TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM
TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG
KHOA CÔNG NGHỆ THÔNG TIN

NGUYỄN VĨNH HƯNG - 52200097
KHƯU TRÙNG DƯƠNG - 52200154
NGUYỄN HÒA AN - 52200182

CHUYẾN TÀU CUỐI CÙNG

BÁO CÁO CUỐI KỲ
PHÁT TRIỂN TRÒ CHƠI

THÀNH PHỐ HỒ CHÍ MINH, NĂM 2025

TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM
TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG
KHOA CÔNG NGHỆ THÔNG TIN

NGUYỄN VĨNH HƯNG - 52200097
KHƯU TRÙNG DƯƠNG - 52200154
NGUYỄN HÒA AN - 52200182

CHUYẾN TÀU CUỐI CÙNG

BÁO CÁO CUỐI KỲ
PHÁT TRIỂN TRÒ CHƠI

Người hướng dẫn
ThS. NCS. Vũ Đình Hồng

THÀNH PHỐ HỒ CHÍ MINH, NĂM 2025

LỜI CẢM ƠN

Chúng em xin chân thành cảm ơn thầy Vũ Đình Hồng người đã tận tình định hướng, cung cấp kiến thức chuyên môn và đưa ra những nhận xét quan trọng giúp nhóm hoàn thiện đồ án một cách tốt nhất trong suốt quá trình thực hiện, giúp nhóm hoàn thiện đồ án một cách tốt nhất.

TP. Hồ Chí Minh, ngày 12 tháng 11 năm 2025

Tác giả

(Ký tên và ghi rõ họ tên)

Hung

Nguyễn Vĩnh Hưng

Duong

Khưu Trùng Dương

An

Nguyễn Hòa An

CÔNG TRÌNH ĐƯỢC HOÀN THÀNH
TẠI TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG

Chúng tôi xin cam đoan đây là công trình nghiên cứu của riêng chúng tôi và được sự hướng dẫn khoa học của ThS. NCS. Vũ Đình Hồng. Các nội dung nghiên cứu, kết quả trong đề tài này là trung thực và chưa công bố dưới bất kỳ hình thức nào trước đây. Những số liệu trong các bảng biểu phục vụ cho việc phân tích, nhận xét, đánh giá được chính tác giả thu thập từ các nguồn khác nhau có ghi rõ trong phần tài liệu tham khảo.

Ngoài ra, trong Dự án còn sử dụng một số nhận xét, đánh giá cũng như số liệu của các tác giả khác, cơ quan tổ chức khác đều có trích dẫn và chú thích nguồn gốc.

Nếu phát hiện có bất kỳ sự gian lận nào chúng tôi xin hoàn toàn chịu trách nhiệm về nội dung Dự án của mình. Trường Đại học Tôn Đức Thắng không liên quan đến những vi phạm tác quyền, bản quyền do tôi gây ra trong quá trình thực hiện (nếu có).

TP. Hồ Chí Minh, ngày 12 tháng 11 năm 2025.

Tác giả

(Ký tên và ghi rõ họ tên)

Hung

Nguyễn Vĩnh Hưng

Duong

Khưu Trùng Dương

An

Nguyễn Hòa An

CHUYẾN TÀU CUỐI CÙNG

TÓM TẮT

Chuyến Tàu Cuối Cùng là một trò chơi 2D theo phong cách pixel narrative–puzzle kết hợp symbolic combat, lấy bối cảnh trong một chuyến tàu siêu thực – nơi nhân vật chính phải đối diện với các tầng ký ức, tội lỗi và sự thật bị chôn giấu. Trò chơi được xây dựng bằng Unity theo mô hình GameObject–Component, kết hợp hệ thống hội thoại phân nhánh, hệ thống puzzle (Sokoban & Jigsaw), AI địch theo FSM, hệ thống camera bằng Cinemachine và cơ chế Save/Load bằng JSON.

Dự án tập trung vào việc thiết kế một trải nghiệm mang tính nội tâm, nơi lựa chọn của người chơi ảnh hưởng trực tiếp tới chỉ số Truthfulness, từ đó mở ra nhiều hướng phát triển cốt truyện và ending khác nhau. Hệ thống được phân tích và thiết kế theo các mô hình UML (Class Diagram, Activity Diagram và Sequence Diagram), đảm bảo tính mạch lạc và dễ mở rộng.

Kết quả cho thấy trò chơi hoạt động ổn định, các module chính (di chuyển, hội thoại, puzzle, lưu tiến trình, AI…) được triển khai thành công và tuân theo kiến trúc phân lớp rõ ràng.

MỤC LỤC

DANH MỤC HÌNH VẼ

DANH MỤC BẢNG BIỂU

DANH MỤC CÁC CHỮ VIẾT TẮT

AI	Artificial Intelligence
HP	Health Points
NPC	Non-Player Character
MC	Main Character
SPX	Sound Effect

CHƯƠNG 1. MỞ ĐẦU VÀ TỔNG QUAN ĐỀ TÀI

1.1 Giới thiệu đề tài

"Chuyến Tàu Cuối Cùng (The Last Train)" là một trò chơi narrative-driven puzzle kết hợp symbolic combat lấy bối cảnh bên trong một chuyến tàu siêu thực – nơi thực tại, ký ức và tâm lý con người hòa lẫn vào nhau. Người chơi vào vai An, một người đàn ông đang trong trạng thái hấp hối sau một tai nạn nghiêm trọng. Trong cơn hấp hối ấy, não bộ anh tạo ra một hành trình cuối cùng: chuyến tàu xuyên qua các ga ký ức méo mó, nơi mỗi ga là một "phiên tòa nội tâm" buộc anh đối diện với những sai lầm đã chôn giấu suốt nhiều năm.

Game tập trung vào khai thác yếu tố tâm lý – đạo đức – bản ngã, thông qua các puzzle logic, chiến đấu mang tính biểu tượng và hệ thống lựa chọn ảnh hưởng trực tiếp đến chỉ số Truthfulness (mức độ thành thật của nhân vật với chính mình). Concept mang phong cách tối giản, ánh sáng lạnh, kết hợp âm thanh ambient buồn tạo nên trải nghiệm cảm xúc mạnh mẽ, vừa u tối vừa sâu sắc.

Trò chơi hướng đến trải nghiệm chiêm nghiệm hơn là hành động, giúp người chơi dần khám phá câu chuyện thật phía sau hành trình của An – một hành trình tự vấn, đối mặt với tội lỗi và quyết định xem liệu anh có thể vượt qua chính mình để tìm thấy sự cứu rỗi cuối cùng.

1.2 Lý do chọn đề tài

Trò chơi với các hình ảnh chuyến tàu, ga ký ức, người đồng hành ảo, boss là người tình cũ, đồng nghiệp bị hại… tất cả đều mang tính ẩn dụ mạnh mẽ. Điều này giúp game không chỉ là một hành trình giải đố mà là một biểu tượng hóa đời sống nội tâm – phù hợp với xu hướng nghệ thuật tối giản, trừu tượng đang được ưa chuộng. Đồng thời thai thác sâu vào mảng tối nội tâm con người, nơi cảm xúc tội lỗi, hối hận và sự thật bị bóp méo trở thành chất liệu chính để xây dựng gameplay và câu chuyện.

Người chơi không chỉ *đọc* câu chuyện mà còn *tương tác* trực tiếp với sự giằng xé trong tâm trí nhân vật. Hệ thống Truthfulness đóng vai trò phản ánh đạo đức của An qua từng lựa chọn, đem lại nhiều hướng phát triển cốt truyện và kết thúc đa dạng.

Việc thiết kế "Chuyến Tàu Cuối Cùng" với mong muốn tạo ra được sản phẩm có chiều sâu, mang tính nhân văn và để lại dư âm dài cho người chơi.

1.3 Mục tiêu thực hiện đề tài

1.3.1 Mục tiêu chung

Xây dựng một trò chơi pixel-2D narrative-puzzle với phong cách u ám, đậm chất nội tâm, trong đó người chơi khám phá hành trình tự vấn của nhân vật An, đối diện với những ký ức và tội lỗi được nhân hóa thành các NPC và Boss tượng trưng.

1.3.2 Mục tiêu cụ thể

- Mục tiêu về cốt truyện – thế giới:
    - Hoàn thiện câu chuyện xoay quanh 4 ga ký ức và 1 ga phán xét cuối.
    - Xây dựng 4 nhân vật quan trọng (Nam, Lan, Hoàng, Người Đồng Hành).
    - Thiết kế twist cuối: mọi thứ đều là dấu hiệu của cơn hấp hối.
    - Xây dựng nhiều ending (Good – Neutral – Bad – Meta).

CHƯƠNG 2. CƠ SỞ LÝ THUYẾT

2.1 Tổng quan về Unity Engine

2.1.1 Khái quát về Unity

Unity là một trong những game engine phổ biến nhất hiện nay nhờ tính dễ sử dụng, hỗ trợ đa nền tảng và cộng đồng phát triển lớn. Unity cung cấp giao diện trực quan kết hợp với ngôn ngữ lập trình C#, giúp người dùng có thể tạo ra game 2D hoặc 3D mà không cần xây dựng lại toàn bộ từ đầu. Nhờ thư viện API phong phú và công cụ mạnh mẽ, Unity phù hợp cả cho sinh viên, nhà phát triển độc lập lẫn các studio lớn.

2.1.2 Cấu trúc dự án Unity

Một dự án Unity được tổ chức theo những khu vực chính:

- **Assets:** nơi lưu trữ toàn bộ tài nguyên của game như hình ảnh, âm thanh, mô hình, script…
- **Scenes:** mỗi scene đại diện cho một màn chơi hoặc một khu vực trong game.
- **Prefabs:** mẫu đối tượng giúp tái sử dụng dễ dàng, rất hữu ích cho những đối tượng xuất hiện nhiều lần như NPC, item, UI…
- **Scripts:** chứa các mã C# để điều khiển logic và hành vi của game.

Cách tổ chức này giúp việc quản lý dự án rõ ràng, hạn chế trùng lặp và hỗ trợ làm việc nhóm hiệu quả hơn.

2.1.3 Mô hình GameObject – Component

Unity hoạt động dựa trên mô hình **GameObject–Component**, tức là mọi đối tượng trong game đều là một GameObject, và các hành vi của nó được tạo nên bởi các Component gắn vào.

Ví dụ các Component chính trong 2D:

- **Transform** xác định vị trí.
- **SpriteRenderer** hiển thị hình ảnh.

- **Collider** dùng cho va chạm.
- **Script** chứa logic tùy chỉnh.

Nhờ cấu trúc này, việc mở rộng chức năng chỉ đơn giản là gắn thêm component, giúp game linh hoạt và dễ phát triển.

2.2 Transform và không gian trong Unity

2.2.1 Transform

Transform lưu trữ Position, Rotation và Scale của đối tượng. Đây là thành phần bắt buộc mà mọi GameObject đều có. Trong game 2D, Position thường là yếu tố được quan tâm nhất vì quyết định vị trí nhân vật, vật thể và UI trong không gian.

- **Position:** vị trí của đối tượng.
- **Rotation:** hướng xoay.
- **Scale:** kích thước.

2.2.2 Local Space và World Space

- **World Space:** tọa độ tuyệt đối trong toàn cảnh game.
- **Local Space:** tọa độ tương đối so với đối tượng cha.

Hiểu rõ hai hệ tọa độ giúp tránh lỗi sai lệch vị trí, đặc biệt khi làm animation, move theo parent hoặc dùng camera.

2.2.3 Sorting Layer và Order in Layer

Trong game 2D, các đối tượng có thể bị che nhau. Unity sử dụng Sorting Layer và Order in Layer để quyết định đối tượng nào hiển thị phía trước, đối tượng nào nằm phía sau. Đây là yếu tố quan trọng khi thiết kế cảnh và nhân vật.

2.2.3.1 Sorting Layer

Sorting Layer là cấp độ phân lớp cao nhất dùng để nhóm các sprite theo thứ tự hiển thị. Mỗi sprite renderer thuộc về một Sorting Layer cụ thể (ví dụ: “Background”, “Default”, “Foreground”). Khi Unity quyết định đối tượng nào vẽ trước, hệ thống sẽ so sánh Sorting Layer trước, sau đó mới đến các yếu tố khác.

Các Layer có thứ tự từ dưới lên: Layer ở vị trí cao hơn trong danh sách Sorting Layer sẽ được vẽ sau, tức là nằm trên các layer bên dưới.

Sorting Layer cho phép tách rõ các lớp cảnh, tránh việc phải điều chỉnh thủ công từng đối tượng.

Sorting Layer thường được thiết kế theo nguyên tắc phân tầng:

Background → Middleground → Characters → Foreground → UI

2.2.3.2 Order in Layer

Sau khi Unity xác định Sorting Layer, hệ thống tiếp tục so sánh **Order in Layer**, đây là mức độ phân lớp nhỏ hơn bên trong mỗi Sorting Layer.

- Giá trị Order in Layer càng **cao** thì sprite được vẽ **sau**, tức là hiển thị **phía trước**.
- Đây là cách để điều chỉnh chi tiết thứ tự hiển thị giữa các đối tượng cùng thuộc một Sorting Layer.
- Ví dụ: nhân vật (order = 3) đứng sau một bụi cây (order = 5) dù cả hai cùng thuộc Sorting Layer "Characters".

Cơ chế hoạt động tổng quát:

Unity thực hiện render theo thứ tự sau:

1. Sắp xếp theo **Sorting Layer** → từ layer thấp đến layer cao.
2. Đối với các sprite trong cùng một layer: sắp xếp theo **Order in Layer**.
3. Nếu Order in Layer giống nhau: so sánh **material render queue**.
4. Cuối cùng: dùng **z-index** hoặc **distance to camera** (nếu chế độ Transparency Sort Mode cho phép).

Nhờ cơ chế này, Unity đảm bảo trình tự hiển thị nhất quán, giúp nhà phát triển kiểm soát chính xác đối tượng nào xuất hiện trước – sau trong khung hình.

2.3 Hệ thống vật lý 2D

2.3.1 Rigidbody2D

Rigidbody2D cho phép đối tượng chịu ảnh hưởng của trọng lực, lực đẩy, vận tốc hoặc va chạm. Tùy mục đích, đối tượng có thể chọn dạng:

- **Dynamic:** chịu đầy đủ vật lý.
- **Kinematic:** di chuyển thủ công nhưng vẫn tương tác với collider khác.
- **Static:** không di chuyển, thường dùng cho tường hoặc chướng ngại vật.

2.3.2 Collider2D

Collider tạo vùng va chạm cho đối tượng. Unity hỗ trợ nhiều dạng collider khác nhau như:

- **BoxCollider2D**
- **CircleCollider2D**
- **PolygonCollider2D**

Ngoài ra, thuộc tính **IsTrigger** cho phép phát hiện đối tượng đi vào mà không tạo va chạm vật lý.

2.3.3 Sự kiện va chạm

Unity cung cấp các hàm callback:

- **OnTriggerEnter2D()**
- **OnTriggerExit2D()**
- **OnCollisionEnter2D()**

Các sự kiện này phục vụ tương tác như mở cửa, hiển thị UI, trigger hội thoại, chuyển cảnh,…

2.4 Animation và Animator Controller

2.4.1 Animation Clip

Animation Clip là chuỗi chuyển động được định nghĩa sẵn. Ví dụ: animation đứng yên, đi bộ hoặc tương tác với vật thể.

