  - Chọn sprite walk/attack tương ứng dirForAnim + spriteNum.
- Với quái có attack-anim dài hơn 1 tile:
  - Offset sprite lên/trái một đoạn (up/left) để khớp với hitbox.
  - Có thể vẽ với kích thước gốc của frame tấn công (image.getWidth/Height).
- Vẽ debug:
  - Viền đỏ hitbox solidArea.
  - Nếu đòn đánh đang active: vẽ attackBox bán trong suốt để dễ debug.

c) EntitySpriteManager – đổi frame & load ảnh

Lớp: entity.EntitySpriteManager

- updateSprite(Entity e): tăng spriteCounter, sau mỗi 8 frame thì flip spriteNum (1 ↔ 2).

- setup(String imagePath, int width, int height):
    - Đọc file .png từ resources.
    - Dùng UtilityTool.scaleImage để scale đúng width, height.

Mọi entity (player, quái, NPC…) đều dùng chung chuẩn setup() để load sprite.

4.2.3. Player – nhân vật điều khiển được

Lớp: player_manager.Player extends Entity

Bổ sung thêm:

- Điều khiển:
    - InputController input, PlayerMovement pm, PlayerAnimation pa.
    - Interact iR để tương tác object/NPC.
- Level & EXP:
    - level, exp, expToNext.
    - Các base stats: baseHp, baseAtk, baseDef, tăng mỗi level (hpPerLevel, atkPerLevel, defPerLevel).
    - Hàm gainExp(), levelUp(), calcExpToNext() – lên level sẽ:
        - Tăng HP/ATK/DEF,
        - Hồi full máu,
        - Hiện thông báo MessageUI.
- Combat & vũ khí:
    - Weapon currentWeapon + equipWeapon(Weapon weapon):
        - Config thời gian windup/active/recover/cooldown cho combat.
        - Set kích thước attackBox.

        - Load sprite tấn công tương ứng vũ khí (spriteKey()).
    - handleAttackInput():
        - Chống spam bằng attackBtnLock.
        - Gọi CombatSystem.startAttack() khi phím tấn công được nhấn.
- Tương tác NPC & hội thoại:
    - handleNPCInteraction():
        - Dùng cChecker.checkEntity() để tìm NPC đứng cạnh.
        - Nếu đứng gần mà chưa bấm talk: hiển thị gợi ý "Press 'E' to talk...".
        - Khi bấm E: gọi npc.speak(gp), set GameState.DIALOGUE.
- setDefaultValues():
    - Spawn player tại map 3 (nhà, gần dungeon).
    - Set defaultSpeed, actualSpeed, hướng ban đầu.

Player override update() để:

- Nếu bị knockback: chỉ xử lý knockback + combat tick.
- Ngược lại: đọc input → tính movement → di chuyển.
- Update animation, xử lý buff speed, attack input, NPC interaction.

4.2.4. Monster – kế thừa Entity, thêm AI & EXP

Lớp: monster_data.Monster extends Entity

Thêm các cấu hình:

- Combat:
    - attackDamage, attackKnockback, attackTriggerRadius, faceLockThreshold.
    - Kích thước đòn đánh: atkW, atkH.

- EXP:
    - expReward: số EXP thưởng khi quái chết.
    - initExpFromStats(): auto tính EXP từ HP/ATK/DEF.

AI tấn công:

- Override update():
    - Gọi decideAttack() để quyết định có bắt đầu đòn đánh không.
    - Giữ nguyên vị trí nếu đang tung đòn (không cho di chuyển trong lúc attack).
    - Sau đó gọi super.update() để di chuyển + combat tick.
- decideAttack():
    - Lấy player, tính Rectangle thân của quái & player.
    - Nếu chạm thân → đánh luôn.
    - Nếu không, dựng "reach-rectangle" kéo dài theo hướng đang nhìn, nếu player nằm trong đó → bắt đầu đòn.
    - Khi bắt đầu đòn: faceOnceToward(p) + attackDir = direction + CombatSystem.startAttack().

Xử lý chết & drop:

- onDeath():
    - Tìm Player, cộng EXP (p.gainExp(expReward)).
    - 25% cơ hội drop bình máu: gp.om.spawnHealthPosion(mapIndex, worldX, worldY).

Một Số class quái cụ thể:

- SlimeMonster:

- Quái yếu, HP/ATK thấp, không có riêng sprite attack (hasAttackAnim = false).
- Di chuyển WanderMovement đơn giản, attackBox nhỏ.

- OrcMonster:
  - Quái mạnh hơn, có sprite tấn công nhiều hướng.
  - Dùng AI AggroSwitchMovement: wander → chase khi player vào bán kính aggro.
  - Combat timing nặng hơn, knockback mạnh hơn.
- SkeletonLord:
  - Boss 2 phase:
    - Phase 1: tốc độ và sát thương bình thường.
    - Khi HP ≤ 50%: chuyển phase 2 (enraged = true):
      - Tăng speed (BASE_SPEED → ENRAGE_SPEED).
      - Mở rộng attackBox.
      - Đổi sang bộ sprite phase2 (p2Up1…).
      - Thay timing đòn (ra chiêu nhanh hơn).
  - Override update() để mỗi frame kiểm tra HP% và cập nhật phase, speed, combat config.

4.2.5. NPC & hệ thống hội thoại

Lớp: npc_data.NPC_Oldman extends Entity

- Là NPC dùng để mở đầu cốt truyện, cho vũ khí đầu game, hướng dẫn người chơi.
- Cấu hình:
  - Dùng sprite riêng /npc/oldman_*.

- setController(new WanderMovement(1, 120)) để ông cụ đi qua lại cho sống động.
- setDialogue() gán sẵn nhiều set hội thoại trong mảng dialogues[set][line].

- Khi Player bấm talk:
  - Player.handleNPCInteraction() gọi npc.speak(gp).
  - Trong speak():
    - NPC xoay mặt về phía player (facePlayer()).
    - Lấy DialogueUI từ uiManager và gọi startDialogue(currentSet) để hiển thị từng dòng chữ trên màn hình.

4.2.6. WorldObject, Weapon, Portal – một số vật phẩm & tương tác map tiêu biểu

WorldObject – base class cho object map

Lớp: object_data.WorldObject

- Dùng cho mọi object trong map: bình máu, rương, vũ khí rơi dưới đất, portal…
- Thuộc tính:
  - worldX, worldY, width, height.
  - staticImage để vẽ.
  - solidArea, collision, mapIndex.
- Hàm setup(path, w, h) để load ảnh & scale.
- draw() mặc định vẽ ảnh ở screenX, screenY (đối tượng sẽ được vẽ qua ObjectManager với camera)

Weapon – vũ khí trong thế giới

Lớp: object_data.weapons.Weapon extends WorldObject (abstract)

- Vừa là vật phẩm trong world, vừa chứa config combat khi equip:

- Kích thước attackBox: atkBoxW/atkBoxH.
- Thời gian đòn đánh: windup, active, recover, cooldown.
- Hệ damage(hiện tại chưa thể ứng dụng trong code):
    - atkMultiplier() + atkFlat() dùng trong computeDamage(Player p, Entity target).

- Khi player nhặt và trang bị:
    - Player.equipWeapon() lấy thông tin từ Weapon để cấu hình CombatComponent + load sprite attack phù hợp (spriteKey()).

ObjectPortal – cổng dịch chuyển map

Lớp: object_data.ObjectPortal extends WorldObject

- Có 2 frame f1, f2 để tạo hiệu ứng chuyển động.
- Thuộc tính:
    - targetMap, targetWorldX, targetWorldY: chỉ định nơi dịch chuyển.
- update() tăng animCounter, getRenderImage() trả về frame 1/2 theo frameDuration, tạo cảm giác portal nhấp nháy.

4.3. Entity Manager Layer

Lớp logic "Entity" (Player, Monster, NPC, WorldObject) đã được tách riêng, nhưng để game dễ quản lý theo từng map và xử lý spawn/respawn, vẽ theo thứ tự, nhóm sử dụng một tầng quản lý (Entity Manager Layer).

Tầng này gồm 4 manager chính:

- EntityManager – điều phối Player, Monster, NPC;
- MonsterManager – quản lý toàn bộ quái theo map, spawn & respawn;
- NPCManager – quản lý NPC theo map;

- ObjectManager – quản lý vật phẩm, cửa, portal, shop… theo map.

Các manager này giúp GamePanel chỉ cần gọi em.update(), em.draw() và om.draw() mà không phải tự tay quản từng list.ư

4.3.1. EntityManager – điều phối Player, Monster, NPC

Lớp: entity_manager.EntityManager

Vai trò:

- Đóng vai trò "trung tâm" quản lý Player, MonsterManager, NPCManager.
- Cung cấp API đơn giản cho GamePanel:
    - getPlayer()
    - getMonsters(int map)
    - getNPCs(int map)
    - update(int currentMap)
    - draw(Graphics2D g2, int currentMap)

Cấu trúc:

- Trong constructor:

```
this.player = new Player(gp, input);
this.mM = new MonsterManager(gp);
this.npcM = new NPCManager(gp);
```

Khi tạo EntityManager, Player + toàn bộ quái & NPC cơ bản cũng được khởi tạo theo.

Quy trình update:

```
public void update(int currentMap) {
    player.update();
    mM.update(currentMap, player.worldX, player.worldY);

    List<Entity> monsters = mM.getMonsters(currentMap);
```

```
    for (Entity n : npcM.getNPCs(currentMap)) {
        n.update();
    }

    CombatSystem.resolvePlayerHits(player, monsters);

    for (Entity e : monsters) {
        if (e instanceof Monster m) {
            CombatSystem.resolveMonsterHitAgainstPlayer(m, player);
        }
    }
}
```

- Cập nhật Player trước (đọc input, di chuyển, tấn công).
- Cập nhật quái thông qua MonsterManager.update() – bao gồm luôn xử lý quái chết/respawn.
- Cập nhật toàn bộ NPC trên map hiện tại.
- Sau đó xử lý va chạm combat:
    - resolvePlayerHits(...) – player đánh trúng quái.
    - Với từng quái: resolveMonsterHitAgainstPlayer(...) – quái đánh trúng player.