2.4.2 Animator Controller

Animator Controller đóng vai trò như "bộ não" của hệ thống hoạt ảnh, quyết định clip nào sẽ phát và khi nào chuyển clip khác.

2.4.3 State Machine

Unity cho phép tạo một sơ đồ trạng thái để mô tả hành vi của nhân vật.

Ví dụ: Idle → Walk → Run → Idle.

Sự chuyển trạng thái dựa trên tham số boolean, trigger hoặc float.

2.4.4 Animation Event

Animation Event giúp gọi hàm C# vào thời điểm cụ thể, thường dùng cho hiệu ứng âm thanh hoặc hành động đồng bộ với chuyển động.

2.5 Hệ thống Input

Unity hỗ trợ:

- **Input Manager cũ (Legacy):** đơn giản, thường dùng trong game 2D nhỏ.
- **New Input System:** linh hoạt hơn, phù hợp đa nền tảng và chơi bằng controller.

Cả hai hệ thống đều cho phép bắt sự kiện bàn phím, chuột và cảm ứng để điều khiển nhân vật hoặc tương tác UI.

2.6 Hệ thống UI

2.6.1 Canvas

Canvas là nơi chứa toàn bộ phần giao diện của game. Unity hỗ trợ nhiều chế độ hiển thị khác nhau cho phù hợp từng bối cảnh.

2.6.2 TextMeshPro

Đây là công cụ hiển thị văn bản sắc nét, chuyên nghiệp hơn so với UI Text mặc định.

2.6.3 Các thành phần UI phổ biến

Bao gồm: Image, Button, Slider, Panel…

Chúng được sử dụng để xây menu, hộp thoại, thông báo hoặc hệ thống lựa chọn.

2.6.4 Raycast và thứ tự hiển thị

Unity dùng Raycast để xác định UI nào được nhấn, và Canvas Sorting Order để xác định UI nào chồng lên UI nào.

2.7 Camera và Cinemachine

Cinemachine là plug-in hỗ trợ điều khiển camera tự động. Với các game 2D, Cinemachine giúp camera bám theo nhân vật mượt mà mà không cần viết code phức tạp.

2.7.1 Virtual Camera

Mỗi virtual camera có thể có thiết lập riêng: tốc độ bám, vùng confiner, hướng nhìn,…

2.7.2 Chuyển camera

Cinemachine Camera được chuyển qua lại dựa trên Priority hoặc thông qua Timeline, rất tiện cho việc làm cutscene.

2.8 Scene Management

Unity sử dụng **SceneManager** để tải, đổi hoặc gộp scene.

Thông thường, SceneManager.LoadScene() được dùng để đi từ màn chơi này sang màn chơi khác.

Việc quản lý cảnh đúng cách giúp game mượt hơn và tránh các lỗi như mất dữ liệu hoặc tải không đầy đủ tài nguyên.

2.9 Lập trình C# trong Unity

Lập trình C# là phần cốt lõi trong phát triển trò chơi bằng Unity. Mặc dù Unity cung cấp giao diện kéo–thả trực quan, nhưng phần lớn hành vi của đối tượng, logic

điều khiển và tương tác trong game đều được triển khai bằng script C#. Hiểu rõ vòng đời của MonoBehaviour, cách giao tiếp giữa các script và cơ chế xử lý bất đồng bộ sẽ giúp quá trình xây dựng game trở nên mạch lạc và ổn định hơn.

2.9.1 Vòng đời của MonoBehaviour

Trong Unity, mọi script được gán vào GameObject đều kế thừa từ lớp MonoBehaviour. Unity không thực thi mã nguồn một cách tuyến tính; thay vào đó, mỗi script trải qua một chuỗi các phương thức vòng đời được hệ thống tự động gọi theo thời điểm thích hợp. Chu trình này bao gồm các giai đoạn khởi tạo, thiết lập dữ liệu, thực thi theo từng khung hình, xử lý vật lý và kết thúc vòng đời của đối tượng.

Ở mức khái quát, quá trình này có thể được mô tả như sau:

- **Awake()** được gọi ngay khi đối tượng được nạp vào bộ nhớ, thường dùng để khởi tạo dữ liệu cơ bản.
- **Start()** được gọi khi đối tượng bắt đầu hoạt động trong scene, đảm bảo rằng các thành phần liên quan đã hoàn tất khởi tạo.
- **Update()** được thực thi mỗi khung hình và là nơi diễn ra các hành vi có tính chất liên tục theo thời gian.
- **FixedUpdate()** hoạt động theo chu kỳ thời gian cố định, thường được sử dụng trong các tác vụ vật lý để đảm bảo tính ổn định.
- **LateUpdate()** chạy sau Update và thường được dùng cho các tác vụ cần bảo đảm đối tượng đã hoàn tất di chuyển trước khi xử lý hậu kỳ, chẳng hạn điều chỉnh camera hoặc hiệu ứng thị giác.

Các phương thức như **OnEnable(), OnDisable()** và **OnDestroy()** phản ánh trạng thái vòng đời của đối tượng khi được kích hoạt, vô hiệu hóa hoặc bị hủy.

Việc nắm rõ đặc điểm của từng giai đoạn giúp tránh các vấn đề thường gặp như xung đột dữ liệu, lỗi cạnh tranh thời điểm (race condition) hoặc các hiện tượng không đồng nhất trong chuyển động và xử lý vật lý.

2.9.2 Giao tiếp giữa các script

Trong game có nhiều đối tượng cần trao đổi dữ liệu — ví dụ Player báo cho GameManager khi hoàn thành nhiệm vụ, hoặc NPC nhận tín hiệu mở hội thoại. Unity hỗ trợ nhiều cách giao tiếp

2.9.2.1 Tham chiếu trực tiếp (drag & drop)

Cách đơn giản nhất. Người làm game kéo đối tượng vào ô public trong Inspector.

Ưu điểm: dễ dùng, rõ ràng.

Nhược điểm: phụ thuộc scene, khó bảo trì với số lượng lớn.

2.9.2.2 Tìm kiếm bằng API

Một số hàm phổ biến:

- **FindObjectOfType<T>()** → tìm một object kiểu T trong scene.
- **GameObject.Find("Tên")** → tìm object theo tên.
- **transform.Find("ChildPath")** → tìm trong hierarchy con.

Ưu điểm: tiện trong một số trường hợp.

Nhược điểm: tốn hiệu năng nếu gọi thường xuyên.

2.9.2.3 Singleton Pattern

Dùng cho các hệ thống duy nhất:

- **GameManager**
- **AudioManager**
- **DialogueManager**

Ví dụ:

```
public class GameManager : MonoBehaviour
{
    public static GameManager Instance;
    private void Awake()
    {
        Instance = this;
    }
}
```

Sau đó có thể gọi:

```
GameManager.Instance.DoSomething();
```

Ưu điểm: dễ truy cập, gọn.

Nhược điểm: dùng sai dễ gây phụ thuộc lẫn nhau.

2.9.2.4 Event và Delegate

Cách giao tiếp hiện đại hơn, linh hoạt và ít phụ thuộc.

Ví dụ: Player chết → các UI khác tự cập nhật.

```
public static event Action OnPlayerDead;
```

Hệ thống sẽ tự "gửi tín hiệu" cho các script đã đăng ký.

2.9.2.5 ScriptableObject

Ngoài việc lưu dữ liệu, ScriptableObject có thể đóng vai trò như “kênh truyền tin” giữa các scene mà không cần giữ object trong hierarchy.

2.9.3 Coroutine – Xử lý bất đồng bộ trong Unity

Coroutine giúp chạy các tác vụ theo thời gian, thay vì chạy ngay lập tức. Đây là công cụ mạnh mẽ để tạo hiệu ứng, chuyển cảnh, hoặc các thao tác cần delay.

Ví dụ: làm fade-out trong 1 giây:

```
IEnumerator FadeOut()
{
    for (float t = 0; t < 1f; t +=
Time.deltaTime)
    {
        canvasGroup.alpha = 1 - t;
        yield return null;
    }
}
```

Coroutine phù hợp cho:

- Hiệu ứng Fade In/Out
- Mở cửa từ từ
- Delay trước khi đổi scene
- Cutscene theo timeline đơn giản
- Chờ vài giây rồi chạy hành động

Lợi ích:

- Không chặn luồng chính của game
- Rõ ràng và dễ đọc
- Dễ đồng bộ với animation

2.9.4 Xử lý Input trong script C#

Trong Unity, xử lý input là yếu tố then chốt quyết định cách người chơi tương tác với nhân vật và môi trường. Hệ thống Input Manager truyền thống cung cấp cách truy vấn trực tiếp trạng thái bàn phím và chuột theo từng khung hình. Chẳng hạn, việc lấy hướng di chuyển trong game 2D thường được biểu diễn qua hai trục:

```
float moveX = Input.GetAxisRaw("Horizontal");
float moveY = Input.GetAxisRaw("Vertical");
```

Các giá trị này hình thành vector chuyển động, phản ánh phản hồi tức thời từ hành vi của người chơi.

Hệ thống Input System mới hoạt động theo mô hình callback, trong đó hành động của người chơi được truyền thành sự kiện tới script:

```
public void OnMove(InputAction.CallbackContext
context)
{
    moveDir = context.ReadValue<Vector2>();
}
```

Cách tiếp cận này mang tính mô-đun hơn, dễ mở rộng và phù hợp với game hỗ trợ nhiều thiết bị nhập liệu.

2.9.5 Quản lý trạng thái trong script

Việc quản lý trạng thái là cần thiết để điều chỉnh hành vi của NPC hoặc người chơi theo từng hoàn cảnh. Mô hình FSM (Finite State Machine) thường được sử dụng nhờ cấu trúc rõ ràng: mỗi trạng thái đại diện cho một hành vi cụ thể và chuyển đổi giữa các trạng thái diễn ra theo điều kiện đã xác định.

Ví dụ, một nhân vật có thể được mô tả bằng các trạng thái:

```
enum State { Idle, Walking, Talking }
State currentState;
```

FSM giúp logic trở nên minh bạch, tránh xung đột hành vi và hỗ trợ việc mở rộng trong tương lai.

2.9.6 Save và Load Data trong Unity

Trong các trò chơi có tiến trình, lựa chọn hoặc phân đoạn theo chapter, cơ chế lưu và tải dữ liệu giữ vai trò quan trọng nhằm bảo toàn trạng thái của người chơi. Unity cung cấp nhiều phương thức lưu trữ với đặc tính khác nhau, phù hợp cho từng loại dữ liệu và mục đích sử dụng. Ba cơ chế phổ biến nhất bao gồm PlayerPrefs, Serialization và File I/O.

2.9.7 PlayerPrefs

PlayerPrefs là cơ chế lưu trữ dạng khóa–giá trị (key–value) do Unity cung cấp, thích hợp cho dữ liệu dung lượng nhỏ và mang tính checkpoint. Phương thức này thường được sử dụng để:

- Ghi nhận chương hiện tại hoặc mốc tiến trình.
- Đánh dấu các puzzle đã hoàn thành.
- Lưu cài đặt đơn giản như âm lượng hoặc chế độ hiển thị,…

Ưu điểm của PlayerPrefs nằm ở tính đơn giản, khả năng truy xuất nhanh và không yêu cầu cấu trúc dữ liệu phức tạp. Tuy nhiên, PlayerPrefs không phù hợp cho các thông tin lớn hoặc dữ liệu mang tính phân cấp.

2.10 Serialization (JSON)

Serialization cho phép chuyển đổi dữ liệu thành chuỗi văn bản có cấu trúc để lưu trữ hoặc truyền tải. JSON là định dạng được sử dụng rộng rãi nhờ khả năng đọc hiểu cao và dễ tích hợp.

Trong trò chơi, JSON đặc biệt thích hợp cho:

- Lưu trạng thái Truthfulness theo thời điểm,
- Lưu danh sách các chọn lựa quan trọng của người chơi,
- Lưu tiến độ và cấu trúc puzzle (ví dụ: mảnh đã thu thập, vị trí ban đầu, trạng thái hoàn thành),

- Lưu dữ liệu xuyên chapter nếu trò chơi có tính tuyến tính hoặc nhiều nhánh.

Phương pháp này đem lại khả năng mở rộng mạnh hơn PlayerPrefs, đồng thời phù hợp với những trò chơi đòi hỏi khôi phục trạng thái phức tạp.

2.10.1 File I/O

File I/O cho phép đọc và ghi trực tiếp vào hệ thống tập tin. Đây là phương pháp linh hoạt nhất, phù hợp cho dữ liệu cần mức độ kiểm soát cao hoặc mang tính lưu trữ dài hạn.

Các trường hợp sử dụng phổ biến gồm:

- Lưu "nhật ký lựa chọn" của người chơi.
- Ghi các bản ghi phục vụ kiểm thử hoặc tái tạo lỗi.
- Lưu cấu hình nội bộ dành cho nhà phát triển.

Do có đặc tính tự do cao, File I/O đòi hỏi lập trình viên quản lý đường dẫn, định dạng dữ liệu và cơ chế sao lưu cẩn thận để tránh lỗi ghi chồng hoặc hỏng file.

2.10.2 Tổ chức mã nguồn (Code Architecture)

Một dự án Unity có tính ổn định cao thường dựa trên cách tổ chức mã nguồn hợp lý. Ba mô hình phổ biến gồm:

- **Manager Pattern:** tập trung các chức năng điều phối ở những lớp trung tâm như GameManager, UIManager, AudioManager.
- **Component-based Architecture:** chia nhỏ chức năng thành nhiều script độc lập, giúp mã nguồn linh hoạt và dễ tái sử dụng.
- **Event-driven Architecture:** giảm sự phụ thuộc trực tiếp giữa các thành phần bằng cách sử dụng sự kiện; phù hợp cho các hệ thống phản ứng như UI hoặc các hệ thống gameplay theo trạng thái.

Việc lựa chọn kiến trúc phù hợp giúp dự án dễ bảo trì, dễ mở rộng và hạn chế phát sinh lỗi trong quá trình phát triển.

CHƯƠNG 3. PHÂN TÍCH VÀ THIẾT KẾ HỆ THỐNG

3.1 Các Actor của hệ thống

Bảng 3.1 Các Actor của hệ thống

STT	Tác nhân	Mô tả vai trò
1	**Người chơi**	Điều khiển nhân vật An, tương tác với môi trường, giải puzzle, đánh quái lựa chọn hội thoại và trải nghiệm cốt truyện.
2	**Hệ thống**	Quản lý input, hội thoại, sự kiện, camera, puzzle, lưu tiến trình, cập nhật Truthfulness và điều phối flow của chapter.

3.2 Phân tích yêu cầu hệ thống

3.2.1 Yêu cầu chức năng

Hệ thống trò chơi cần đáp ứng các chức năng chính sau:

- **Hệ thống điều khiển nhân vật:** cho phép người chơi di chuyển, tương tác với đối tượng và kích hoạt các sự kiện trong môi trường.
- **Hệ thống hội thoại (Dialogue System):** trình bày nội dung hội thoại, hỗ trợ phân nhánh và phản hồi theo lựa chọn.
- **Hệ thống puzzle:** gồm các cơ chế vận hành, kiểm tra điều kiện hoàn thành và phản hồi cho người chơi.
- **Hệ thống lưu tiến trình:** ghi nhận trạng thái chapter, tiến độ puzzle, chỉ số Truthfulness hoặc các lựa chọn quan trọng.
- **Hệ thống camera:** đảm bảo theo dõi nhân vật và chuyển cảnh mượt mà.
- **Hệ thống UI:** hiển thị thông tin, menu, thanh trạng thái và hướng dẫn.

3.2.2 Yêu cầu phi chức năng

- **Tính ổn định:** hệ thống cần vận hành nhất quán, không gây lỗi trong quá trình chuyển scene hoặc tương tác.

- **Hiệu năng:** đảm bảo tốc độ khung hình ổn định, đặc biệt với môi trường 2D.
- **Khả năng mở rộng:** hỗ trợ thêm chapter, puzzle hoặc NPC mới mà không ảnh hưởng đến cấu trúc hiện có.
- **Trải nghiệm người dùng:** hành vi của UI, camera và hội thoại phải mượt mà, trực quan.

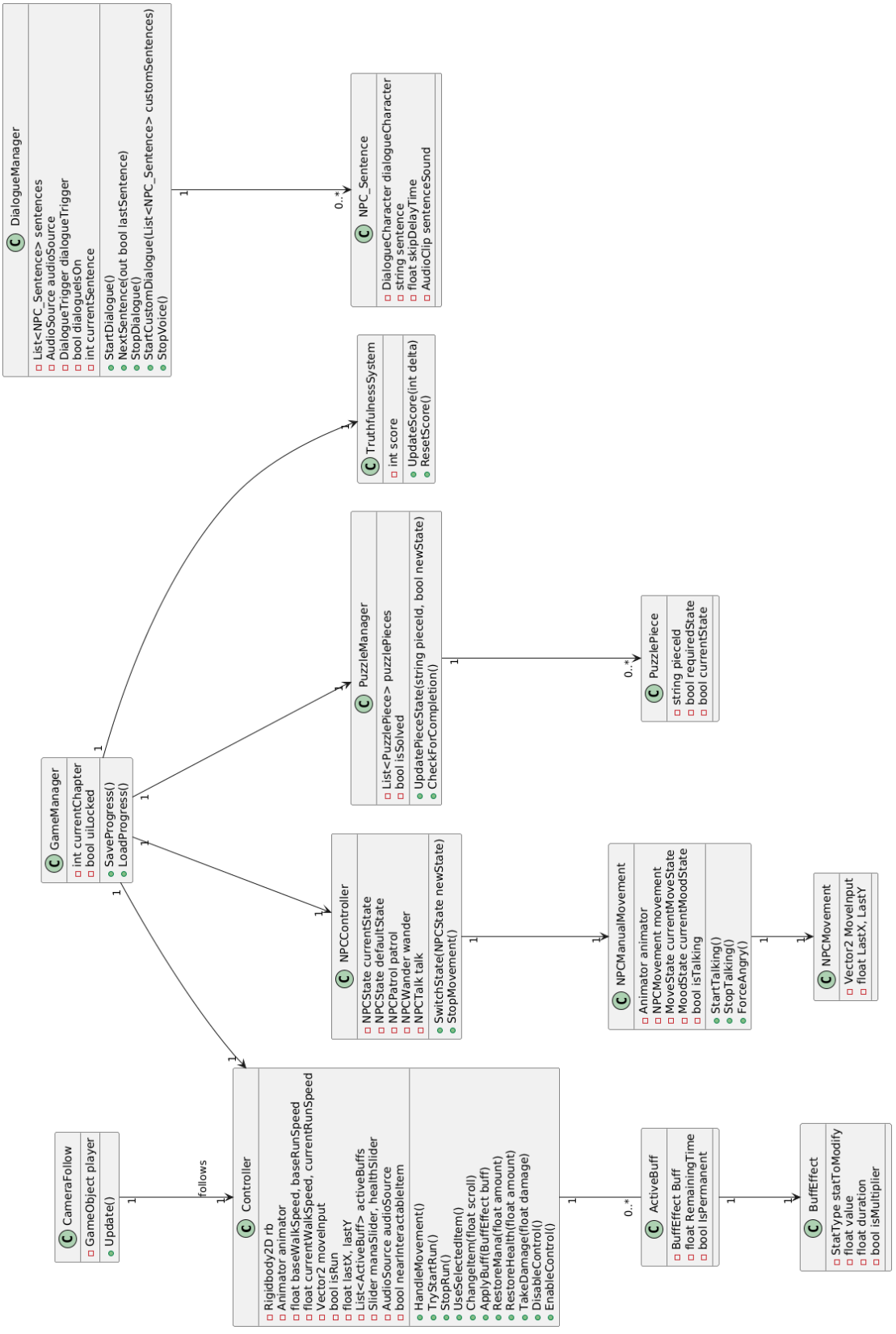
3.3 Chức năng của hệ thống

Bảng 3.2 Các chức năng của hệ thống

STT	Tên chức năng	Mô tả
1	**Di chuyển nhân vật**	Xử lý input để nhân vật An di chuyển trong môi trường 2D top-down.
2	**Tương tác đối tượng**	Tương tác với NPC, bảng thông báo, trigger cutscene, puzzle entrance.
3	**Hệ thống hội thoại**	Hiển thị hội thoại, phân nhánh theo lựa chọn, cập nhật chỉ số Truthfulness.
4	**Puzzle System**	Quản lý puzzle Sokoban và Jigsaw: vào puzzle, kiểm tra điều kiện, trả về kết quả.
5	**Cutscene và camera**	Kích hoạt cutscene, animation timeline, chuyển camera, khóa/giải khóa điều khiển.
6	**Hệ thống Save/Load**	Lưu checkpoint chapter, trạng thái puzzle, Truthfulness và các lựa chọn quan trọng.

3.4 Sơ đồ UML của hệ thống

3.4.1 Sơ đồ lớp (Class Diagram)



Hình 3.1 Sơ đồ lớp tổng thể

PlayerController (Controller)

- Xử lý input, di chuyển, Animator.
- Quản lý input, di chuyển (moveInput), chạy (isRun), thanh máu (healthSlider) và mana (manaSlider).
- Tương tác với InventoryManager để sử dụng item và thay đổi item.

NPCController

- Hiển thị biểu tượng tương tác, trigger hội thoại.

DialogueManager

- Quản lý hiển thị hội thoại, phân nhánh, sự kiện sau hội thoại.

TruthfulnessSystem

- Cập nhật điểm thật lòng theo lựa chọn người chơi.

PuzzleManager

- Quản lý logic Jigsaw và Sokoban.

GameManager

- Giữ tiến trình chapter, sync dữ liệu, lock/unlock UI.

CameraController

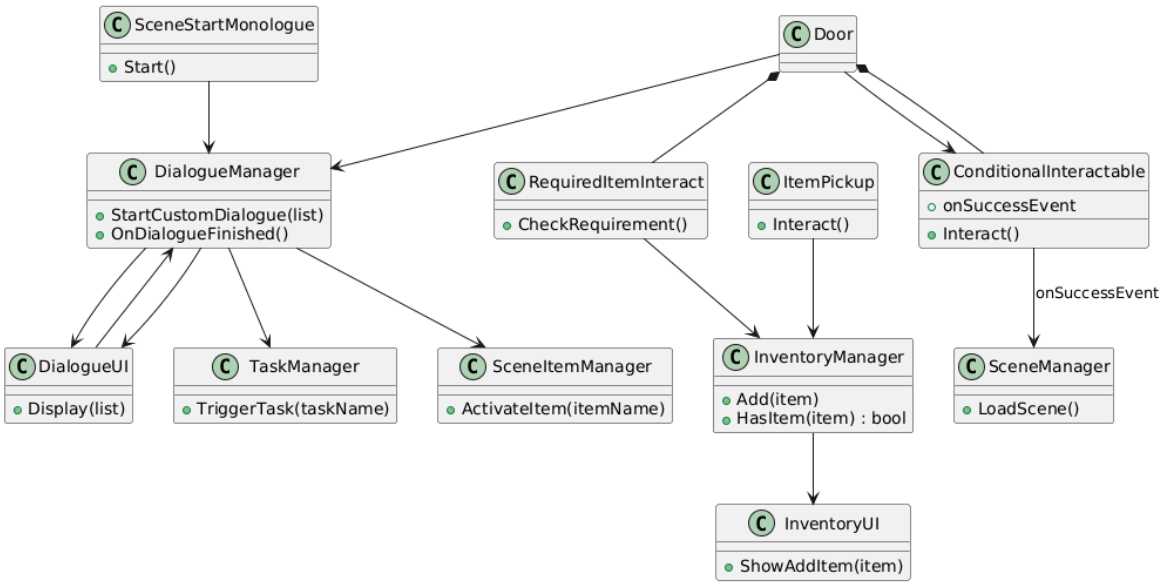
- Theo dõi nhân vật, điều khiển camera khi cutscene.

SaveSystem

- Lưu JSON: puzzle, truthfulness, checkpoint.

Phân tích từng module:

3.4.1.1 Sơ đồ lớp trong chapter khám phá bản đồ



Hình 3.2 Sơ đồ lớp trong chapter khám phá bản đồ

Sơ đồ mô tả cách các thành phần trong **Opening Map** tương tác với nhau khi người chơi bắt đầu game, nhận thoại, nhặt item, mở cửa, hoặc chuyển scene. Đây là chuỗi khởi động câu chuyện của map mở đầu, nơi các sự kiện như mở vật phẩm, kích hoạt nhiệm vụ, spawn vật thể được kích hoạt thông qua lời thoại.

Hệ thống bao gồm 3 nhóm chức năng chính:

- Nhóm hệ thống hội thoại (Monologue & Dialogue System)

  Các lớp chính:

  - SceneStartMonologue
  - DialogueManager
  - DialogueUI
  - TaskManager
  - SceneItemManager
  - Luồng hoạt động:

  1. **SceneStartMonologue.Start()**
     - Khi scene mở ra, lớp này khởi động monologue hoặc cuộc hội thoại ban đầu.
     - Nó **gọi đến DialogueManager**.

2. **DialogueManager.StartCustomDialogue(list)**
    - Nhận một list lời thoại.
    - Đẩy dữ liệu sang **DialogueUI** để hiển thị.
3. **DialogueUI.Display(list)**
    - UI đảm nhiệm việc hiển thị từng câu thoại, ảnh nhân vật, biểu tượng…
4. Sau khi UI báo hoàn thành một câu → **DialogueUI** → **DialogueManager**
5. Khi toàn bộ cuộc thoại kết thúc → **DialogueManager.OnDialogueFinished()**
    - Có thể kích hoạt:
        - **TaskManager.TriggerTask(taskName)**
        - **SceneItemManager.ActivateItem(itemName)**

- Nhóm hệ thống Inventory & Item Pickup

  Các lớp chính:
    - ItemPickup
    - InventoryManager
    - InventoryUI

  Luồng hoạt động:

1. **ItemPickup.Interact()**
    - Khi người chơi nhấn E vào vật phẩm.
2. **ItemPickup → InventoryManager.Add(item)**
    - Thêm item vào kho.
3. **InventoryManager → InventoryUI.ShowAddItem(item)**
    - Hiển thị popup + icon mới thêm vào.
4. **InventoryManager.HasItem(item)**
    - Được các hệ thống khác (như cửa hoặc tương tác có điều kiện) dùng để kiểm tra yêu cầu.

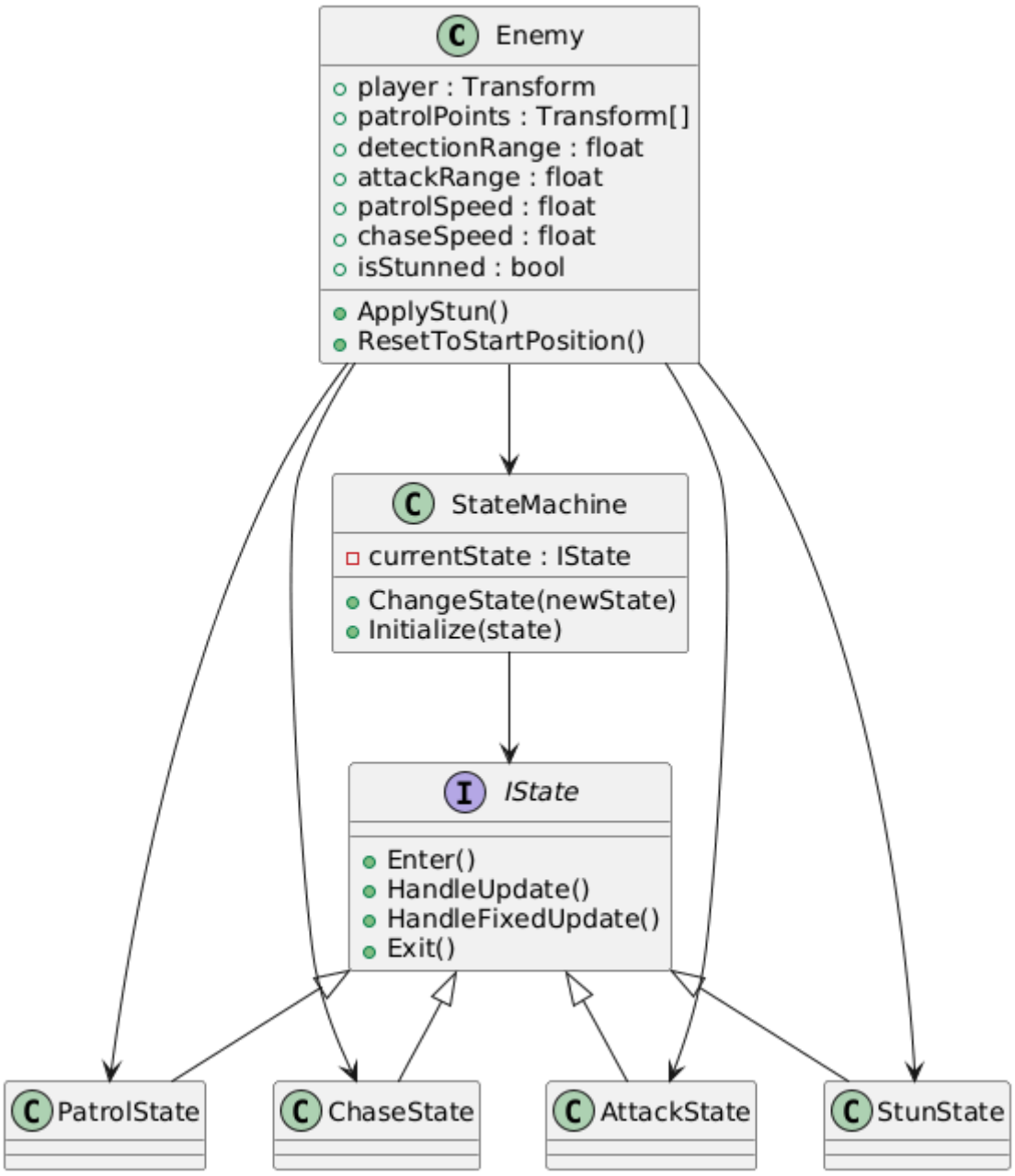
Đây là hệ thống nhặt đồ – lưu trữ – hiển thị. Nhiều hệ thống phụ thuộc vào kho đồ để quyết định xem Player có đủ điều kiện tiếp tục hay không.\