Quy trình vẽ:

```
public void draw(Graphics2D g2, int currentMap) {
    List<Entity> all = new ArrayList<>();
    all.addAll(mM.getMonsters(currentMap));
    all.addAll(npcM.getNPCs(currentMap));
    all.add(player);

    all.sort((a, b) -> Integer.compare(a.worldY, b.worldY));
    for (Entity e : all) e.draw(g2);
}
```

- Gộp tất cả quái + NPC + Player vào một list.

- Sort theo worldY để vẽ từ trên xuống dưới → đảm bảo hiệu ứng "đè lớp" tự nhiên (entity phía dưới đứng "lên trên").
- Gọi draw() của từng Entity, bên trong sẽ dùng EntityDraw để chuyển world → screen theo camera.

4.3.2. MonsterManager – spawn & respawn quái

Lớp: entity_manager.MonsterManager

Vai trò:

- Quản lý **danh sách quái của từng map**.
- Định nghĩa **các điểm spawn cố định** (SpawnSlot) và **cơ chế respawn** tự động.
- Sinh nhiều loại quái khác nhau (SLIME, BAT, ORC, BOSS) theo ID.

Lưu trữ quái:

- Map<Integer, List<Entity>> monstersByMap:
  - key: mapId
  - value: list quái (Entity) đang tồn tại trong map.
- Danh sách slot spawn:

```
private static class SpawnSlot {
    final int mapId;
    final int worldX, worldY;
    final String monsterId;       // "SLIME", "BAT", "ORC", "BOSS"...
    final long respawnDelayMs;
    Entity current;
    long lastDeathTime = 0L;
}
```

Mỗi slot tương ứng với "một chỗ cắm quái": biết map, vị trí, loại quái, thời gian hồi sinh.

Khởi tạo spawn ban đầu:

- setupSpawnSlots():
    - Tạm thời đổi chunkM.pathMap sang "map0", "map1" để load đầy đủ tile và check collision.
    - Với map 0: duyệt một vùng toạ độ dạng lưới, mỗi vài ô tile spawn một con SLIME (nếu chỗ đó không phải tường).
    - Với map 1: xen kẽ ORC và BAT bằng cách dùng mobCount % 2.
    - Thêm một slot đặc biệt cho BOSS (SkeletonLord) với respawn ~10 phút.
    - Cuối cùng load lại map cũ của ChunkManager để không ảnh hưởng gameplay.
- initialSpawn():
    - Duyệt tất cả SpawnSlot và gọi spawnNow(slot) để sinh quái ban đầu vào monstersByMap.

Cập nhật & respawn:

```
public void update(int mapId, int playerX, int playerY) {
    // 1. Update quái đang sống
    List<Entity> list = monstersByMap.get(mapId);
    if (list != null) {
        Iterator<Entity> it = list.iterator();
        while (it.hasNext()) {
            Entity e = it.next();
            e.update();
            if (isDead(e)) {
                registerDeath(e);
                it.remove();
            }
        }
    }

    // 2. Respawn cho map hiện tại
    handleRespawn(mapId, playerX, playerY);
}
```

- Đầu tiên update AI + di chuyển + combat cho từng quái.
- Nếu quái chết (isDead()): ghi lại lastDeathTime trong SpawnSlot tương ứng, remove khỏi list.
- handleRespawn(...):
    - Kiểm tra waited >= respawnDelayMs → đủ thời gian mới respawn.
    - Nếu bật useDistanceCheck thì còn check cả quãng cách với player (isFarFromPlayer()).
    - Khi đủ điều kiện, gọi spawnNow(slot) để sinh một quái mới đúng loại & đúng vị trí.

Tạo quái theo ID:

```
private Entity createMonster(String id, int mapId) {
    return switch (id) {
        case "SLIME" -> { ... }    // random RedSlime / Slime
        case "BAT"   -> new BatMonster(gp, mapId);
        case "ORC"   -> new OrcMonster(gp, mapId);
        case "BOSS"  -> new SkeletonLord(gp, mapId);
        default      -> null;
    };
}
```

Chưa quá tối ưu nhưng hiện tại quái vẫn đang ít nên tạm thời dùng phương pháp switch - case

4.3.3. NPCManager – quản lý NPC theo map

Lớp: entity_manager.NPCManager

Vai trò:

- Lưu và cập nhật tất cả NPC trên từng map.
- Tách riêng khỏi MonsterManager để tránh lẫn lộn giữa NPC và quái.

Update & draw:

```
public void update(int mapId) {
    for (Entity npc : getNPCs(mapId)) {
        npc.update();
    }
}

public void draw(Graphics2D g2, int mapId) {
    for (Entity npc : getNPCs(mapId)) {
        npc.draw(g2);
    }
}
```

- Trong thực tế, EntityManager thường sẽ gọi npcM.getNPCs() để gộp vẽ chung với quái + player (sortable theo worldY), còn NPCManager vẫn cung cấp draw riêng khi cần.

4.3.4. ObjectManager – quản lý vật phẩm, cửa, portal, shop

Lớp: entity_manager.ObjectManager

Vai trò:

- Quản lý WorldObject trên từng map:
    - Cửa (ObjectDoor), chìa khóa (ObjectKey),
    - Vũ khí (Axe…), shop (Shop),
    - Portal (ObjectPortal),
    - Bình máu drop từ quái (HealthPosion), …
- Chịu trách nhiệm spawn cố định lúc vào game và spawn động (drop từ quái).

Constructor:

```
public ObjectManager(GamePanel gp) {
    this.gp = gp;
```

```
    spawnObjects();
}
```

Gọi spawnMap0(), spawnMap1(), spawnMap2(), spawnMap3() để đặt object ban đầu.\

Ví dụ:

```
private void spawnMap0() {
    int t = gp.tileSize;
    addObject(new Shop(gp, 0),      46 * t, 15 * t);
    addObject(new ObjectDoor(gp, 0),48 * t - 23, 18 * t);
    addObject(new ObjectPortal(gp,0),47 * t + 12, 47 * t + 12);
}
```

API chính:

- addObject(WorldObject obj, int wx, int wy):
    - Gán worldX/worldY cho object.
    - Thêm vào objectsByMap theo obj.mapIndex.
- getObjects(int mapId): trả về list object của map.

Update & vẽ:

```
public void update(int mapId) {
    for (WorldObject o : getObjects(mapId))
        o.update();
}

public void draw(Graphics2D g2, int mapId, Entity player) {
    if (player == null) return;

    // Tính vùng camera (world)
    final int leftWorld  = player.worldX - player.screenX - gp.tileSize;
    final int rightWorld = player.worldX + (gp.screenWidth - player.screenX) +
gp.tileSize;
    final int topWorld   = player.worldY - player.screenY - gp.tileSize;
    final int botWorld   = player.worldY + (gp.screenHeight - player.screenY) +
gp.tileSize;
```

```
    for (WorldObject o : getObjects(mapId)) {
        // Culling: nằm ngoài camera thì bỏ qua
        ...
        int sx = o.worldX - player.worldX + player.screenX;
        int sy = o.worldY - player.worldY + player.screenY;

        BufferedImage img = o.getRenderImage() hoặc o.staticImage;
        if (img != null) g2.drawImage(img, sx, sy, null);

        // Vẽ hitbox đỏ để debug
        g2.drawRect(...);
    }
}
```

Nhờ ObjectManager:

- GamePanel không cần biết chi tiết portal, shop, cửa, vũ khí… ở map nào.
- Khi muốn reload trạng thái object (ví dụ khi restart game), chỉ cần gọi:

om.reloadMapObjects(currentMap);

Khi quái chết và drop bình máu:

```
public void spawnHealthPosion(int mapIndex, int wx, int wy) {
    HealthPosion potion = new HealthPosion(gp, mapIndex);
    addObject(potion, wx, wy);
}
```

4.4. Combat System (package combat)

Hệ thống combat được thiết kế theo kiểu component + system: mỗi Entity có một CombatComponent lưu trạng thái đòn đánh, còn các class trong package combat là các "system" xử lý logic: chuyển pha đòn đánh, tính hitbox, kiểm tra va chạm, gây sát thương, i-frame và knockback.

Mục tiêu thiết kế:

- Tách logic combat ra khỏi Player / Monster để dễ tái sử dụng.
- Hỗ trợ 4 pha đòn đánh chuẩn: *Windup* → *Active* → *Recover* → *Cooldown*.

- Hỗ trợ attackBox riêng (không trùng hitbox thân), có thể align theo hướng và kích thước khác nhau.
- Đảm bảo mỗi swing chỉ đánh trúng 1 mục tiêu một lần (hit-once-per-swing).
- Có i-frame và knockback thống nhất cho mọi Entity.

4.4.1. CombatComponent & CombatContext

Lớp: combat.CombatComponent

Vai trò:

- Là component lưu trữ toàn bộ trạng thái combat của một Entity:
    - Thời lượng các pha:
      windupFrames, activeFrames, recoverFrames, cooldownFrames.
    - Kích thước hitbox tấn công:
      attackWidth, attackHeight, Rectangle attackBox.
    - Trạng thái pha hiện tại:
      isAttacking, attackPhase (0–3), phaseTimerFrames,
      cooldownCounterFrames.
    - Thông số knockback: knockbackForce.
    - Tập hitThisSwing để ghi lại các mục tiêu đã bị đánh trúng trong 1 swing.

API chính:

- Thiết lập:
    - setTimingFrames(windup, active, recover, cooldown)
    - setAttackBoxSize(width, height)
    - setKnockbackForce(int kb)
- Trạng thái:

- isAttacking(), isAttackActive() (chỉ true khi pha 2 – Active).
- getAttackPhase(), getAttackBox()…

- Hit-once-per-swing:
  - wasHitThisSwing(target), markHit(target), clearHitThisSwing().

Interface: combat.CombatContext

- Định nghĩa tối thiểu những gì một "chủ đòn" cần cung cấp cho hệ thống combat:
  - getWorldX(), getWorldY() – vị trí trong world.
  - getSolidArea() – hitbox thân (body).
  - getDirection() – hướng tấn công.
  - isDead() – kiểm tra đã chết chưa.
- Entity implements CombatContext, nên CombatSystem có thể làm việc với Player, Monster… một cách uniform.