- Nhóm hệ thống tương tác object, mở cửa, chuyển scene:
  Các lớp chính:
  - Door
  - RequiredItemInteract
  - ConditionalInteractable
  - SceneManager (Unity SceneManager, hoặc class bạn wrap lại)

  Cấu trúc & quan hệ:
  - *Door – RequiredItemInteract
    - Một cửa có chứa một logic kiểm tra item.
    - RequiredItemInteract sẽ kiểm tra:
      - Player có item cần thiết?
      - Ví dụ: *"Cần chìa khóa gì"*
  - *Door -- ConditionalInteractable
    - Đây là class dùng để bật event khi thỏa điều kiện.
    - Cửa có thể:
    - Mở ngay (nếu không yêu cầu item),
    - Hoặc chỉ mở sau khi RequiredItemInteract.CheckRequirement() = true.
  - RequiredItemInteract → InventoryManager
    - Logic kiểm tra item gọi InventoryManager.HasItem().
  - Door → DialogueManager & DialogueUI
    - Nếu người chơi không có item yêu cầu:
      - Door có thể gọi DialogueManager → DialogueUI để hiển thị thoại:
        - "Cánh cửa khóa… có vẻ cần một cái chìa gì đó."
  - ConditionalInteractable → SceneManager.LoadScene()

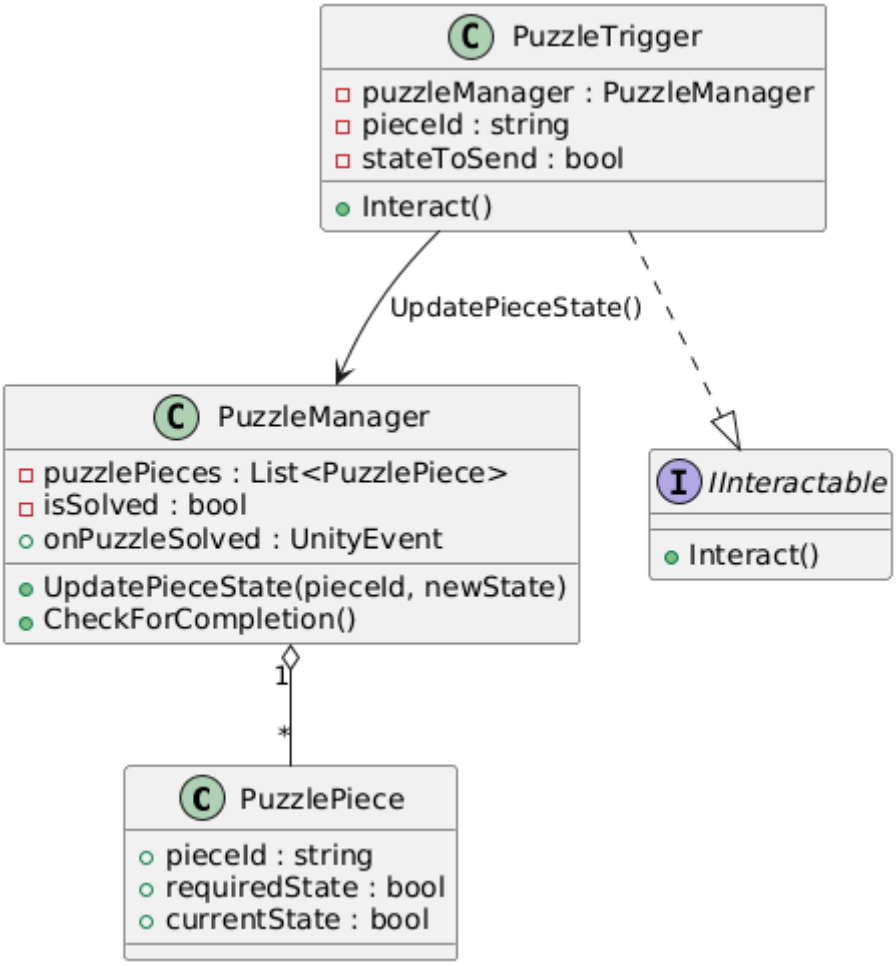
- Khi điều kiện thỏa mãn → chạy event thành công" (onSuccessEvent) → LoadScene
- Dùng để chuyển người chơi sang map kế tiếp.

3.4.1.2 Sơ đồ lớp hệ thống Enemy FSM



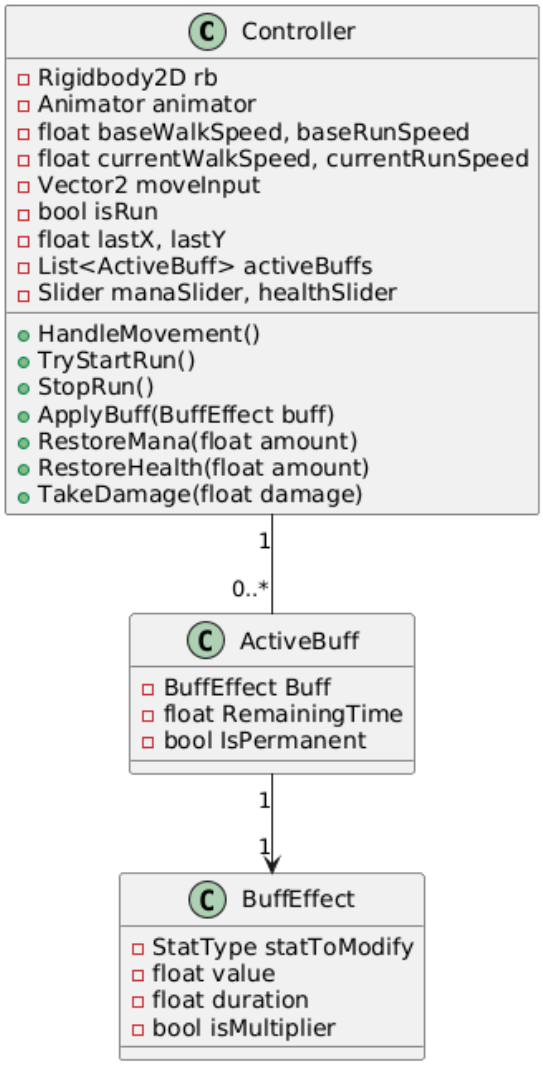
Hình 3.3 Sơ đồ lớp hệ thống Enemy FSM

3.4.1.3 Sơ đồ lớp Puzzle



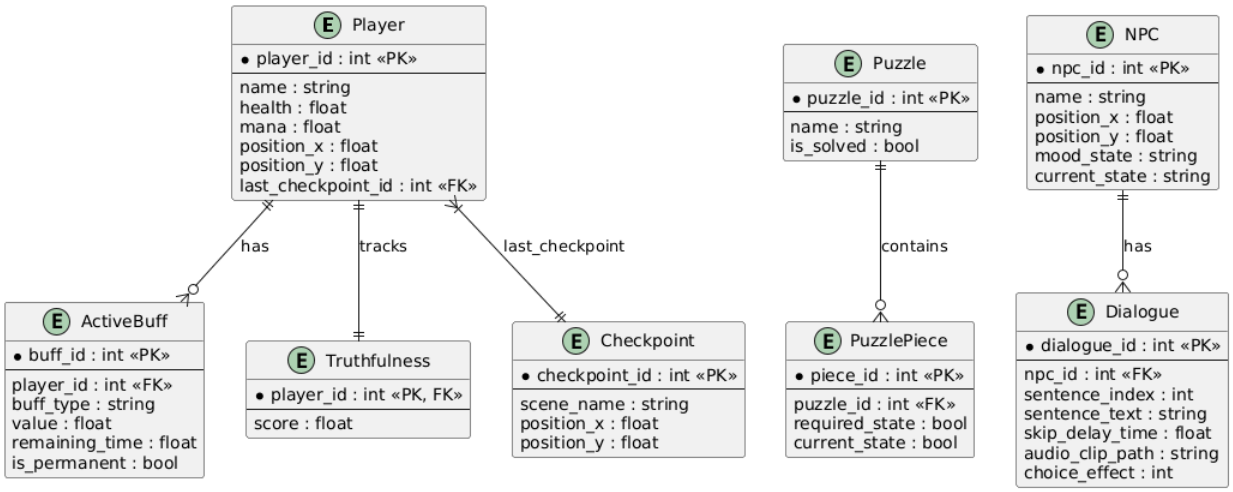
Hình 3.4 Sơ đồ lớp Puzzle

3.4.1.4 Sơ đồ lớp module Player và Buff System



Hình 3.5 Sơ đồ lớp module Player và Buff System

3.4.2 Sơ đồ ERD



Hình 3.6 Sơ đồ ERD

Sơ đồ ERD này được thiết kế để lưu trạng thái game của người chơi, câu đố, NPC, hội thoại, điểm thật thà (truthfulness), vật phẩm và checkpoint. Thiết kế này đảm bảo khả năng mở rộng, tối ưu truy vấn, và liên kết dữ liệu giữa các hệ thống gameplay nhờ vào việc:

1. **Tính nhất quán dữ liệu:**

- Mỗi bảng có khóa chính (PK) rõ ràng, các quan hệ sử dụng khóa ngoại (FK).
- Các trạng thái game (buff, puzzle, dialogue) được lưu trong bảng riêng, dễ cập nhật mà không làm thay đổi bảng Player.

2. **Khả năng mở rộng:**

- Có thể thêm buff mới, NPC mới, câu đố mới mà không phải thay đổi cấu trúc chính.
- choice_effect trong Dialogue cho phép mở rộng hệ thống lựa chọn và tác động tới điểm Truthfulness.

3. **Tối ưu hóa truy vấn:**

- Truy vấn buff hiện tại của player: join Player + ActiveBuff.
- Truy vấn câu đố hoàn thành: join Puzzle + PuzzlePiece.
- Truy vấn hội thoại NPC: join NPC + Dialogue.

4. **Tương thích với class diagram:**

- Player ↔ Controller + ActiveBuff + BuffEffect
- Puzzle ↔ PuzzleManager + PuzzlePiece
- NPC + Dialogue ↔ NPCController + DialogueManager
- Truthfulness ↔ TruthfulnessSystem
- Checkpoint ↔ SaveSystem

Thiết kế CSDL game này tuân theo các nguyên tắc:

1. Chuẩn hóa (Normalization):
   - 1NF (First Normal Form): Mỗi cột chứa giá trị nguyên tử, không lặp lại danh sách trong một ô. Ví dụ, `ActiveBuff` lưu mỗi buff trên một dòng, không lưu danh sách buff trong cột `buffs`.
   - 2NF (Second Normal Form): Mỗi bảng có PK riêng và các thuộc tính phụ thuộc hoàn toàn vào PK. Ví dụ, `PuzzlePiece` phụ thuộc vào `piece_id`, `puzzle_id` là FK.
   - 3NF (Third Normal Form): Không tồn tại phụ thuộc bắc cầu; tất cả thuộc tính không khóa phụ thuộc trực tiếp vào PK.
2. Tách biệt dữ liệu tạm thời và dữ liệu cố định:
   - Buff, PuzzlePiece, Dialogue được tách ra thành bảng riêng để dễ cập nhật, tránh lưu tất cả trong Player hay NPC.
   - `Truthfulness` tách ra vì là thông tin tính cách/player state và có thể mở rộng thành các điểm khác trong tương lai.
3. Khả năng mở rộng và bảo trì:
   - Có thể thêm buff mới, NPC mới, câu đố mới, hay item mà không thay đổi bảng Player.
   - Quan hệ 1-n cho phép thêm nhiều buff, nhiều câu thoại, nhiều mảnh ghép cho câu đố mà vẫn giữ được cấu trúc.

3.4.2.1 Phân tích chi tiết từng bảng

a. Bảng Player

- Vai trò: Lưu trạng thái chính của người chơi.
- Thuộc tính: Health, Mana, vị trí, và last_checkpoint_id.
- Lý do thiết kế:
    - Tách Player và Checkpoint để cho phép nhiều checkpoint trong game nhưng mỗi người chơi chỉ liên kết với một checkpoint hiện tại.
    - Có FK last_checkpoint_id giúp dễ truy xuất vị trí lưu game.

b. Bảng ActiveBuff + BuffEffect (tương thích class diagram)

- Vai trò: Lưu các hiệu ứng tạm thời/vĩnh viễn của player.
- Thuộc tính: Buff type, giá trị, thời gian còn lại, is_permanent.
- Thiết kế logic:
    - Buffs được lưu riêng để xử lý thời gian thực (update mỗi frame hoặc deltaTime).
    - FK player_id để liên kết trực tiếp với Player, cho phép truy vấn tất cả buff của player.

c. Bảng Puzzle + PuzzlePiece

- **Vai trò:** Lưu cấu trúc câu đố.
- Thiết kế logic:
    - Tách câu đố và các mảnh ghép để:
        - Kiểm tra từng mảnh ghép đúng hay sai (current_state == required_state).
        - Hỗ trợ event PuzzleSolved khi tất cả mảnh ghép đúng.
    - Cho phép nhiều loại câu đố khác nhau, dễ mở rộng mà không ảnh hưởng tới Player.

d. Bảng NPC + Dialogue

- Vai trò: Lưu trạng thái NPC và hội thoại.
- Thiết kế logic:
    - Dialogue được tách riêng giúp:
    - Thay đổi nội dung câu thoại mà không thay đổi NPC.

- Lưu các audio clip, skip delay, và choice effect ảnh hưởng tới Truthfulness.

- Quan hệ 1-n: Một NPC có thể có nhiều câu thoại, phù hợp với gameplay.

e. Bảng Truthfulness

- Vai trò: Điểm "thật thà" của player.
- Thiết kế logic:
  - Tách riêng để dễ mở rộng cho nhiều loại điểm khác (morality, reputation).
  - 1-1 với Player, dễ truy cập và update trực tiếp khi lựa chọn hội thoại.

f. Bảng Checkpoint

- Vai trò: Lưu trạng thái lưu game.
- Thiết kế logic:
  - Không gán trực tiếp vào Player (tránh dư thừa), mà Player có FK last_checkpoint_id.
  - Hỗ trợ nhiều checkpoint trong game, Player có thể quay lại checkpoint bất kỳ.

3.4.2.2 Quan hệ giữa các bảng

- **1-n (Player → ActiveBuff):** Một player có thể có nhiều buff cùng lúc.
- **1-n (Puzzle → PuzzlePiece):** Một câu đố bao gồm nhiều mảnh ghép.
- **1-n (NPC → Dialogue):** Một NPC có nhiều câu thoại.
- **1-1(Player → Truthfulness):** Mỗi player có điểm tính cách duy nhất.
- **0..1 (Player → Checkpoint):** Player có thể chưa tới checkpoint nào, nên là 0..1.