4.4.2. AttackPhaseSystem – state machine đòn đánh

Lớp: combat.AttackPhaseSystem (static, final)

Vai trò:

- Cài đặt state machine 4 pha cho một đòn đánh:
  - 0: idle (không đánh)
  - 1: windup (lấy đà)
  - 2: active (bật hitbox, có thể gây sát thương)
  - 3: recover (hạ tay)
- Điều khiển chuyển pha theo từng frame, đồng thời align attackBox với thân Entity.
- API:

- canStart(CombatComponent cc)
  chỉ cho bắt đầu đòn nếu chưa đánh (!isAttacking) và cooldownCounterFrames == 0.

- start(CombatComponent cc, CombatContext owner):
  - Kiểm tra owner còn sống, attackWidth/Height > 0.
  - Đặt isAttacking = true, attackPhase = 1 (windup).
  - Gán phaseTimerFrames = windupFrames.
  - Gọi alignAttackBox() lần đầu.
  - clearHitThisSwing() để reset mục tiêu đã trúng.

- update(CombatComponent cc, CombatContext owner):
  - Giảm cooldownCounterFrames mỗi frame nếu > 0.
  - Nếu đang attacking:
    - Trong pha Active (2), liên tục alignAttackBox() (cho trường hợp Entity vẫn di chuyển).
    - Giảm phaseTimerFrames.
    - Khi timer về 0:
      - 1 → 2: chuyển Windup → Active, set timer = activeFrames.
      - 2 → 3: chuyển Active → Recover, tắt attackBox (set size 0), clear hit list.
      - 3 → 0: kết thúc đòn, set isAttacking = false, set cooldownCounterFrames = cooldownFrames, tắt attackBox.

Căn chỉnh attackBox theo hướng:

- alignAttackBox(CombatComponent cc, CombatContext owner):

- Lấy body = owner.getSolidArea() rồi tính (bx, by, bw, bh) trong world.
- Xác định dir:
    - Mặc định dùng owner.getDirection().
    - Đối với Monster (boss) đang đánh: dùng hướng đã lock e.attackDir (đảm bảo đòn không "xoay theo" trong lúc Active).
- Tính vị trí ax, ay của attackBox theo dir:
    - "up": canh giữa theo chiều ngang, đặt box phía trên thân.
    - "down": đặt box dưới thân.
    - "left": đặt box bên trái.
    - "right" (default): đặt box bên phải.
- Cập nhật Rectangle attackBox bằng setBounds(ax, ay, width, height).

Nhờ đó, mọi đòn đánh (Player, Orc, SkeletonLord…) đều dùng chung logic phase và hitbox.

4.4.3. CombatSystem – façade cho toàn bộ combat

Lớp: combat.CombatSystem

Vai trò:

- Là "cửa vào" duy nhất của gameplay đến hệ thống combat.
- Đóng gói các lời gọi đến AttackPhaseSystem, StatusSystem, HitResolve*, KnockbackService, DamageProcessor.
- API tiêu biểu:
- Đọc trạng thái:
    - isAttacking(CombatComponent cc)
    - isAttackActive(CombatComponent cc)

- getPhase(CombatComponent cc)

- Bắt đầu & cập nhật đòn:
  - canStartAttack(CombatComponent cc)
  - startAttack(CombatComponent cc, CombatContext owner)
  - update(CombatComponent cc, CombatContext owner) → gọi AttackPhaseSystem.update.

- Tick combat cho 1 entity:

```
public static void tick(Entity e) {
    if (e == null || e.combat == null) return;
    update(e.combat, e);        // phase, hitbox, cooldown
    updateStatus(e);            // i-frame
}
```

Va chạm, knockback:

- computePlayerAttackKnockback(Player p) → nhờ KnockbackService.
- computeMonsterAttackKnockback(Monster m, Player p).
- resolvePlayerHits(player, monsters) → gọi HitResolvePlayer.
- resolveMonsterHitAgainstPlayer(m, player) → gọi HitResolveMonster.

Nhờ lớp này, EntityManager chỉ cần:

```
CombatSystem.resolvePlayerHits(player, monsters);
CombatSystem.resolveMonsterHitAgainstPlayer(m, player);
```

4.4.4. Xử lý va chạm & gây sát thương

CollisionUtil – tính body rect trong world

- combat.CollisionUtil.getEntityBodyWorldRect(Entity e):
  - Dùng e.worldX/Y + solidArea để trả về Rectangle thân trong toạ độ world.

- Đảm bảo nếu solidArea null vẫn có size fallback.

HitResolvePlayer – Player đánh quái

- HitResolvePlayer.resolve(Player player, List<Entity> monsters):
    - Chỉ chạy nếu Player còn sống và player.combat đang ở pha Active.
    - Lấy attack = player.combat.getAttackBox(), nếu width/height <= 0 thì bỏ qua.
    - Tính rawDamage = player.getATK(), knockback vector từ computePlayerAttackKnockback.
    - Duyệt từng quái trong danh sách:
        - Cast sang Monster, skip nếu đã chết.
        - Lấy monsterBody = CollisionUtil.getEntityBodyWorldRect(m).
        - Nếu attack giao với monsterBody và quái chưa bị đánh trong swing này:
            - Gọi DamageProcessor.applyDamage(m, rawDamage, knockbackX, knockbackY).
            - Đánh dấu quái đã trúng bằng CombatSystem.markHitLanded(player.combat, m).

HitResolveMonster – Quái đánh Player

- HitResolveMonster.resolve(Monster m, Player player):
    - Tương tự Player:
        - Chỉ xử lý nếu quái còn sống, player còn sống, combat của quái đang Active.
        - Lấy attack = m.combat.getAttackBox().

- Tính playerBody.
- Nếu giao nhau và player chưa bị đánh trong swing đó:
  - Tính damage từ m.getATK().
  - Tính knockback computeMonsterAttackKnockback(m, player).
  - Gọi DamageProcessor.applyDamage(player, rawDamage, kb[0], kb[1]).
  - Đánh dấu đã trúng.

DamageProcessor – áp dụng sát thương, i-frame và knockback

- DamageProcessor.applyDamage(Entity target, int rawDamage, int knockbackX, int knockbackY):
  - Skip nếu target null, đã chết, hoặc đang invulnerable.
  - Tính damage đã trừ phòng thủ:

```
int damage = Math.max(1, rawDamage - target.getDEF());
```

  - Gọi target.reduceHP(damage).
  - Bật i-frame:
  - setInvulnerable(true), setInvulnCounter(invulnFrames).
  - Gây knockback:
  - target.applyKnockback(knockbackX, knockbackY, target.getKnockbackFrames());
  - Gọi hook target.onDamaged(damage) để sau này có thể thêm hiệu ứng (âm thanh, flash sprite…).

4.4.5. StatusSystem – i-frame và trạng thái tạm thời

Lớp: combat.StatusSystem

Vai trò:

- Cập nhật các trạng thái tạm thời của Entity mỗi frame, hiện tại tập trung vào invulnerability frame (i-frame) sau khi bị đánh.

Hàm chính:

```
public static void update(Entity e) {
    if (e == null) return;

    if (e.isInvulnerable()) {
        int next = e.getInvulnCounter() - 1;
        if (next <= 0) {
            e.setInvulnCounter(0);
            e.setInvulnerable(false);
        } else {
            e.setInvulnCounter(next);
        }
    }
}
```

- Nếu entity đang invulnerable:
    - Giảm dần invulnCounter.
    - Khi về 0 thì tắt trạng thái i-frame.
- CombatSystem.tick(e) luôn gọi StatusSystem.update(e) sau khi update phase, nên logic i-frame được đồng bộ với combat.

4.5. AI Movement System (package ai.movement)

Hệ thống AI Movement được thiết kế theo kiểu Strategy pattern:
mỗi Entity có một trường MovementController controller, và việc quyết định hướng đi hoàn toàn do các class cài đặt MovementController đảm nhiệm (WanderMovement, ChaseMovement, AggroSwitchMovement…).

Nhờ đó, logic di chuyển của quái / NPC:

- Không bị nhét cứng vào lớp Entity hay Monster.

- Có thể thay thế / kết hợp nhiều kiểu AI (đi lang thang, đuổi theo, chuyển trạng thái aggro…) chỉ bằng cách đổi controller.

4.5.1. MovementController – interface chiến lược di chuyển

Lớp: ai.movement.MovementController

```
public interface MovementController {
    void decide(Entity e);
}
```

- Mỗi frame, trong Entity.update(), nếu controller != null thì game sẽ gọi controller.decide(e):
    - Trong decide(), AI **chỉ** quyết định:
        - e.actualSpeed (tốc độ hiện tại),
        - e.direction ("up" | "down" | "left" | "right").
    - Còn việc chuyển đổi direction + actualSpeed thành worldX/worldY được thực hiện bởi EntityMovement.

Nhờ phân tách này, mọi loại di chuyển đều dùng chung hệ thống va chạm, knockback, camera, chỉ khác "bộ não" chọn hướng.

4.5.2. WanderMovement – AI đi lang thang

Lớp: ai.movement.WanderMovement

Vai trò:

- Cho quái / NPC di chuyển ngẫu nhiên quanh map hoặc trong một vùng giới hạn (bounded).
- Hạn chế "rung giật" bằng cách giữ hướng tối thiểu (minHoldFrames) và xử lý "rào mềm".

Các chế độ khởi tạo:

1. Lang thang tự do (không giới hạn vùng):

```
new WanderMovement(speed, changeEveryFrames);
new WanderMovement(speed, changeEveryFrames, minHoldFrames);
```

- speed: tốc độ entity.
- changeEveryFrames: bao lâu thì xét đổi hướng 1 lần (ví dụ 120 frame ≈ 2s @60fps).
- minHoldFrames: số frame phải giữ nguyên hướng tối thiểu để tránh việc đổi liên tục.

2. Lang thang trong vùng giới hạn:

```
new WanderMovement(speed, changeEveryFrames, minHoldFrames,
                   minX, minY, maxX, maxY, fencePadding);
```

- minX, minY, maxX, maxY: biên vùng hình chữ nhật trong world (px).
- fencePadding: "rào mềm", khi đến gần biên thì AI ưu tiên quay vào trong.

Logic trong decide(Entity e):

- Luôn set e.actualSpeed = speed.
- Nếu đang trong giai đoạn "giữ hướng" (hold < minHoldFrames) và không sát rào:
  - Giữ nguyên currentDir, set e.direction = currentDir, tăng hold.
- Nếu cần đổi hướng:
  - Khi:
    - Đã quá changeEveryFrames, **hoặc**
    - Gần chạm "rào mềm", **hoặc**
    - Bị kẹt va chạm (isBlocked(e) = collisionXOn || collisionYOn).
  - Reset counter, pick hướng mới:
    - Lọc bỏ các hướng sẽ đi ra ngoài bounds (wouldExceedBounds).

- Nếu đang gần rào: chọn hướng quay ngược về trong (pickInwardIfNearFence).
- Nếu không: random trong các hướng hợp lệ.

Kết quả: Slime/NPC di chuyển ngẫu nhiên trông tự nhiên, không rung và không vượt khỏi vùng cho phép.

4.5.3. ChaseMovement – AI đuổi theo Player

Lớp: ai.movement.ChaseMovement

Vai trò:

- Dùng cho quái "đuổi theo" người chơi khi đã bị kích hoạt aggro.
- Di chuyển mượt, tránh rung khi player chạy chéo.

Cấu hình:

```
public ChaseMovement(GamePanel gp,
                     Supplier<Player> targetSup,
                     int moveSpeed,
                     int stopRadiusPx)
```

- targetSup: hàm cung cấp Player (dùng Supplier<Player> để tránh phụ thuộc cứng, an toàn nếu player chưa spawn).
- moveSpeed: tốc độ đuổi.
- stopRadiusPx: khoảng cách an toàn, nếu quái đã ở trong bán kính này thì dừng lại (không dí sát vào tâm người chơi).

Các tham số chống giật:

- retargetEveryFrames = 8: mỗi 8 frame mới xét lại hướng.
- minHoldFrames = 8: giữ hướng tối thiểu.
- axisBiasPx = 6: "độ lệch" cần có để ưu tiên một trục (ngang/dọc).

Logic decide(Entity e):

- Lấy Player target từ targetSup.
- Tính dx, dy giữa quái và player, kiểm tra khoảng cách:
    - Nếu trong stopRadius → actualSpeed = 0, đứng lại.
- Nếu vẫn chạy:
    - Nếu còn trong minHoldFrames → giữ currentDir để tránh đổi hướng liên tục.
    - Cứ mỗi retargetEveryFrames:
        - So sánh |dx| và |dy| với một axisBiasPx:
            - Nếu |dx| > |dy| + bias → ưu tiên trục ngang (left/right).
            - Nếu |dy| > |dx| + bias → ưu tiên trục dọc (up/down).
            - Nếu chênh lệch nhỏ → giữ hướng cũ (tránh rung).

Nhờ giữ hướng + bias, quái đuổi player theo đường gấp khúc mượt mà, không "run rung" khi player chạy chéo.

4.5.4. AggroSwitchMovement – chuyển giữa idle và aggro

Lớp: ai.movement.AggroSwitchMovement

Vai trò:

- Kết hợp 2 controller:
    - idleController: AI khi đang "bình thường" (thường là WanderMovement).
    - aggroController: AI khi đã bị "kích hoạt" (thường là ChaseMovement).
- Dùng một Predicate<Entity> aggroCondition để quyết định khi nào nên bật/tắt trạng thái aggro (ví dụ: player trong bán kính X).

Các trường quan trọng:

- boolean isAggro: trạng thái hiện tại.

- onFrames = 3, offFrames = 6:
    - Số frame cần giữ điều kiện true để chuyển sang aggro.
    - Số frame cần giữ điều kiện false để thoát trạng thái aggro.
- onCount, offCount: bộ đếm frame để tránh bật/tắt liên tục (hysteresis).

Logic decide(Entity e):

1. Tính wantAggro = aggroCondition.test(e):
    - Nếu true: tăng onCount, giảm offCount.
    - Nếu false: tăng offCount, giảm onCount.
2. Chuyển trạng thái:
    - Nếu đang không aggro mà onCount >= onFrames → isAggro = true, reset offCount.
    - Nếu đang aggro mà offCount >= offFrames → isAggro = false, reset onCount.
3. Giao việc cho controller tương ứng:
    - Nếu isAggro == true → aggroController.decide(e).
    - Ngược lại → idleController.decide(e).

Ứng dụng thực tế (ví dụ trong OrcMonster):

- idleController = new WanderMovement(...)
- aggroController = new ChaseMovement(...)
- aggroCondition: player sống và nằm trong bán kính 6 * tileSize.

Nhờ đó, Orc:

- Bình thường chỉ đi lang thang quanh khu vực.

- Khi player tiến lại gần một lúc (không chỉ lướt qua trong 1 frame), Orc chuyển sang chế độ đuổi.
- Nếu player chạy xa khỏi vùng aggro đủ lâu, Orc sẽ "hạ nhiệt", quay về đi lang thang.

4.6. Tile & Chunk System (gói tile)

Để map lớn mà vẫn chạy mượt, game không dùng 1 mảng 2D khổng lồ cho toàn bộ bản đồ, mà chia nhỏ bản đồ thành **chunk** (khối 32×32 tile) và **streaming** các chunk nằm quanh người chơi.

Tầng Tile & Chunk gồm 3 thành phần:

- TileManager – quản lý tileset, ảnh và cờ collision của từng loại tile.
- ChunkManager – quản lý việc load/unload các chunk .tmx theo vị trí camera.
- CollisionChecker – dùng TileManager + ChunkManager để kiểm tra va chạm với nền (tile collision).

4.6.1. TileManager – quản lý tileset & collision

Lớp: tile.TileManager

Vai trò:

- Load tileset (ảnh chứa nhiều ô tile) từ file PNG.
- Tách tileset thành từng Tile riêng và lưu vào mảng Tile[] tile.
- Đọc file cấu hình .tsx để gán cờ collision cho từng loại tile.
- Vẽ tile ra màn hình dựa trên các Chunk đang active.
- Cung cấp API isCollisionAtWorld(...) để các hệ thống khác hỏi "ô nền tại vị trí (x, y) có phải là collision hay không?".

Cấu trúc:

```
public class Tile {
    public BufferedImage image;
```

```
    public boolean collision = false;
}
```

Load tileset:

```
public void loadTileset(String path, int tileSize) {
    BufferedImage tileset = ImageIO.read(...);
    int cols = tileset.getWidth() / tileSize;
    int rows = tileset.getHeight() / tileSize;
    tile = new Tile[cols * rows];

    // cắt từng ô tile từ tileset
    for (int y = 0; y < rows; y++) {
        for (int x = 0; x < cols; x++) {
            tile[index] = new Tile();
            tile[index].image = tileset.getSubimage(...);
            tile[index].collision = false;
            index++;
        }
    }
}
```

Load thuộc tính collision từ TSX:

```
public void loadTilesetProperties(String tsxPath) {
    // đọc file .tsx (XML) kèm tileset
    // tìm <tile id="..."> và property name="collision" value="true"
    // => gán tile[tileIndex].collision = true
}
```

Nhờ đó, level designer chỉ cần đánh dấu collision=true trong Tiled, game tự hiểu ô nào là tường, nước, vách đá không đi xuyên được.

Vẽ tile theo chunk:

```
public void draw(Graphics2D g2, ChunkManager chunkM) {
    int playerPosX = gp.em.getPlayer().worldX;
    int playerPosY = gp.em.getPlayer().worldY;

    // Tính vùng màn hình (screenLeft/right/top/bottom) theo player
```

```
    // ...

    for (Chunk c : chunkM.getActiveChunks()) {
        int chunkWorldX = c.chunkX * c.size * gp.tileSize;
        int chunkWorldY = c.chunkY * c.size * gp.tileSize;

        // Bỏ qua chunk nằm ngoài màn hình
        // ...

        // Vẽ từng tile trong chunk
        for (int row = 0; row < c.size; row++) {
            for (int col = 0; col < c.size; col++) {
                int tileNum = c.mapTileNum[row][col];
                // tính tileWorldX/tileWorldY → tileScreenX/tileScreenY
                // dùng tile[tileNum].image để vẽ
            }
        }

        // Vẽ khung đỏ quanh chunk để debug phân vùng
    }
}
```

API collision theo world:

```
public boolean isCollisionAtWorld(int worldX, int worldY, ChunkManager chunkM) {
    int tileNum = chunkM.getTileNumAtWorld(worldX, worldY);
    if (tileNum < 0 || tileNum >= tile.length) return false;
    return tile[tileNum].collision;
}
```

Đây là hàm mà CollisionChecker sẽ gọi để kiểm tra nền tại 1 điểm world có cứng hay không.

4.6.2. ChunkManager – streaming map theo chunk

Lớp: tile.ChunkManager

Vai trò:

- Chia bản đồ Tiled thành nhiều file .tmx nhỏ: chunkX_Y.tmx, mỗi chunk kích thước chunkSize × chunkSize tile (ví dụ 32×32).

- Load và giữ trong RAM chỉ những chunk gần người chơi, các chunk xa sẽ được unload.
- Cung cấp API để truy vấn tile ID tại 1 ô bất kỳ (getTileNum, getTileNumAtWorld) phục vụ vẽ và collision

Cấu trúc dữ liệu:

- Chunk:

```
public class Chunk {
    public int chunkX, chunkY;
    public int[][] mapTileNum;
    public int size;
}
```

Trong ChunkManager:

```
private final int chunkSize;
private final Map<String, Chunk> chunks;
private final GamePanel gp;
private ExecutorService loader = Executors.newSingleThreadExecutor();
public String pathMap = "map0";
```

Việc load chunk được thực hiện thông qua:

```
private ExecutorService loader = Executors.newSingleThreadExecutor();
```

- loader là một luồng riêng (background thread) chuyên dùng để đọc file .tmx và parse dữ liệu tile.
- Luồng này chạy song song với luồng chính (main game loop đang update + render 60 FPS).
- Nhờ đó, khi cần load thêm chunk mới (do người chơi di chuyển sang vùng map khác), game không bị khựng hình vì phải chờ I/O đọc file.