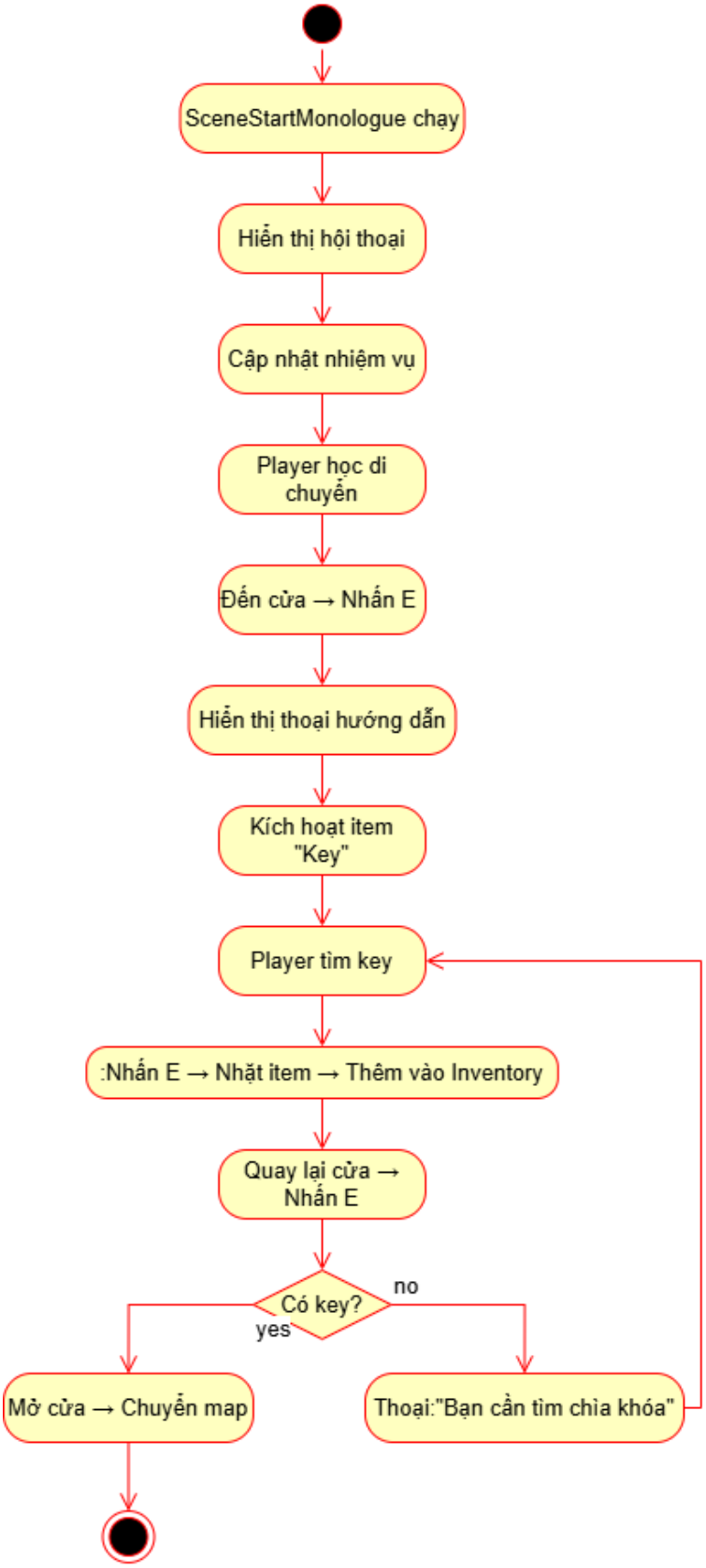
Các quan hệ này cho phép xử lý các hệ thống gameplay riêng biệt nhưng liên kết chặt chẽ, đồng thời hỗ trợ truy vấn tối ưu, ví dụ:

- Kiểm tra buff hiện tại: join Player + ActiveBuff

- Kiểm tra hoàn thành câu đố: join Puzzle + PuzzlePiece
- Xử lý hội thoại NPC: join NPC + Dialogue + Truthfulness

3.4.3 Sơ đồ hoạt động (Activity Diagram)

3.4.3.1 Sơ đồ hoạt động khám map hướng dẫn



Hình 3.7 Sơ đồ hoạt động khám phá map hướng dẫn

Diễn giải chi tiết từng bước:

5. Bắt đầu (Nút tròn đen): Luồng hoạt động của màn chơi bắt đầu.
6. SceneStartMonologue chạy: Khi cảnh (scene) bắt đầu, một đoạn độc thoại (monologue) hoặc một đoạn cắt cảnh (cutscene) ngắn sẽ được chạy để giới thiệu bối cảnh hoặc câu chuyện cho người chơi.
7. Hiển thị hội thoại: Sau đoạn độc thoại, một hộp thoại xuất hiện trên màn hình, có thể là lời của nhân vật chính hoặc một nhân vật hướng dẫn, cung cấp thông tin ban đầu.
8. Cập nhật nhiệm vụ: Hệ thống cập nhật nhật ký nhiệm vụ của người chơi. Ví dụ: nhiệm vụ mới có thể là "Tìm cách thoát khỏi căn phòng".
9. Player học di chuyển: Đây là giai đoạn hướng dẫn người chơi các thao tác di chuyển cơ bản (ví dụ: sử dụng các phím W, A, S, D).
10. Đến cửa → Nhấn E: Người chơi được hướng dẫn di chuyển đến một cánh cửa và tương tác với nó bằng cách nhấn phím "E" (một phím tương tác phổ biến trong game).
11. Hiển thị thoại hướng dẫn: Khi người chơi tương tác với cửa, một hộp thoại hướng dẫn khác sẽ hiện lên. Rất có thể nội dung là: "Cánh cửa đã bị khóa. Bạn cần tìm chìa khóa."
12. Kích hoạt item "Key": Sau khi người chơi biết mình cần chìa khóa, hệ thống game sẽ "kích hoạt" hoặc "spawn" vật phẩm chìa khóa ("Key") ở một vị trí nào đó trong màn chơi để người chơi có thể tìm thấy.
13. Player tìm key: Nhiệm vụ của người chơi lúc này là khám phá khu vực xung quanh để tìm chiếc chìa khóa vừa được kích hoạt.
14. Nhấn E → Nhặt item → Thêm vào Inventory: Khi tìm thấy chìa khóa, người chơi lại nhấn phím "E" để nhặt nó. Vật phẩm "Key" sẽ được thêm vào túi đồ (Inventory) của người chơi.
15. Quay lại cửa → Nhấn E: Sau khi đã có chìa khóa, người chơi quay trở lại cánh cửa và nhấn "E" một lần nữa để thử mở nó.

16. Có key? (Nút kiểm tra điều kiện): Tại đây, hệ thống sẽ kiểm tra xem trong túi đồ của người chơi có vật phẩm "Key" hay không.

- Nếu CÓ (Nhánh "yes"):
    - Mở cửa → Chuyển map: Cửa sẽ mở ra. Người chơi đi qua và được chuyển sang màn chơi/bản đồ (map) tiếp theo.
    - Kết thúc (Nút tròn đen có viền): Luồng hoạt động của màn hướng dẫn này kết thúc.
- Nếu KHÔNG (Nhánh "no"):
    - Thoại: "Bạn cần tìm chìa khóa": Một thông báo sẽ hiện lên cho người chơi biết rằng họ vẫn cần tìm chìa khóa.

Vòng lặp: Luồng sẽ quay trở lại bước "Player tìm key". Điều này tạo ra một vòng lặp, buộc người chơi phải tiếp tục tìm kiếm cho đến khi họ nhặt được chìa khóa và quay lại mở cửa.

3.4.3.2 Sơ đồ hoạt động khám phá mê cung và tìm đường thoát

Activity Diagram mô tả luồng hoạt động của Player cho một nhiệm vụ cụ thể: tìm rương, giải puzzle, nhận key và mở cổng thoát.

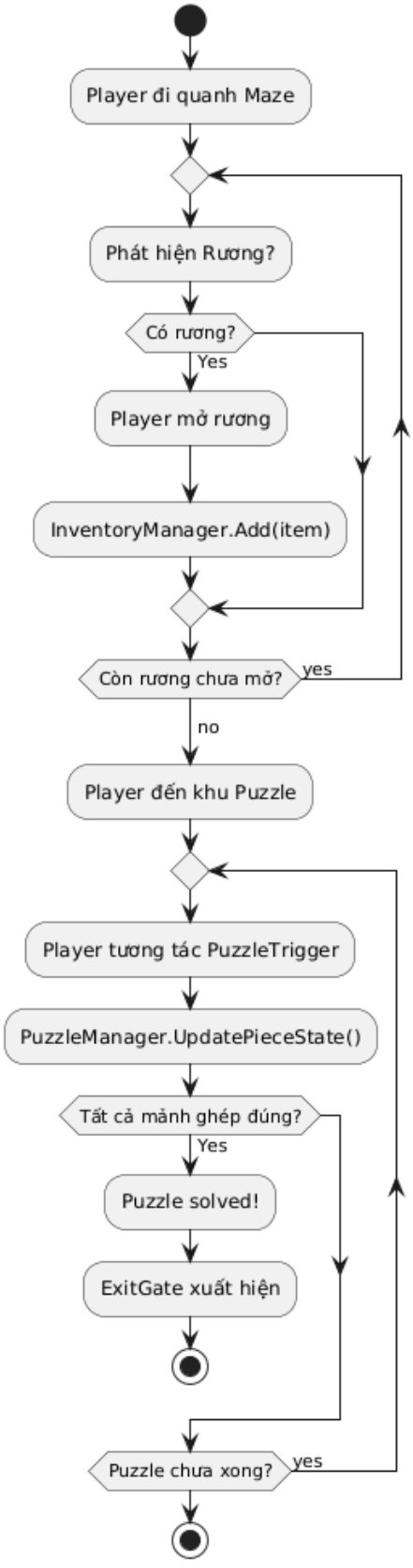
Sơ đồ bắt đầu khi người chơi nhận nhiệm vụ trong Maze. Player bắt đầu di chuyển tự do trong môi trường để tìm vị trí của rương chứa vật phẩm cần thiết. Khi Player tới gần rương và nhấn phím tương tác, trạng thái chuyển sang hoạt động "Mở giao diện puzzle"**.**

Giai đoạn giải puzzle gồm hai nhánh:

- Người chơi giải đúng → Thành công → Nhận key → Chuyển sang hoạt động tiếp theo.
- Người chơi giải sai → Tiếp tục thử lại (loop).

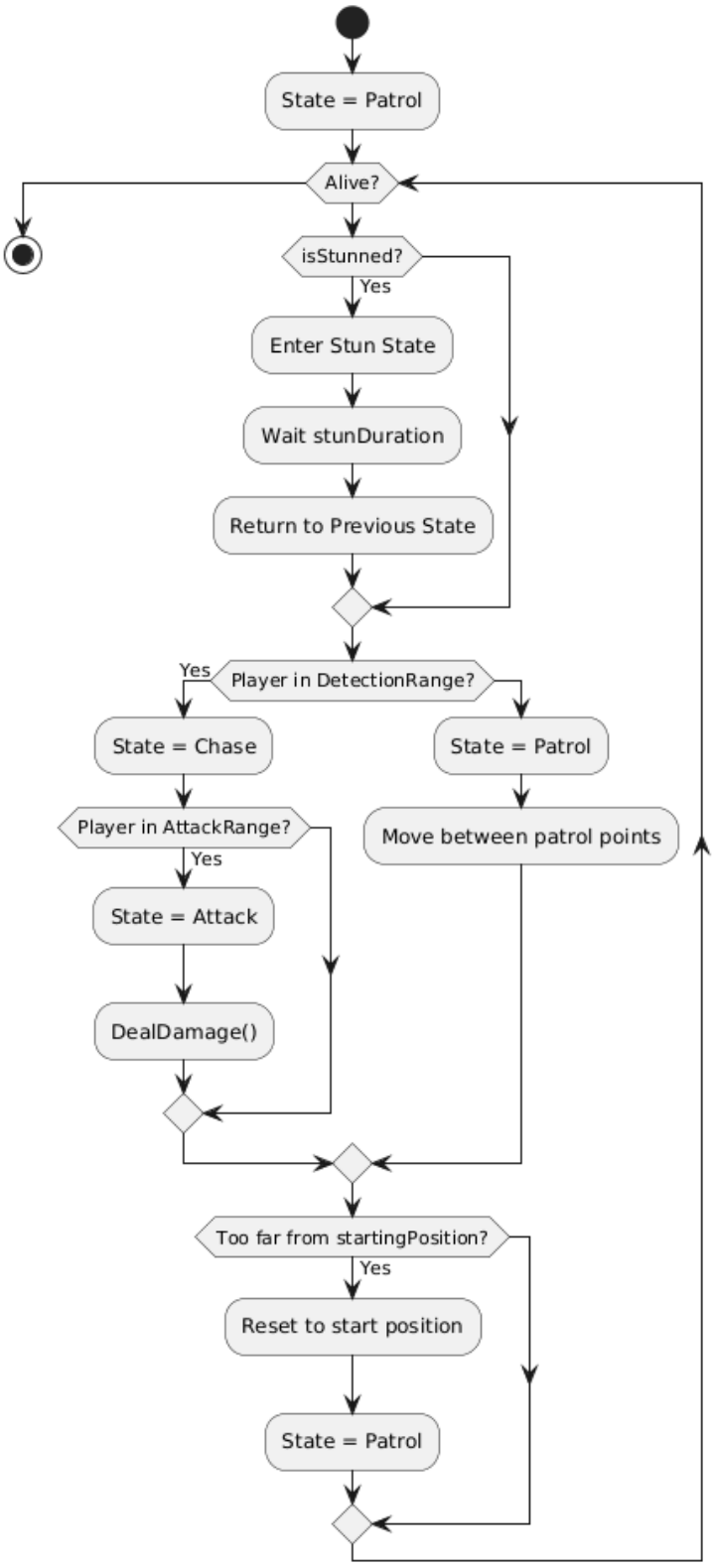
Sau khi có key, Player quay lại cổng thoát. Tại đây có kiểm tra điều kiện:

- Nếu không có key, một thông báo hiển thị và flow quay lại bước "Player tìm rương" hoặc "Player tự do khám phá".
- Nếu có key, hoạt động "Mở cổng" xảy ra và nhiệm vụ kết thúc.



Hình 3.8 Sơ đồ hoạt động khám phá mê cung và tìm đường thoát

3.4.3.3 Sơ đồ luồng hoạt động của Enemy AI FSM



Hình 3.9 Sơ đồ luồng hoạt động của Enemy AI FSM

Flowchart mô tả logic ra quyết định của Enemy AI trong maze. Đây là bản đơn giản hóa FSM (Finite State Machine) gồm các trạng thái chính: Patrol – Detect – Chase – Attack – Lost Target.