Cụ thể:

- Luồng chính chỉ gọi loadChunkAsync(...), việc nặng như:
    - mở file .tmx,
    - đọc XML,
    - parse <data encoding="csv"> thành mảng số,
    - gán vào mapTileNum
      sẽ được đẩy sang ExecutorService xử lý ở background.
- Cách làm này tận dụng tốt CPU nhiều nhân, nhiều luồng trên máy hiện đại:
    - một core lo logic + render game,
    - một core khác lo load map nền.
- Nếu toàn bộ công việc (update, render, load map, đọc file, tính toán nặng) đều dồn vào một luồng duy nhất thì:
    - CPU bị dùng không tối ưu,
    - chỉ cần load map hoặc đọc file lâu là FPS tụt, game bị giật.

Nhờ tách nhỏ như vậy, kiến trúc dễ mở rộng: sau này nếu game lớn hơn (map nhiều, asset nặng), có thể nâng từ newSingleThreadExecutor() lên thread pool rộng hơn mà không phải thay đổi logic GamePanel hay các hệ thống khác.

Load 1 chunk từ file TMX:

```
private Chunk loadChunkFromFile(int chunkX, int chunkY, String pathMap) {
    String path = "/" + pathMap + "/chunk" + chunkX + "_" + chunkY + ".tmx";
    InputStream is = getClass().getResourceAsStream(path);
    // đọc XML, lấy phần <data encoding="csv">...</data>
    // parse CSV -> numbers[]
    // mapTileNum[row][col] = num - 1 (0-based index cho Tile[])
}
```

Load async theo camera:

```
public void loadChunkAsync(int chunkX, int chunkY, String pathMap) {
    // nếu key đã có trong map thì bỏ qua
    loader.submit(() -> {
        Chunk c = loadChunkFromFile(chunkX, chunkY, pathMap);
        if (c != null) chunks.put(key, c);
    });
}
```

Cập nhật các chunk active quanh player:

```
public void updateChunks(int playerWorldX, int playerWorldY) {
    int buffer = gp.tileSize * (chunkSize / 2);

    // tính screenLeft, screenRight, screenTop, screenBottom
    // dựa trên player.worldX, player.screenX, buffer...

    int chunkLeft   = screenLeft   / (chunkSize * gp.tileSize);
    int chunkRight  = screenRight  / (chunkSize * gp.tileSize);
    int chunkTop    = screenTop    / (chunkSize * gp.tileSize);
    int chunkBottom = screenBottom / (chunkSize * gp.tileSize);

    // For từng chunk trong vùng [chunkLeft..chunkRight] x
[chunkTop..chunkBottom]
    // gọi loadChunkAsync() nếu chưa có
    // Sau đó gọi unloadFarChunks(...) để xoá các chunk quá xa
}
```

- unloadFarChunks(left, right, top, bottom):
    - Giữ lại các chunk nằm trong vùng [left-1 … right+1] × [top-1 … bottom+1].
    - Các chunk xa hơn bị remove khỏi map → giải phóng RAM.

Truy vấn tile tại 1 ô / world:

```
public int getTileNum(int tileCol, int tileRow) {
    // tính chunkX, chunkY từ tileCol, tileRow
    // chuyển về toạ độ trong chunk: inChunkCol, inChunkRow
    // lấy Chunk c = getChunk(chunkX, chunkY)
    // trả về c.mapTileNum[inChunkRow][inChunkCol] hoặc 0 nếu chưa load
}
```

```
public int getTileNumAtWorld(int worldX, int worldY) {
    int tileCol = worldX / gp.tileSize;
    int tileRow = worldY / gp.tileSize;
    return getTileNum(tileCol, tileRow);
}
```

Đổi map:

```
public void loadMap(String mapName) {
    clearChunks();      // xoá chunk cũ
    this.pathMap = mapName;
}
```

Ngoài ra có loadAllChunksSync() dùng trong các đoạn code cần scan toàn map (ví dụ khởi tạo điểm spawn quái trong MonsterManager).

4.6.3. CollisionChecker – va chạm tile

Lớp: tile.CollisionChecker *(code không dán ở đây, nhưng cấu trúc chung như sau)*

Vai trò:

- Kiểm tra va chạm giữa Entity / Object với nền tile dựa trên:
    - TileManager (biết tile nào collision).
    - ChunkManager (biết tile ID ở vị trí world nào).
- Cài đặt các hàm:
    - checkTile(Entity e, int nextX, int nextY) – kiểm tra nếu Entity di chuyển tới (nextX,nextY) có đâm vào tile collision không.
    - Các hàm khác: checkWorldObject, checkPlayer, checkEntity… (phần này liên quan va chạm entity/object, đã dùng ở EntityMovement & Combat).

Nguyên lý hoạt động checkTile:

1. Lấy hitbox thân của Entity (solidArea) tại vị trí dự kiến:

- nextX = e.worldX + dx, nextY = e.worldY + dy.

- Xét 4 cạnh của solidArea tại tọa độ world: leftWorldX, rightWorldX, topWorldY, bottomWorldY.

2. Chuyển sang tọa độ tile:

- tileCol = worldX / gp.tileSize, tileRow = worldY / gp.tileSize.

3. Dùng chunkM.getTileNum(...) + tileM.tile[tileNum].collision:

    - Nếu bất kỳ tile nào trong vùng solidArea là collision:

        - Set cờ e.collisionOn = true (hoặc collisionXOn, collisionYOn tùy chiều).

4. Trong EntityMovement.moveWithDelta(...), nếu !e.collisionOn thì mới cập nhật e.worldX/Y.

Nhờ vậy:

- Hệ thống map sử dụng chunk streaming nhưng từ góc nhìn của AI / Movement, chỉ cần gọi cChecker.checkTile(...) → không phải xử lý trực tiếp các file .tmx.

- Việc đổi map (map0, map1, map3…) chỉ cần chunkM.loadMap("mapX"), các hàm collision vẫn hoạt động như cũ.

4.7. Save/Load System (gói game_data – lớp SaveManager)

Hệ thống Save/Load được thiết kế tách riêng khỏi logic game, gồm:

- Lớp dữ liệu thuần (data model): GameData, PlayerData, ObjectData.

- Bộ điều phối lưu/khôi phục: SaveManager – đọc/ghi JSON bằng thư viện Gson.

- Mục tiêu:

- Lưu trạng thái game về file JSON để dễ debug, dễ mở rộng.

- Không serialize trực tiếp Player, Monster, GamePanel (dễ lỗi, vòng tham chiếu), mà chỉ lưu những trường cần thiết.
- Hỗ trợ khôi phục lại:
    - Vị trí, máu, level, EXP, vũ khí của người chơi.
    - Map hiện tại (chỉ số map và path map0, map1, …).
    - Trạng thái quái (vị trí, còn sống/chết).

4.7.1. Data model: GameData, PlayerData, ObjectData

PlayerData – dữ liệu người chơi:

```
public class PlayerData {
    public int worldX, worldY;
    public int health, maxHealth;
    public String weaponName;
    public int mapIndex;
    public int exp, level;
}
```

- Lưu lại vị trí player trong world, máu hiện tại/tối đa.
- Lưu weaponName để sau khi load có thể tạo lại đúng loại vũ khí (Axe, Sword, Pick…).
- Lưu mapIndex (map hiện tại), cùng với exp và level để khôi phục tiến trình nhân vật.

ObjectData – dữ liệu quái/vật thể động:

```
public class ObjectData {
    public String type;
    public int worldX, worldY;
    public boolean active;
}
```

- type: tên loại (ví dụ tên quái) – phục vụ mở rộng sau này nếu cần respawn đúng loại.

- worldX, worldY: toạ độ.
- active: trạng thái còn sống/hoạt động hay không (ví dụ quái đã bị giết thì active = false).

GameData – gói toàn bộ state cần lưu:

```
public class GameData {
    public PlayerData player;
    public List<ObjectData> objects;
    public int mapIndex;
    public String mapPath;
}
```

- player: toàn bộ thông tin người chơi.
- objects: danh sách ObjectData (ở đây chủ yếu dùng cho quái).
- mapIndex: chỉ số map hiện tại.
- mapPath: chuỗi đường dẫn map ("map0", "map1", …) để đồng bộ với ChunkManager.

Các lớp này chỉ chứa trường public, không có logic, nên rất phù hợp để Gson serialize/deserialize sang JSON.

4.7.2. SaveManager – luồng lưu game (saveGame)

Lớp: game_data.SaveManager

- Khởi tạo thư viện Gson với setPrettyPrinting() để file JSON dễ đọc.
- Đảm bảo thư mục lưu tồn tại:

```
private static final String SAVE_DIR = "saves";
private static final String SAVE_FILE = "savegame.json";
```

Bước lưu game (saveGame(GamePanel gp)):

1. Chuẩn bị thư mục và player:

- Lấy Player player = gp.em.getPlayer().
- Nếu player null → in log lỗi và dừng.

2. Đóng gói PlayerData:

```
PlayerData playerData = new PlayerData(
    player.worldX,
    player.worldY,
    player.getHP(),
    player.getMaxHP(),
    (player.getCurrentWeapon() != null ? player.getCurrentWeapon().name :
null),
    gp.currentMap,
    player.getExp(),
    player.getLevel()
);
```

3. Đóng gói danh sách quái (ObjectData):

- Lấy danh sách quái từ EntityManager theo gp.currentMap.
- Với mỗi Monster:
  - Lưu mon.name, mon.worldX, mon.worldY.
  - active = !mon.isDead() để biết quái đã bị giết hay chưa.

4. Lưu thông tin map:

- mapIndex = gp.currentMap
- mapPath = gp.chunkM.pathMap (ví dụ "map1").

5. Tạo GameData và ghi ra JSON:

```
GameData data = new GameData(playerData, monsterList, mapIndex, mapPath);
String json = gson.toJson(data);
Files.write(Paths.get(SAVE_DIR, SAVE_FILE), json.getBytes());
```

6. In log [SaveManager] Game saved successfully. nếu thành công, hoặc in stacktrace nếu có exception.

   Kết quả: trạng thái game được lưu vào saves/savegame.json, thuận tiện cho việc debug và chỉnh tay nếu cần.

4.7.3. SaveManager – luồng load game (loadGame)

Bước load game (loadGame(GamePanel gp)):

1. Đọc file JSON:
   - Kiểm tra tồn tại saves/savegame.json.
   - Nếu không có → in log và không làm gì (game chạy như new game).
   - Đọc toàn bộ nội dung file và parse bằng Gson:

```
GameData data = gson.fromJson(json, GameData.class);
```

2. Khôi phục Player:

```
Player player = gp.em.getPlayer();
if (player != null && data.player != null) {
    player.worldX = data.player.worldX;
    player.worldY = data.player.worldY;

    // exp & level
    player.setExp(data.player.exp);
    player.setLevel(data.player.level);

    // stats + máu
    player.setStats(data.player.maxHealth, player.getATK(), player.getDEF());
    player.setHP(data.player.health);
}
```

3. Khôi phục vũ khí:
   - Dựa vào data.player.weaponName, tạo lại đúng Weapon:

```
switch (data.player.weaponName) {
    case "Leviathan Axe" -> w = new Axe(gp, data.mapIndex);
```

```
    case "Argonaut Hero's Sword" -> w = new Sword(gp, data.mapIndex);
    case "Steve Pick" -> w = new Pick(gp, data.mapIndex);
}
```

- Gán lại cho player:

```
player.setCurrentWeapon(w);
player.equipWeapon(w); // để CombatComponent + animation dùng thông số của
weapon
```

4. Khôi phục map & Chunk/Objects:

- Đặt lại map hiện tại:

```
gp.currentMap = data.player.mapIndex;
player.mapIndex = gp.currentMap;
String newMap = "map" + gp.currentMap;

gp.chunkM.loadMap(newMap);
if (gp.om != null) {
    gp.om.reloadMapObjects(gp.currentMap);
}
gp.em.update(gp.currentMap);
```

Nhờ đó, world tiles & objects được reset theo map tương ứng với save.

5. Khôi phục quái:

- Lấy danh sách quái hiện tại của map (gp.em.getMonsters(gp.currentMap)).
- Với từng cặp entity – ObjectData tương ứng:
    - Set mon.worldX, mon.worldY theo data.
    - Nếu saved.active == false → mon.setHP(0) (coi như đã chết).
    - Nếu saved.active == true → mon.revive() để hồi full HP.

6. Khôi phục hệ thống interact:

- Sau khi player đã được load, tạo lại Interact:

```
if (gp.em != null && gp.em.getPlayer() != null) {
```

```
    gp.iR = new Interact(gp, gp.em.getPlayer(), gp.em.getPlayer().input);
}
```

7. In log: [SaveManager] Game loaded successfully. nếu không có lỗi.

4.8. Sound System (gói sound_manager)

Hệ thống âm thanh được tách riêng trong gói sound_manager, gồm hai lớp chính:

- Sound – wrapper đơn giản quanh Clip của Java, chịu trách nhiệm tải và phát 1 file âm thanh.
- SoundManager – Singleton điều phối nhạc nền và hiệu ứng âm thanh (BGM + SFX) cho toàn game.

Nhờ tách riêng, các module khác (Player, UI, Object…) chỉ cần gọi SoundManager.getInstance().playSE(...) hoặc playMusic(...) mà không phải quan tâm chi tiết về Clip, AudioInputStream…

4.8.1. Lớp Sound – wrapper cho Clip

Lớp: sound_manager.Sound

```
public class Sound {
    private Clip clip;

    public void setFile(URL url) { ... }
    public void play() { ... }
    public void loop() { ... }
    public void stop() { ... }
}
```

- setFile(URL url)
    - Đọc file .wav từ resource (AudioSystem.getAudioInputStream(url)).
    - Khởi tạo Clip và clip.open(ais).
    - Nếu lỗi (file sai đường dẫn / lỗi IO) → in stacktrace để dễ debug.
- play()

- Đặt lại frame về đầu (clip.setFramePosition(0)) rồi clip.start().
- Dùng cho âm thanh ngắn (SFX) hoặc 1 lần phát BGM.

- loop()
  - Đặt lại từ đầu, rồi clip.loop(Clip.LOOP_CONTINUOUSLY).
  - Dùng cho nhạc nền chạy lặp liên tục.
- stop()
  - Nếu clip đang chạy (clip.isRunning()) thì clip.stop().
  - Dùng khi chuyển scene, dừng BGM hoặc cắt SFX.

Lớp Sound được thiết kế nhỏ gọn, chỉ bao bọc thao tác thường dùng với Clip, giúp code phía trên gọn hơn và không lặp lại.

4.8.2. SoundManager – Singleton điều phối BGM & SFX

Lớp: sound_manager.SoundManager

- Áp dụng Singleton pattern:

```
private static SoundManager instance;

public static SoundManager getInstance() {
    if (instance == null) instance = new SoundManager();
    return instance;
}
```

Cả game chỉ có 1 đối tượng SoundManager, tránh việc tạo nhiều Clip không cần thiết và khó quản lý.

a) Danh sách ID âm thanh – enum SoundID

```
    public enum SoundID {

 MUSIC_THEME,   // nhạc nền chính
 HIT,           // hiệu ứng va chạm (đang để sẵn)
```

```
    DEAD,           // hiệu ứng chết
    COIN,           // nhặt tiền/vật phẩm
    UNLOCK          // mở cửa, mở khoá
}
```

b) Quản lý file và clip

Trong SoundManager:

```
private final EnumMap<SoundID, URL> soundFiles = new EnumMap<>(SoundID.class);

private final Sound music = new Sound();  // nhạc nền (loop)
private final Sound se    = new Sound();  // SFX (1 phát)
private boolean isMusicPlaying = false;
```

- soundFiles: map từ SoundID → URL resource trong thư mục /sound.
- music: đối tượng Sound riêng cho nhạc nền (có thể loop).
- se: đối tượng Sound chuyên dùng cho hiệu ứng ngắn (đá, nhặt đồ…).

Khởi tạo đường dẫn trong constructor:

```
public SoundManager() {
    soundFiles.put(SoundID.MUSIC_THEME,
getClass().getResource("/sound/ThemeMusic.wav"));
    soundFiles.put(SoundID.COIN,        getClass().getResource("/sound/coin.wav")
);
    soundFiles.put(SoundID.UNLOCK,      getClass().getResource("/sound/unlock.wav
"));
    // (các sound khác có thể thêm sau)
}
```

c) Điều khiển nhạc nền (BGM)

```
public void playMusic(SoundID id) {
    URL url = soundFiles.get(id);
    if (url != null) {
        if (isMusicPlaying) return;   // tránh play chồng nhiều lần
        music.setFile(url);
        music.play();
        music.loop();
        isMusicPlaying = true;
```

```
    }
}

public void stopMusic() {
    music.stop();
    isMusicPlaying = false;
}
```

- playMusic:
    - Nhận SoundID (thường là MUSIC_THEME).
    - Nếu BGM đã chạy (isMusicPlaying == true) thì không tạo mới nữa, tránh chồng nhiều clip.
    - Gọi music.setFile(...), music.play() và music.loop() để BGM lặp vô hạn.
- stopMusic:
    - music.stop(), reset cờ isMusicPlaying.
    - Dùng khi vào màn hình khác, màn hình game over… để dừng nhạc hiện tại.

d) Phát hiệu ứng âm thanh (SFX)

```
public void playSE(SoundID id) {
    URL url = soundFiles.get(id);
    if (url != null) {
        se.setFile(url);
        se.play();
    }
}
```

- Dùng cho hiệu ứng như:
    - Nhặt đồ: SoundManager.getInstance().playSE(SoundID.COIN);
    - Mở cửa: SoundManager.getInstance().playSE(SoundID.UNLOCK);
- Mỗi khi gọi playSE, Sound se sẽ:
    - Load file tương ứng, reset frame về đầu, phát 1 lần.

4.9. UI System (packet ui)

Hệ thống UI được tách riêng khỏi logic gameplay, tổ chức quanh lớp trừu tượng BaseUI và bộ điều phối UIManager. Mỗi loại giao diện (HUD, màn hình menu, pause, game over, hiệu ứng…) là một lớp con của BaseUI và tự khai báo khi nào được update / vẽ dựa trên GameState. Điều này giúp UI dễ mở rộng, dễ bật/tắt theo state mà không làm rối GamePanel.

4.9.1. UIManager – điều phối toàn bộ UI

- BaseUI là lớp cơ sở cho tất cả UI:
    - update() – logic cập nhật (counter, hiệu ứng, input…).
    - draw(Graphics2D g2) – vẽ UI lên màn hình.
    - shouldRenderIn(GameState state) – UI xuất hiện ở state nào (PLAY, PAUSE, START, GAME_OVER, DIALOGUE…).
    - shouldUpdate() / shouldDraw() – có cho update / vẽ ở frame hiện tại hay không.
- UIManager giữ một list các BaseUI:

```
private final List<BaseUI> uiList = new ArrayList<>();
public void add(BaseUI ui) { uiList.add(ui); }
```

- Quy trình hoạt động:
    - Trong update(GameState state):
        - Duyệt qua tất cả UI, UI nào shouldRenderIn(state) → gọi ui.update().
    - Trong draw(Graphics2D g2, GameState state):
        - Chỉ vẽ những UI thỏa mãn shouldRenderIn(state) và shouldDraw().
- UIManager cung cấp hàm get(Class<T> type) để các hệ thống khác (vd: Interact, ObjectManager, …) truy cập nhanh một UI cụ thể như MessageUI, DialogueUI mà không cần giữ reference riêng.