- Khi AI bắt đầu, nó ở trạng thái Patrol, di chuyển giữa các waypoint trong maze. Ở mỗi chu kỳ cập nhật, AI thực hiện kiểm tra xem Player có nằm trong tầm nhìn không.
- Nếu Player xuất hiện và nằm trong line of sight, AI chuyển sang trạng thái Detect, sau đó là Chase. Trong trạng thái Chase, Enemy liên tục cập nhật hướng di chuyển dựa trên vị trí Player.
- Nếu khoảng cách đủ gần, AI chuyển sang hành động Attack: gây sát thương cho Player.
- Nếu mất dấu Player (vượt quá khoảng cách hoặc bị khuất tầm nhìn quá lâu), AI chuyển sang Lost Target, quay lại waypoint gần nhất, rồi trở về Patrol.

3.4.3.4 Sơ đồ luồng hoạt động Sokoban

Đây là sơ đồ mô tả logic điển hình của Sokoban, bao gồm các bước:

1. **Player chạm PuzzleEntry**

Khi nhân vật tiếp xúc khu vực kích hoạt, puzzle Sokoban được bắt đầu.

2. **BoardGen tạo map Sokoban**

Hệ thống sinh bản đồ từ dữ liệu có sẵn (layout, crate, goal, tường…).

3. **Chế độ puzzle bắt đầu**
   - Khóa điều khiển ở thế giới chính
   - Chuyển sang chế độ puzzle mini-game
4. **Nhận input điều khiển (lên/xuống/trái/phải)**

Mỗi bước di chuyển được xử lý tuần tự.

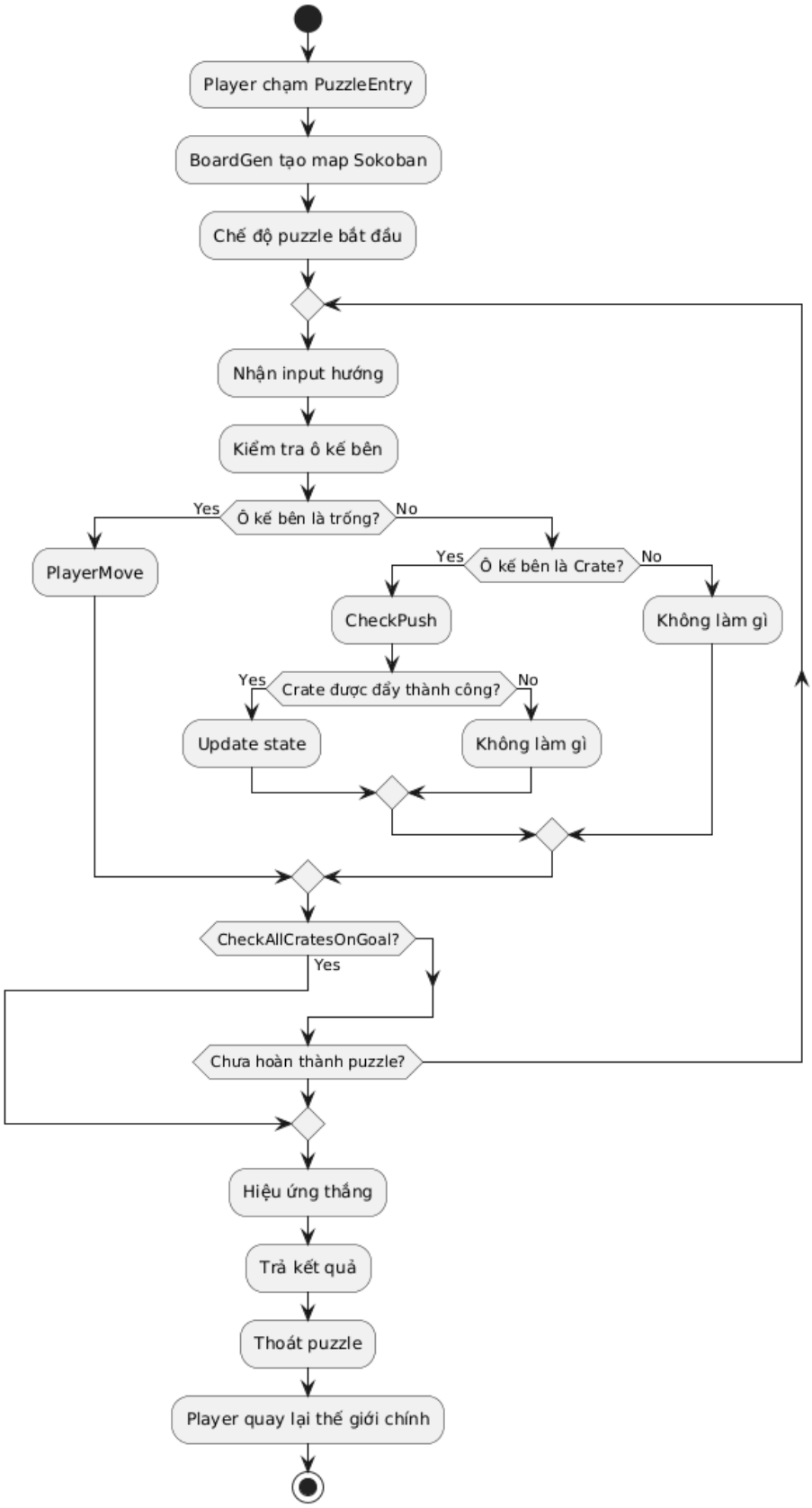
5. **Kiểm tra ô kế bên dựa trên hướng input**
   - Nếu ô kế bên **trống** → Player di chuyển.

- Nếu ô kế bên **là crate** → chuyển sang bước kiểm tra đẩy.

6. **CheckPush**
    - Kiểm tra ô đằng sau crate có trống không.
    - Nếu trống → crate được đẩy.
7. **Update state**
    - Cập nhật vị trí Player
    - Cập nhật vị trí crate
    - Đánh dấu crate có nằm đúng mục tiêu (goal) không
8. **CheckAllCratesOnGoal**

Nếu tất cả crate đứng đúng vị trí goal → puzzle hoàn thành

9. **Puzzle hoàn thành**
    - Phát hiệu ứng chiến thắng
    - Trả kết quả về gameplay chính
    - Thoát chế độ puzzle
10. **Player quay về map chính**



Hình 3.10 Sơ đồ hoạt động Sokoban

3.4.3.5 Sơ đồ hoạt động ghép giấy

Đây là mini-game ghép hình gồm nhiều mảnh giấy, cấu trúc của puzzle dạng lắp ghép, tập trung vào thu thập, ghép hình, và mở khóa nội dung mới… Mô tả chi tiết luồng như sau:

1. **Player đang ở Map chính**

Người chơi đi khám phá khu vực và tìm các mảnh giấy.

2. **Thu thập từng mảnh giấy**

Mỗi khi Player nhặt: Gọi AddPaperPiece(id) để lưu vào bộ sưu tập.

3. **Khi đủ tất cả mảnh**

PuzzleEntry được mở khóa (hiện biểu tượng / ánh sáng / bàn puzzle).

4. **Player tương tác PuzzleEntry**

Nhấn nút / click để bắt đầu puzzle ghép hình.

5. **Load UI Puzzle**

- Chuyển sang Canvas chứa mini-game
- Ẩn UI của map chính

6. **Shuffle mảnh ghép**

Toàn bộ các mảnh bị xáo trộn ngẫu nhiên

Người chơi kéo–thả (DragDrop) từng mảnh vào đúng vị trí.

7. **Kiểm tra hoàn thành (CheckCompletion)**

Nếu tất cả mảnh đúng vị trí → thắng

8. **Ghép hoàn chỉnh**

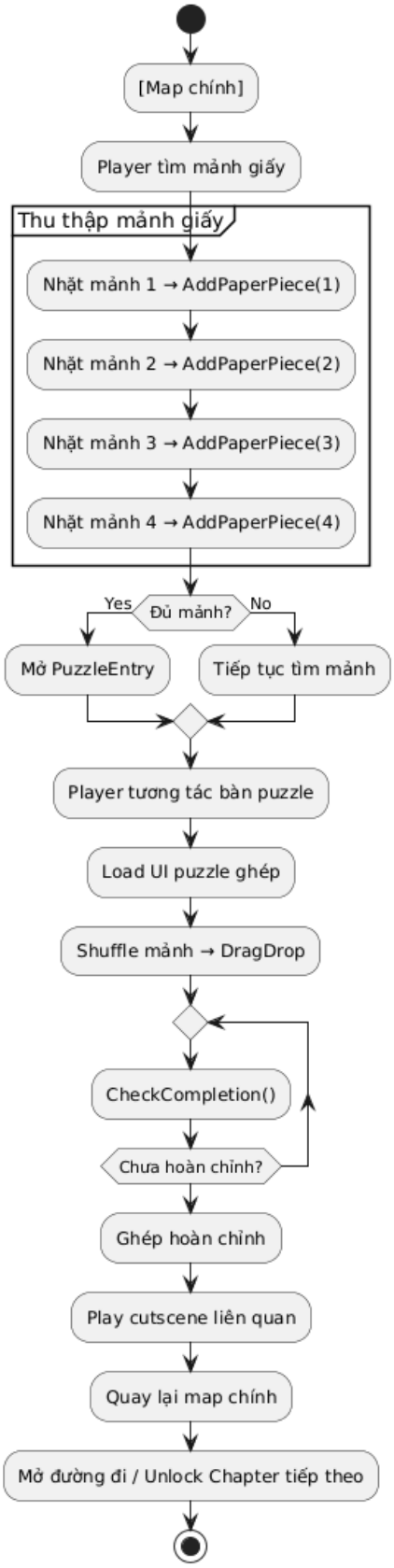
- Ghép ảnh hoàn thiện
- Hiệu ứng sáng / âm thanh chiến thắng

9. **Phát cutscene**

- Hồi ức
- Mở khóa lore
- Hình ảnh câu chuyện

10. **Quay lại map chính**

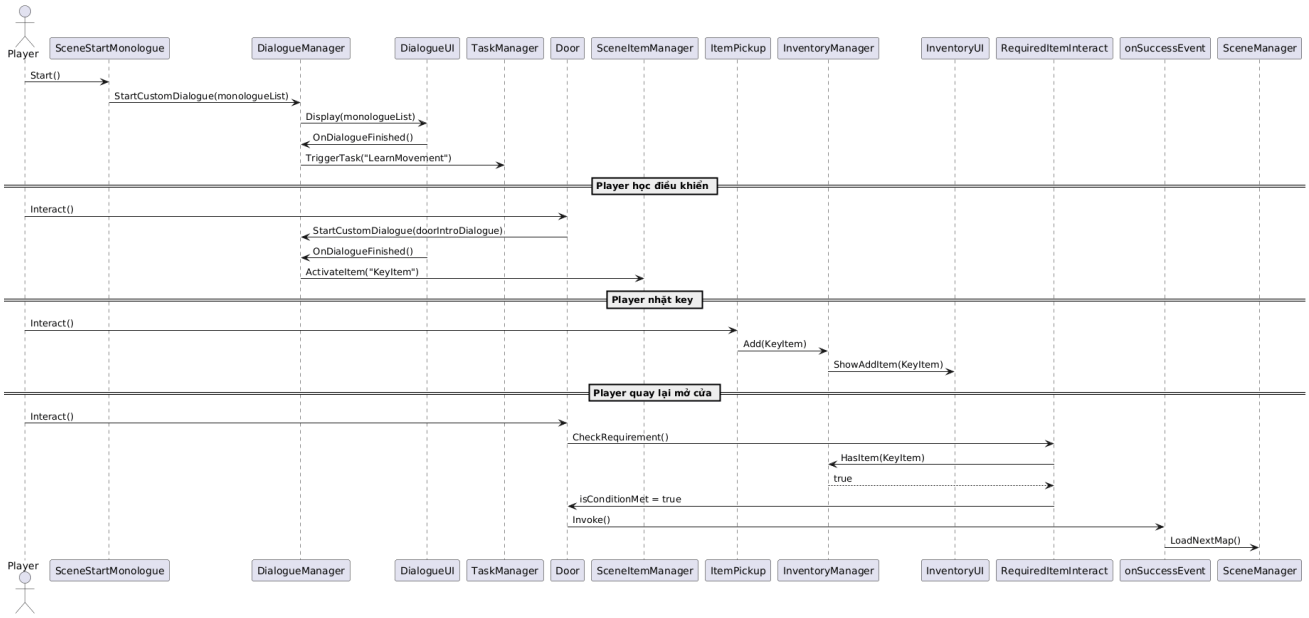
11. **Mở đường đi mới / unlock chapter**
    - Unlock quest
    - Mở một vùng bị khóa
    - Trigger story mới



Hình 3.11 Sơ đồ hoạt động ghép giấy

3.4.4 Sơ đồ tuần tự (Sequence Diagram)

3.4.4.1 Sơ đồ tuần tự khám phá map hướng dẫn



Hình 3.12 Sơ đồ tuần tự khám phá map hướng dẫn

Các đối tượng tham gia:

- Player: Người chơi, là tác nhân chính khởi tạo các hành động.
- SceneStartMonologue: Một đối tượng kịch bản (script) chịu trách nhiệm bắt đầu đoạn độc thoại khi màn chơi bắt đầu.
- DialogueManager: Quản lý việc hiển thị và luồng của các đoạn hội thoại.
- DialogueUI: Chịu trách nhiệm hiển thị giao diện người dùng (UI) của hộp thoại.
- TaskManager: Quản lý và theo dõi các nhiệm vụ của người chơi.
- Door: Đối tượng cánh cửa trong game mà người chơi tương tác.
- SceneItemManager: Quản lý các vật phẩm trong màn chơi, ví dụ như việc kích hoạt/hiện chúng ra.
- ItemPickup: Đối tượng vật phẩm có thể nhặt được (trong trường hợp này là chìa khóa).
- InventoryManager: Quản lý túi đồ (dữ liệu) của người chơi.

- InventoryUI: Chịu trách nhiệm hiển thị giao diện túi đồ.
- RequiredItemInteract: Một thành phần kiểm tra xem người chơi có vật phẩm cần thiết để tương tác hay không.
- onSuccessEvent: Một sự kiện được kích hoạt khi tương tác thành công.
- SceneManager: Quản lý việc chuyển đổi giữa các màn chơi (scenes).

Diễn giải chi tiết luồng sự kiện:

1. Bắt đầu màn chơi (Scene Start)

- Player -> SceneStartMonologue: Khi màn chơi bắt đầu, phương thức Start() được gọi.
- SceneStartMonologue -> DialogueManager: Nó yêu cầu DialogueManager bắt đầu một đoạn hội thoại tùy chỉnh (StartCustomDialogue) với nội dung là monologueList.
- DialogueManager -> DialogueUI: DialogueManager ra lệnh cho DialogueUI hiển thị (Display) nội dung hội thoại đó lên màn hình.
- DialogueManager -> TaskManager: Khi hội thoại kết thúc (OnDialogueFinished), DialogueManager thông báo cho TaskManager kích hoạt nhiệm vụ mới là "LearnMovement" (TriggerTask).
- Giai đoạn "Player học điều khiển": Tại thời điểm này, hệ thống chờ người chơi tự do di chuyển và khám phá.

2. Tương tác với Cửa lần đầu

- Player -> Door: Người chơi tiến đến và tương tác (Interact()) với đối tượng Door.
- Door -> DialogueManager: Cánh cửa bị khóa, nên nó kích hoạt một đoạn hội thoại khác (doorIntroDialogue), có thể là "Cửa đã bị khóa".
- DialogueManager -> SceneItemManager: Sau khi hội thoại này kết thúc, DialogueManager yêu cầu SceneItemManager kích hoạt (ActivateItem) vật phẩm có tên là "KeyItem". Đây là lúc chìa khóa xuất hiện trong màn chơi.

- Giai đoạn "Player nhặt key": Nhiệm vụ của người chơi bây giờ là đi tìm và nhặt chìa khóa.

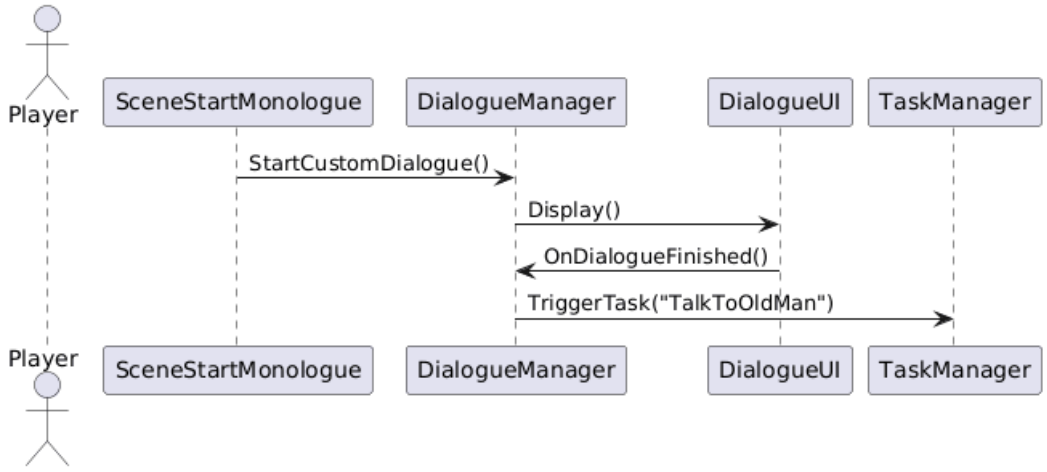
3. Nhặt Chìa khóa

- Player -> ItemPickup: Người chơi tìm thấy chìa khóa và tương tác (Interact()) với nó. ItemPickup chính là đối tượng của chìa khóa.
- ItemPickup -> InventoryManager: Đối tượng chìa khóa yêu cầu InventoryManager thêm nó vào túi đồ (Add(KeyItem)).
- InventoryManager -> InventoryUI: InventoryManager cập nhật giao diện túi đồ (ShowAddItem) để thông báo cho người chơi biết họ vừa nhặt được chìa khóa.
- Giai đoạn "Player quay lại mở cửa": Sau khi có chìa khóa, người chơi quay trở lại cánh cửa.

4. Mở Cửa thành công

- Player -> Door: Người chơi tương tác (Interact()) với Door một lần nữa.
- Door -> RequiredItemInteract: Lần này, Door gọi thành phần RequiredItemInteract của nó để kiểm tra điều kiện (CheckRequirement()).
- RequiredItemInteract -> InventoryManager: Thành phần này hỏi InventoryManager xem người chơi có vật phẩm "KeyItem" không (HasItem(KeyItem)).
- InventoryManager -> RequiredItemInteract: InventoryManager xác nhận là có và trả về giá trị true.
- RequiredItemInteract -> onSuccessEvent: Vì điều kiện đã được thỏa mãn (isConditionMet = true), nó kích hoạt sự kiện thành công (Invoke()).
- onSuccessEvent -> SceneManager: Sự kiện thành công này ra lệnh cho SceneManager tải màn chơi tiếp theo (LoadNextMap()).

3.4.4.2 Sơ đồ tuần tự mở hội thoại mở đầu chapter Mê cung



Hình 3.13 Sơ đồ tuần tự mở hội thoại mở đầu chapter Mê cung

Sơ đồ này mô tả một luồng sự kiện rất phổ biến ở đầu game hoặc đầu một màn chơi mới: tự động bắt đầu một cuộc hội thoại để giới thiệu bối cảnh và sau đó giao nhiệm vụ đầu tiên cho người chơi.

Các đối tượng tham gia:

- **Player:** Người chơi, là tác nhân (actor) chính trong game, nhưng trong luồng này, họ chỉ là người tiếp nhận thông tin một cách thụ động.
- **SceneStartMonologue:** Một đối tượng kịch bản (script) được thiết kế để tự động chạy khi màn chơi (scene) bắt đầu. Vai trò của nó là khởi động chuỗi sự kiện này.
- **DialogueManager:** "Trình quản lý hội thoại". Đây là bộ não xử lý logic của các cuộc hội thoại. Nó quyết định hội thoại nào sẽ được hiển thị và điều gì sẽ xảy ra sau khi hội thoại kết thúc.
- **DialogueUI:** "Giao diện người dùng của hội thoại". Đối tượng này chỉ có một nhiệm vụ: hiển thị hộp thoại, văn bản, hình ảnh nhân vật... lên màn hình cho người chơi xem. Nó tách biệt phần giao diện ra khỏi phần logic.
- **TaskManager:** "Trình quản lý nhiệm vụ". Đối tượng này chịu trách nhiệm theo dõi, kích hoạt và cập nhật các nhiệm vụ (quests) cho người chơi.

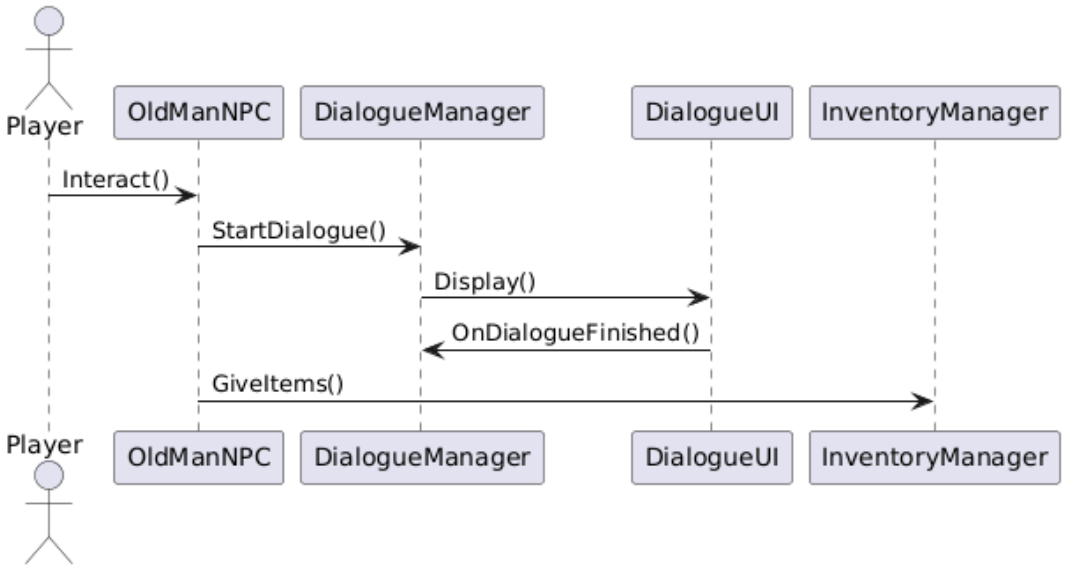
Diễn giải chi tiết luồng sự kiện (theo thứ tự từ trên xuống):

1. **SceneStartMonologue → DialogueManager : StartCustomDialogue()**

- Sự kiện: Khi màn chơi được tải, đối tượng SceneStartMonologue được kích hoạt.
- Hành động**:** Nó ngay lập tức gửi một thông điệp (gọi hàm) đến DialogueManager, yêu cầu bắt đầu một cuộc hội thoại đã được định sẵn.

2. **DialogueManager → DialogueUI : Display()**
    - Sự kiện: DialogueManager nhận được yêu cầu bắt đầu hội thoại.
    - Hành động: DialogueManager xử lý logic và sau đó ra lệnh cho DialogueUI hiển thị nội dung hội thoại lên màn hình. Điều này thể hiện một thiết kế tốt: Manager quản lý logic, còn UI chỉ lo việc hiển thị.
3. **DialogueUI → DialogueManager : OnDialogueFinished()**
    - Sự kiện: Người chơi đã đọc xong đoạn hội thoại (ví dụ: bằng cách nhấp chuột hoặc nhấn một phím).
    - Hành động: DialogueUI gửi một thông báo phản hồi lại cho DialogueManager để báo rằng hội thoại đã kết thúc.
4. **DialogueManager → TaskManager : TriggerTask("TalkToOldMan")**
    - Sự kiện: DialogueManager biết rằng hội thoại giới thiệu đã hoàn tất.
    - Hà**nh động:** Nó tiếp tục luồng sự kiện bằng cách gửi một yêu cầu đến TaskManager để kích hoạt một nhiệm vụ mới cho người chơi. Nhiệm vụ này có tên hoặc mã định danh là **"TalkToOldMan"** (Nói chuyện với người đàn ông lớn tuổi).

3.4.4.3 Sơ đồ tuần tự nhiệm vụ nói chuyện với ông già bí ẩn



Hình 3.14 Sơ đồ tuần tự nhiệm vụ nói chuyện với ông già bí ẩn

Các đối tượng tham gia:

- **Player:** Người chơi, là người chủ động bắt đầu tương tác.
- **OldManNPC:** Nhân vật không phải người chơi (NPC), cụ thể là một "Ông lão". Đây là đối tượng mà người chơi tương tác.
- **DialogueManager:** Trình quản lý hội thoại, chịu trách nhiệm xử lý logic của cuộc trò chuyện.
- **DialogueUI:** Giao diện người dùng của hội thoại, chịu trách nhiệm hiển thị cuộc trò chuyện lên màn hình.
- **InventoryManager:** Trình quản lý túi đồ, chịu trách nhiệm xử lý việc thêm/bớt vật phẩm vào túi đồ của người chơi.

Diễn giải chi tiết luồng sự kiện:

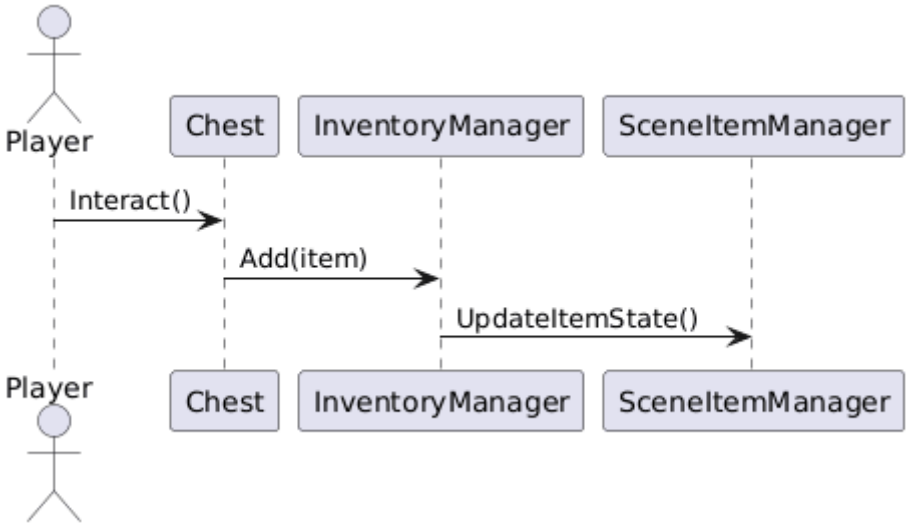
1. **Player → OldManNPC : Interact()**
    - Hành động: Người chơi di chuyển đến gần nhân vật OldManNPC và nhấn phím tương tác (ví dụ: phím E). Hành động này gửi một thông điệp Interact() đến đối tượng OldManNPC.
2. **OldManNPC → DialogueManager : StartDialogue()**
    - Hành động: Khi nhận được tín hiệu tương tác, OldManNPC không tự mình xử lý hội thoại. Thay vào đó, nó ủy thác công việc này cho một

hệ thống chuyên biệt là DialogueManager. Nó yêu cầu DialogueManager bắt đầu cuộc hội thoại được liên kết với chính nó (ông lão).

3. **DialogueManager → DialogueUI : Display()**
    - Hành động**:** DialogueManager (bộ não logic) ra lệnh cho DialogueUI (bộ phận hiển thị) để vẽ hộp thoại, hiển thị các dòng chữ và hình ảnh nhân vật lên màn hình cho người chơi xem.
4. **DialogueUI → DialogueManager : OnDialogueFinished()**
    - Hành động: Sau khi người chơi đã đọc hết cuộc trò chuyện (bằng cách nhấn nút để qua các dòng thoại), DialogueUI sẽ gửi một tín hiệu phản hồi cho DialogueManager để thông báo rằng cuộc hội thoại đã kết thúc.
5. **OldManNPC → InventoryManager : GiveItems()**
    - Hành động: Sau khi cuộc hội thoại kết thúc, kịch bản của OldManNPC tiếp tục chạy. Hành động tiếp theo là tặng vật phẩm cho người chơi. Tương tự như hội thoại, OldManNPC không trực tiếp "nhét" đồ vào túi người chơi. Nó yêu cầu InventoryManager thực hiện hành động này (GiveItems()). InventoryManager sẽ xử lý logic việc thêm các vật phẩm được định sẵn vào túi đồ của người chơi.

Sơ đồ này mô tả một quy trình hoàn chỉnh và mạch lạc: Tương tác → Bắt đầu hội thoại → Hiển thị hội thoại → Kết thúc hội thoại → Trao vật phẩm.

3.4.4.4 Sơ đồ tuần tự tương tác với vật phẩm



Hình 3.15 Sơ đồ tuần tự tương tác với vật phẩm

Các đối tượng tham gia:

- **Player:** Người chơi, là tác nhân khởi tạo hành động.
- **Chest:** Đối tượng "cái rương" trong màn chơi. Nó chứa vật phẩm mà người chơi muốn lấy.
- **InventoryManager:** "Trình quản lý túi đồ". Hệ thống này chịu trách nhiệm quản lý tất cả các vật phẩm mà người chơi sở hữu.
- **SceneItemManager:** "Trình quản lý vật phẩm trong màn chơi". Hệ thống này theo dõi trạng thái của tất cả các vật phẩm có thể tương tác *trong màn chơi hiện tại* (ví dụ: một cái rương đã được mở chưa, một vật phẩm nằm trên mặt đất đã được nhặt chưa).