Kết quả: UI được tách khỏi game loop chính, có "life cycle" riêng theo GameState.

4.9.2. HUD: HealthUI, PlayerStatusUI, MonsterHealthUI

Các HUD này đều là BaseUI và chủ yếu hiển thị ở state PLAY.

a) HealthUI – thanh máu Player

- Đọc máu hiện tại và tối đa từ gp.em.getPlayer().
- Vẽ một thanh ngang ở góc trên trái:
    - Nền xám, phần đỏ thể hiện tỷ lệ HP / MaxHP.
    - Có viền trắng để rõ ràng.
- shouldRenderIn(GameState state) → chỉ vẽ khi state == PLAY.
- shouldUpdate() = false vì thanh máu được vẽ dựa trên giá trị hiện tại của Player, không có logic riêng.

b) PlayerStatusUI – Level + EXP

- Hiển thị thông tin tiến triển của Player ngay dưới thanh máu:
    - Lv X
    - EXP current / expToNext
    - Thanh EXP nhỏ thể hiện tỷ lệ phần trăm.
- Layout:
    - Một panel nền mờ, bo góc, có viền trắng.
    - Text level + EXP ở phía trên, thanh EXP mảnh ở phía dưới panel.
- shouldRenderIn:

    - Cho phép hiển thị trong PLAY và PAUSE để người chơi luôn xem được level/exp.

- shouldUpdate() = true (dù hiện tại chưa có logic phức tạp, nhưng sẵn sàng mở rộng sau).

c) MonsterHealthUI – thanh máu trên đầu quái

- Duyệt gp.em.getMonsters():
    - Bỏ qua entity không phải Monster hoặc đã chết.
- Tính vị trí màn hình của từng quái dựa trên:
    - monster.worldX/Y và vị trí Player để chuyển world → screen.
- Vẽ thanh máu nhỏ trên đầu mỗi quái:
    - Nền xám đậm, phần đỏ ứng với HP / MaxHP.
    - Có viền đen mảnh.
- Chỉ vẽ trong GameState.PLAY, không cần update độc lập.

Nhóm HUD cung cấp thông tin sinh tồn và tiến triển của Player và quái, bám sát gameplay mà vẫn tách khỏi logic chiến đấu.

4.9.3. Màn hình: MainMenuUI, PauseOverlay, GameOverUI

Các màn hình lớn (full-screen overlay) đều là BaseUI riêng, hiển thị/ẩn hoàn toàn dựa trên GameState.

a) MainMenuUI – màn hình chính

- Chỉ hiển thị ở GameState.START.
- Tải background và atlas nút bằng LoadSave.
- Có 4 nút:
    - Play, Load, Quit, Credits.

- Điều hướng bằng phím (thay đổi focusIndex), khi select():
    - Play:
        - gp.gsm.setState(GameState.PLAY);
        - Bật nhạc nền qua SoundManager.playMusic(MUSIC_THEME).
    - Load:
        - Gọi gp.saveManager.loadGame(gp) nếu có file.
        - Chuyển sang PLAY + bật nhạc nền.
    - Quit:
        - System.exit(0);
    - Credits:
        - Bật showingCredits = true, vẽ màn hình credit riêng.
- handleKey(int code) cho phép thoát khỏi màn hình Credits bằng ESC.

b) PauseOverlay – menu tạm dừng

- Hiển thị khi GameState.PAUSE.
- Vẽ một khung pause ở giữa màn hình + các nút:
    - SoundButton (mute/unmute music).
    - Ba nút URM (UrmButton):
        - Menu – về màn hình START.
        - Replay – restart game.
        - Resume – tiếp tục chơi.
- Dùng focusIndex để di chuyển focus nút (trái/phải) và select() để kích hoạt:
    - toggleMusic() – tắt/bật nhạc nền qua SoundManager.

    - goMenu() – gp.gsm.setState(START).
    - restart() – gp.restartGame(), bật lại nhạc nếu chưa mute.
    - resume() – trở về PLAY.
- highlightFocusedButton(Graphics2D) vẽ một khối sáng mờ xung quanh nút đang được chọn, giúp feedback cho điều khiển bằng bàn phím.

c) GameOverUI – màn hình thua

- Hiển thị khi GameState.GAME_OVER.
- Vẽ overlay đen mờ + chữ "Game Over" lớn ở giữa màn hình.
- Có 2 lựa chọn:
    - Restart → gọi gp.restartGame();
    - Quit → gp.gsm.setState(START);
- Điều hướng bằng commandNum (0/1), hàm moveUp() / moveDown() và select().

Nhóm màn hình này dùng chung cơ chế: toggle bằng GameState, bọc toàn màn hình, điều hướng bằng phím, và tách hẳn khỏi loop gameplay.

4.9.4. Hiệu ứng: FadeUI, MessageUI, DialogueUI

Nhóm này tạo cảm giác "mượt" và giàu feedback khi chơi.

a) FadeUI – hiệu ứng fade in/out toàn màn hình

- Có biến alpha (0 → 1) và phase:
    - phase = 1: fade-out (tăng alpha).
    - phase = 2: fade-in (giảm alpha).
- startFade(Runnable onComplete):
    - Bắt đầu một chuỗi fade-out → chạy onComplete.run() → fade-in.
    - Dùng cho các tình huống như đổi map, restart hoặc chuyển scene.

- Trong draw():
    - Vẽ một hình chữ nhật đen full screen với độ trong suốt theo alpha.

b) MessageUI – thông báo ngắn (touch message)

- Dùng để hiển thị thông báo tạm thời như:
    - “press 'F' to heal health”
    - “That close!”
- showTouchMessage(text, obj, gp):
    - Set message, bật show = true, reset counter.
- update():
    - Tăng counter, sau 120 frame thì tự tắt (ẩn).
- draw():
    - Vẽ text ở một vị trí cố định (vd: hàng thứ 5 tính từ trên).
- shouldDraw() – chỉ return true khi show == true.

c) DialogueUI – hộp thoại NPC

- Điều khiển state GameState.DIALOGUE:
    - Khi startDialogue(String[] src):
        - Set danh sách lines, reset chỉ số dòng/ký tự.
        - Chuyển GameState sang DIALOGUE.
    - isActive() kiểm tra xem đang ở state DIALOGUE và còn dòng để hiển thị.
- Hiệu ứng gõ chữ từng ký tự (typewriter):
    - Mỗi update() tăng frame; đến một ngưỡng (SPEED) thì thêm 1 ký tự vào text.

    - Khi hết ký tự, finished = true.

- Input:
    - Lấy input isTalkPressed() từ Player.
    - Nhấn phím lần đầu:
        - Nếu chưa in xong dòng → in toàn bộ ngay.
        - Nếu dòng đã in xong → chuyển sang dòng tiếp theo (nextLine()).
    - prevPressed dùng để chống spam, đảm bảo mỗi lần nhấn chỉ xử lý 1 sự kiện.
- Khi hết tất cả dòng → endDialogue():
    - lines = null, reset trạng thái.
    - Chuyển GameState về PLAY.
    - Gọi p.input.resetTalkKey() để tránh NPCInteract đọc lại phím E ngay frame tiếp theo.
- draw():
    - Vẽ khung hộp thoại mờ ở đáy màn hình.
    - Text xuống dòng tự động bằng drawMultiline(...).
    - Khi dòng hiện tại in xong, hiển thị hint:
        - [E] to continue nếu còn dòng sau.
        - [E] to close nếu là dòng cuối cùng.

Nhờ các hiệu ứng này, game không chỉ có HUD + menu khô khan, mà còn có hội thoại, fade chuyển cảnh, thông báo nhỏ giúp người chơi dễ hiểu tương tác hơn.

CHƯƠNG 5. KẾT QUẢ DEMO VÀ HƯỚNG DẪN CÀI ĐẶT

5.1. Môi trường thực nghiệm

Hệ thống được xây dựng và kiểm thử trong môi trường sau:

- Hệ điều hành: Windows 10 / Windows 11 (64-bit)
- Ngôn ngữ lập trình: Java
- Phiên bản JDK sử dụng: JDK 24 (có thể tương thích từ JDK 17 trở lên)
- IDE phát triển: IntelliJ IDEA và NetBeans
- Thư viện ngoài:
    - gson-2.10.1.jar – phục vụ đọc/ghi JSON cho hệ thống lưu game (SaveManager)
- Cấu hình máy thử nghiệm (tham khảo):
    - CPU: Intel Core i5 / Ryzen 5 trở lên
    - RAM: ≥ 8 GB
    - GPU: Onboard là đủ (vì game dùng Java2D)

Môi trường trên đảm bảo game chạy ổn định ở 60 FPS với kích thước màn hình và số lượng quái như cấu hình hiện tại.

Game đã có bản build .exe hoàn toàn có thể chạy khi tải cả folder game

Link github quá trình thiết kế game: https://github.com/duongakrapovic/2DJavaGame

Link github báo cáo: https://github.com/BrokenDev375/Bao-Cao-OOP-Java-PTIT.git

5.2. Hướng dẫn cài đặt và chạy phiên bản EXE

5.2.1. Cấu trúc khi chạy bản EXE

Sau khi giải nén thư mục release, người chơi sẽ thấy dạng:

2DJavaGame/

├── Game.exe

├── resources/

│ ├── maptiles/

│ ├── map0/ map1/ map2/ map3/

│ ├── player/

│ ├── monster/

│ ├── sound/

│ ├── ui/

│ └── ...

├── saves/

│ └── savegame.json (sinh ra sau khi Save)

└── README.txt (nếu có)

Lưu ý: Không được xoá thư mục resources/. EXE cần dùng ảnh, map, âm thanh ở đây.

5.2.2. Cách chạy game

Bước 1: Tải và giải nén thư mục game.

Bước 2: Nhấp đúp chuột vào file:

Game.exe

Game sẽ khởi động trực tiếp vào Main Menu, không yêu cầu cài đặt thêm.

Không cần JDK, không cần IDE → thuận tiện cho giảng viên và người chơi.

5.2.3. Lỗi phổ biến & cách khắc phục

1. Mở EXE nhưng không chạy

   Kiểm tra Windows SmartScreen, chọn “Run Anyway”.

2. Chạy nhưng mất ảnh/map/sound

   Đảm bảo thư mục resources/ nằm cùng cấp với Game.exe.

3. Load Game lỗi

   Xoá file saves/savegame.json, game sẽ tạo file mới.

5.3. Hướng dẫn chạy game từ mã nguồn (dành cho người phát triển)

Ngoài bản EXE, người dùng có thể chạy trực tiếp từ mã nguồn Java. Cách thực hiện như sau:

5.3.1. Yêu cầu

- Đã cài sẵn JDK 17 trở lên (nhóm sử dụng JDK 24 khi phát triển).
- Đã cài một IDE Java, ví dụ:
  - IntelliJ IDEA (khuyến nghị)
  - Hoặc NetBeans

5.3.2. Mở project bằng IntelliJ IDEA

1. Mở IntelliJ IDEA → File → Open…
2. Chọn thư mục chứa project 2DJavaGame (thư mục có src/, resources/, libraries/…).
3. Chọn JDK cho project:
   - Vào File → Project Structure → Project
   - Chọn Project SDK = JDK 17+ (nhóm dùng JDK 24).
4. Đảm bảo thư mục resources/ được IDE nhận là Resource Root (để load ảnh, map, sound đúng đường dẫn).

5.3.3. Cài đặt thư viện (Gson)

Vì project được kéo từ GitHub về, IDE sẽ không tự nhận classpath cho Gson. Người dùng cần thêm file JAR thủ công:

Trên IntelliJ IDEA

1. Mở menu: File → Project Structure…
2. Chọn mục Libraries.

3. Nhấn dấu + → chọn Java.
4. Trỏ tới file:
   2DJavaGame/libraries/gson-2.10.1.jar
5. Nhấn Apply → OK để lưu cấu hình.

Trên NetBeans

1. Chuột phải vào tên Project → Properties.
2. Chọn mục Libraries.
3. Nhấn Add JAR/Folder.
4. Chọn file:
   libraries/gson-2.10.1.jar
5. Nhấn OK để hoàn tất.

Sau khi thêm Gson, project sẽ build được phần SaveManager và các class sử dụng JSON mà không báo lỗi thiếu thư viện.

5.4.4. Chạy game từ class Main

1. Trong IDE, tìm class main.Main.
2. Click chuột phải vào Main.java → Run 'Main.main()'.
3. Cửa sổ game được mở, vào Main Menu và chọn PLAY để bắt đầu chơi.
   Các phím điều khiển giống như mục 5.3.

5.4. Hướng dẫn chơi nhanh

- W A S D – Di chuyển
- J / – Tấn công
- E , F– Tương tác / nói chuyện
- F5 để save game

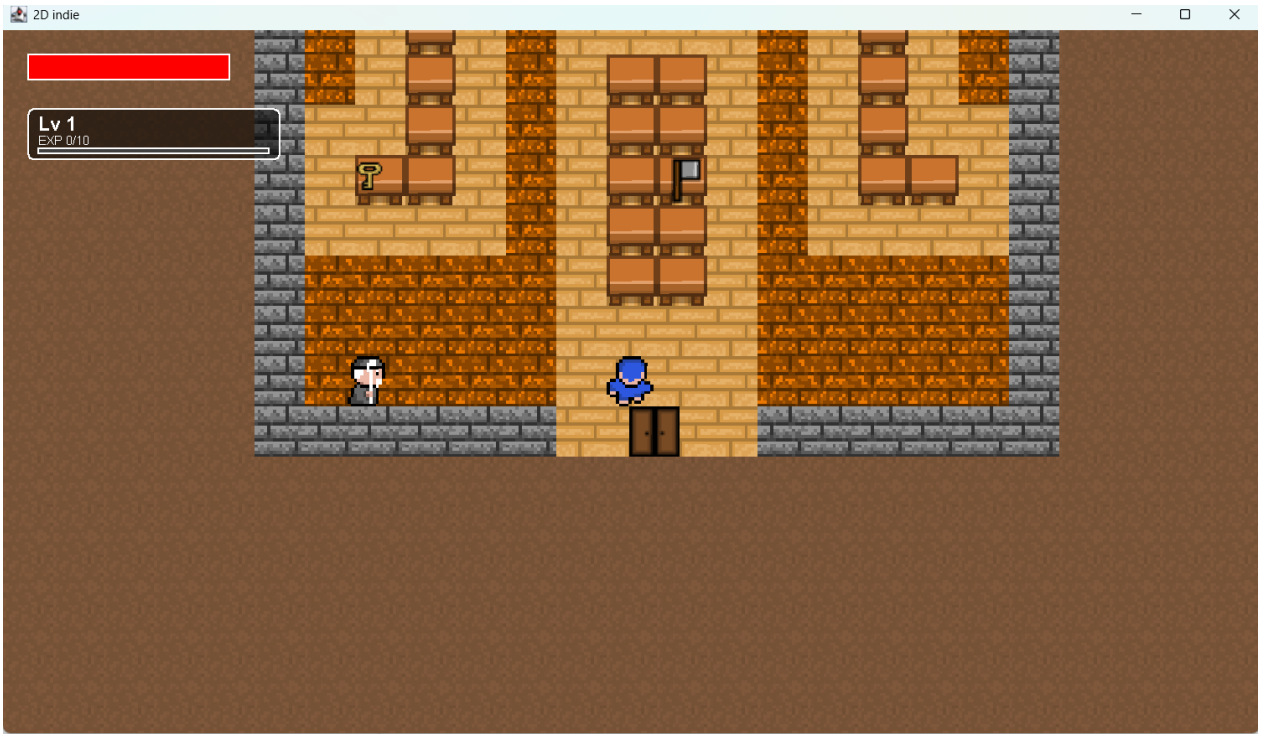
- ESC – Mở Pause Menu
- PLAY – Bắt đầu chơi
- OPTION – Tải lại file save
- CREDITS – Hiển thị thành viên nhóm

Game hỗ trợ:

- Combat có 3 phase tấn công
- Knockback
- AI đuổi/chạy/lảng vảng
- Map chia theo Chunk tải động
- UI có Menu, Pause, GameOver
- Save/Load bằng JSON;
- Âm thanh nền + hiệu ứng sound

5.5. Hình ảnh demo chương trình

Hình 2: Menu trò chơi



Hình 3: Map shop

Hình 4: Hội thoại với NPC

Hình 5: Combat với quái

Hình 6: Menu pause

Hình 7: Hình ảnh thông báo game saved

Hình 8: Hình ảnh game over

CHƯƠNG 6. KẾT LUẬN – HƯỚNG PHÁT TRIỂN VÀ CÁC TÀI LIỆU THAM KHẢO

6.1. Kết luận

Sau quá trình xây dựng và hoàn thiện dự án 2D Java RPG Game, nhóm đã đạt được nhiều kết quả quan trọng, cả về mặt kỹ thuật lẫn kiến thức nền tảng:

- Nắm bắt cách tổ chức một dự án lớn theo mô hình hướng đối tượng (OOP): hiểu cách tách module, gói (package) và quản lý luồng tương tác giữa các hệ thống trong game.
- Áp dụng thành công Java Graphics2D để dựng hình, render map, animation, UI và xử lý va chạm.
- Vận dụng các Design Pattern: Singleton (SoundManager), Strategy (AI Movement), State Machine (GameState), Component (Combat), v.v.
- Xây dựng một game hoàn chỉnh từ đầu, bao gồm:
    - Combat có phase Windup – Active – Recovery
    - AI di chuyển (Wander, Chase, AggroSwitch)
    - Tile & Chunk Loading động để tối ưu hiệu năng
    - UI hệ thống: Menu, Pause, GameOver, Dialogue, HUD
    - Save/Load bằng JSON
    - Object/Item/Weapon/Portal/Shop…
- Phát hành được bản Game dạng .exe, chạy độc lập không cần IDE hay JDK — thuận tiện cho việc demo và phân phối.

Dự án không chỉ giúp nhóm củng cố kiến thức lập trình mà còn tiếp cận gần hơn với quy trình phát triển game thực tế.

6.2. Hướng phát triển trong tương lai

Trong các phiên bản tiếp theo, nhóm có thể mở rộng và cải tiến hệ thống theo các hướng sau:

6.2.1. Thêm nội dung gameplay

- Hệ thống nhiệm vụ (Quest System)
- Bổ sung quái mới, boss mới, nhiều hành vi AI hơn
- Thêm kỹ năng chủ động (Skill System), hiệu ứng skill, cooldown, buff/debuff
- Bổ sung vật phẩm, rương, trang bị (equipment), hệ thống "rarity"
- Thêm chỉ số riêng cho từng vũ khí(sát thương , knockback,.vv)
- Thêm hệ thống Knockback riêng cho từng quái

6.2.2. Mở rộng bản đồ và tài nguyên

- Tăng số lượng map
- Tối ưu việc streaming map bằng Chunk System nâng cao
- Hỗ trợ map lớn (open-world nhỏ)

6.2.3. Tối ưu hóa hiệu năng

- Giảm tải CPU bằng hệ thống batching
- Tối ưu AI update theo distance
- Giảm số lần render các object off-screen

6.2.4. Hỗ trợ hệ thống đa nền tảng

- Xuất bản Linux / MacOS build
- Port game sang web: Java WebStart hoặc chuyển sang libGDX

6.2.5. Phát triển chế độ Multiplayer

- Multiplayer local (cùng máy)
- Multiplayer online, tạo phòng chơi
- Bảng xếp hạng (Leaderboard) theo:

- Level
- Sát thương gây ra
- Tổng thời gian sinh tồn

6.3 TÀI LIỆU THAM KHẢO

1. **Oracle Java Documentation**
   https://docs.oracle.com/javase/
2. **Youtube: RyiSnow – 2D Java Game Development Series**
3. **Tiled Map Editor Documentation**
   https://doc.mapeditor.org/
4. **StackOverflow – Java2D, game-loop, collision topics**
5. **Github** – Tham khảo kiến trúc và pattern từ các dự án game tương tự