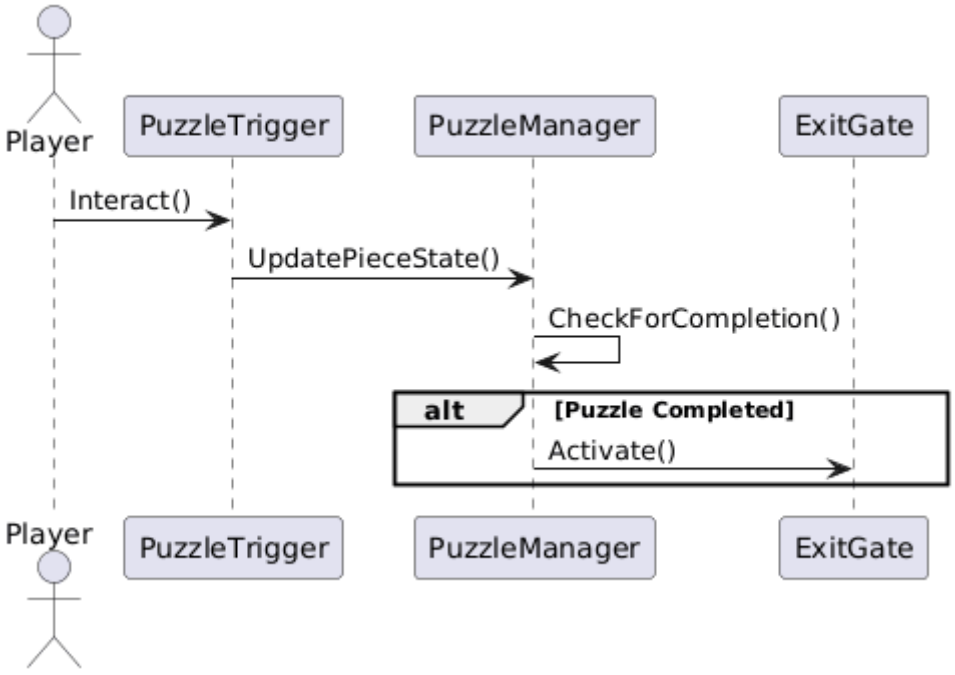
Diễn giải chi tiết luồng sự kiện:

1. **Player → Chest : Interact()**
    - Hành động**:** Người chơi đến gần cái rương và nhấn phím tương tác. Hành động này gửi một thông điệp Interact() đến đối tượng Chest.
2. **Chest → InventoryManager : Add(item)**
    - Hành động**:** Khi nhận được tương tác, đối tượng Chest không tự mình thêm vật phẩm vào túi đồ của người chơi. Thay vào đó, nó ủy thác nhiệm vụ này cho hệ thống chuyên trách là InventoryManager. Chest gửi yêu cầu Add() cùng với dữ liệu của vật

phẩm (item) mà nó chứa tới InventoryManager. InventoryManager sẽ xử lý logic để thêm vật phẩm này vào kho dữ liệu túi đồ của người chơi.

3. **InventoryManager → SceneItemManager : UpdateItemState()**
    - Hành động: Đây là một bước cực kỳ quan trọng để đảm bảo tính nhất quán của game. Sau khi vật phẩm đã được thêm thành công vào túi đồ của người chơi, InventoryManager phải thông báo cho SceneItemManager để cập nhật trạng thái của vật phẩm đó trong màn chơi.
    - Mục **đích:** Lệnh UpdateItemState() này có thể làm những việc sau:
        - Đánh dấu rằng cái rương này đã được mở và không thể mở lại để lấy đồ lần nữa.
        - Thay đổi hình ảnh của cái rương thành dạng đã mở.
        - Lưu trạng thái "đã mở" này vào file save của game, để khi người chơi quay lại khu vực này, cái rương vẫn ở trạng thái đã mở.

3.4.4.5 Sơ đồ tuần tự kích hoạt giải câu đố



Hình 3.16 Sơ đồ tuần tự kích hoạt giải câu đố

Các đối tượng tham gia:

- **Player:** Người chơi, là tác nhân khởi tạo tương tác.
- **PuzzleTrigger:** "Bộ kích hoạt câu đố". Đây là một đối tượng cụ thể trong game mà người chơi có thể tương tác, ví dụ như một cái nút, một tấm dẫm (pressure plate), hoặc một cần gạt. Nó là một *mảnh ghép* của toàn bộ câu đố.
- **PuzzleManager:** "Trình quản lý câu đố". Đây là bộ não trung tâm, chịu trách nhiệm theo dõi trạng thái của tất cả các PuzzleTrigger, biết được điều kiện để giải đố, và kích hoạt kết quả khi câu đố được hoàn thành.
- **ExitGate:** "Cổng ra". Đây là đối tượng kết quả, là phần thưởng khi giải xong câu đố. Nó có thể là một cánh cửa, một cây cầu, hoặc bất kỳ vật cản nào cần được mở/kích hoạt.

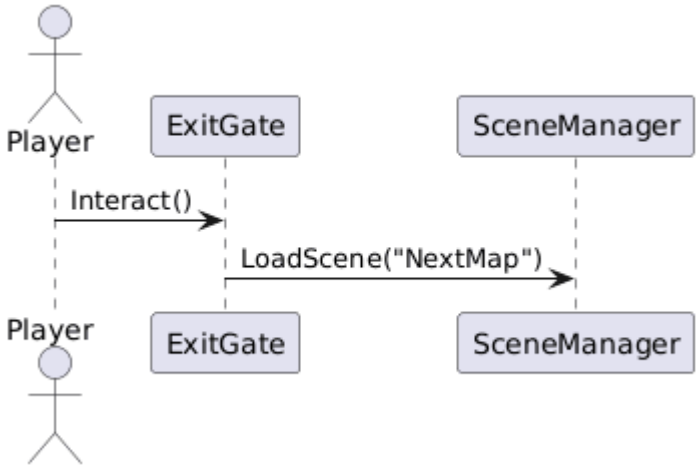
  Diễn giải chi tiết luồng sự kiện:

1. **Player → PuzzleTrigger : Interact()**
   - Hành động**:** Người chơi tương tác với một bộ phận của câu đố. Ví dụ, người chơi nhấn một cái nút.
2. **PuzzleTrigger → PuzzleManager : UpdatePieceState()**
   - Hành động: Khi được tương tác, PuzzleTrigger (cái nút) không tự mình kiểm tra xem câu đố đã hoàn thành chưa. Thay vào đó, nó thông báo cho PuzzleManager (bộ não trung tâm) rằng trạng thái của nó đã thay đổi. Ví dụ: "Nút A vừa được nhấn."
   - Ý nghĩa thiết kế**:** Điều này rất quan trọng. Từng mảnh ghép của câu đố không cần biết về logic tổng thể. Chúng chỉ cần báo cáo trạng thái của mình, giúp cho việc quản lý và tạo ra các câu đố phức tạp trở nên dễ dàng hơn.
3. **PuzzleManager → PuzzleManager : CheckForCompletion()**
   - Hành động: Sau khi nhận được thông tin cập nhật từ một mảnh ghép, PuzzleManager sẽ tự kiểm tra lại toàn bộ logic của câu đố. Mũi tên chỉ vào chính nó thể hiện đây là một hành động nội bộ. PuzzleManager sẽ so sánh trạng thái hiện tại của tất cả các mảnh

ghép với điều kiện chiến thắng (ví dụ: "Có phải cả 3 nút đã được nhấn không?").

4. **Khối alt [Puzzle Completed] (Khối lựa chọn thay thế)**
    - Ý nghĩa: Khối alt (alternative) trong sơ đồ tuần tự biểu thị một đoạn logic có điều kiện. Các hành động bên trong khối này chỉ xảy ra nếu điều kiện [Puzzle Completed] (Câu đố đã hoàn thành) là đúng.
    - Nếu sau bước CheckForCompletion(), câu đố vẫn chưa được giải, thì luồng sự kiện sẽ kết thúc tại đây và không có gì xảy ra tiếp theo.
    - Nếu câu đố đã được giải, luồng sẽ đi vào bên trong khối alt.
5. **PuzzleManager → ExitGate : Activate()**
    - Hành động: Vì điều kiện "Câu đố đã hoàn thành" là đúng, PuzzleManager sẽ gửi một thông điệp Activate() đến đối tượng ExitGate.
    - Kết quả: Cánh cửa (ExitGate) sẽ nhận được lệnh này và thực hiện hành động tương ứng, chẳng hạn như phát một đoạn phim hoạt ảnh mở cửa, vô hiệu hóa vật cản, cho phép người chơi đi qua.

3.4.4.6 Sơ đồ tuần tự luồng hoạt động thoát khỏi mê cung



Hình 3.17 Sơ đồ tuần tự luồng hoạt động thoát khỏi mê cung

Sequence này thể hiện tương tác giữa Player, PuzzleTrigger, PuzzleManager và ExitGate khi người chơi giải câu đố trong Maze. Nó tập trung vào cơ chế kích hoạt puzzle và mở cổng thoát.

Diễn giải chi tiết

1. **Player tương tác PuzzleTrigger**:
    - Khi Player tới gần một đối tượng puzzle (ví dụ nút bấm, cần gạt), hành động Interact() được gọi.
    - PuzzleTrigger nhận sự kiện này và gửi tín hiệu đến PuzzleManager.
2. **PuzzleManager cập nhật trạng thái mảnh ghép**:
    - PuzzleManager lưu trạng thái currentState của mảnh ghép tương ứng (pieceId) dựa trên hành động của Player.
    - Mỗi lần trạng thái được cập nhật, PuzzleManager kiểm tra tất cả các mảnh ghép đã thỏa điều kiện hoàn thành chưa.
3. **Kiểm tra hoàn thành puzzle**:
    - Nếu tất cả các mảnh ghép đều đạt requiredState, PuzzleManager xác nhận puzzle đã được giải thành công.
    - Sự kiện onPuzzleSolved được kích hoạt một lần duy nhất.
4. **Kích hoạt cổng thoát (ExitGate)**:
    - Khi puzzle hoàn thành, ExitGate nhận sự kiện Activate().
    - Cổng thoát trở nên khả dụng để Player tương tác và hoàn thành Chapter Maze.

CHƯƠNG 4. HIỆN THỰC HỆ THỐNG

4.1 Kiến trúc tổng quát

Sơ đồ folder Unity

Mô hình quản lý scene (Menu → Train → Chapter → End)

Kiến trúc tổng quát của game Chuyến Tàu Cuối Cùng được thiết kế theo hướng module hóa, chia thành các hệ thống độc lập nhưng có khả năng giao tiếp với nhau. Điều này giúp dự án dễ mở rộng, dễ bảo trì và tránh xung đột giữa các thành phần.

4.1.1 Mô hình kiến trúc phân tầng

(1) Tầng Presentation (UI Layer)

Tầng hiển thị chịu trách nhiệm giao tiếp trực tiếp với người chơi, bao gồm:

- Canvas tổng: quản lý UI trong từng scene.
- Dialogue UI: hiển thị đoạn hội thoại, lựa chọn của người chơi.
- HUD: hiển thị máu, mana, thông báo nhiệm vụ.
- Puzzle UI: giao diện cho Sokoban, Jigsaw, popup hướng dẫn.
- Menu UI & Pause UI.

2) Tầng Gameplay Logic (Core Game Layer)

Đây là tầng quan trọng nhất, nơi xử lý toàn bộ hành vi và logic của game:

- PlayerController – điều khiển nhân vật, xử lý input.
- EnemyController + FSM – xử lý AI tuần tra, đuổi theo, tấn công.
- DialogueManager – quản lý hội thoại, phân nhánh.
- PuzzleManager – điều khiển Sokoban, Jigsaw, kiểm tra điều kiện hoàn thành.
- TaskManager – quản lý nhiệm vụ theo từng chapter.
- SceneItemManager – bật/tắt vật phẩm, cập nhật trạng thái object.
- CameraController + Cinemachine – theo dõi người chơi, chuyển cảnh mượt mà.

- CutsceneController – điều khiển event tự động (intro chapter, kết thúc chapter).

Tầng này được viết theo kiểu component-based trong Unity, kết hợp với event-driven để giảm phụ thuộc.

(3) Tầng Data & State (Data Layer)

Bao gồm toàn bộ dữ liệu tạm và dữ liệu bền:

- InventoryManager – quản lý vật phẩm, tương tác nhặt đồ.
- TruthfulnessSystem – lưu và cập nhật chỉ số thành thật của nhân vật.
- SaveSystem (JSON I/O) – lưu checkpoint, tiến trình puzzle, chỉ số, NPC state.
- ScriptableObject Data – lưu dữ liệu cố định như dialogue, item info, NPC meta.
- PlayerPrefs – lưu cấu hình (âm lượng, đồ họa).

(4) Tầng Asset & Resource (Resource Layer)

Bao gồm tài nguyên:

- Sprite, Animation, Tilemap
- Âm thanh, nhạc nền
- Prefab cho NPC, Item, Enemy, Object tương tác
- Tất cả Scene: Menu, Train Hub, Chapter 1–4, Final Scene, Ending

Tài nguyên được tổ chức theo cấu trúc thư mục mô-đun để dễ tìm kiếm và thay thế.

4.1.2 Quản lý Scene và luồng di chuyển của game

Trò chơi sử dụng mô hình scene theo chuỗi chương:

Main Menu → Train Hub → Chapter 1 → … → Chapter 4 → Final Judgement → Ending

- Train Hub: Scene trung tâm: Đóng vai trò như lobby kết nối giữa các chapter. Sau mỗi chapter, người chơi trở về toa tàu tương ứng. Một số cửa sổ, NPC và memory fragment chỉ mở khi chapter trước hoàn thành.

- Scene chuyển đổi cảnh:
    - FadeIn/FadeOut
    - Lưu checkpoint ngay trước khi chuyển scene
    - Prefab LoadingScreen được bật nếu cần tải nhiều tài nguyên

4.2 Một bài giải thuật quan trọng

4.2.1 Mẫu thiết kế Singleton

Trong quá trình vận hành game, sẽ có những đối tượng mà toàn bộ hệ thống chỉ nên tồn tại đúng một thực thể, chẳng hạn như nhân vật chính, kho đồ (inventory), hay hệ thống quản lý âm thanh. Những đối tượng này đóng vai trò trung tâm và cần được truy cập từ nhiều nơi khác nhau trong game, nên việc đảm bảo chúng chỉ tồn tại duy nhất một bản là rất quan trọng.

Vì lý do đó, Singleton Pattern trở thành một giải pháp phù hợp. Pattern này giúp chúng ta kiểm soát số lượng thể hiện của một lớp, đồng thời cung cấp một điểm truy cập chung cho toàn bộ chương trình.

Trong phạm vi báo cáo, nhóm lựa chọn ví dụ về Inventory để minh họa cách áp dụng Singleton nhằm quản lý dữ liệu kho đồ thống nhất và tránh việc tạo ra nhiều bản sao gây xung đột.

```
26 references | Unity Script (9 asset references)
public class InventoryManager : MonoBehaviour
{
    #region Singleton
    26 references | Unity Serialized Field
    public static InventoryManager instance;

    0 references | Unity Message
    private void Awake()
    {
        if (instance == null)
        {
            instance = this;
            if (gameObject.scene.name != "DontDestroyOnLoad")
            {
                DontDestroyOnLoad(gameObject);
            }
        }
        else
        {
            Debug.LogError($"Duplicate InventoryManager found on GameObject: {gameObject.name}. Destroying it.");
            Destroy(gameObject);
            return;
        }
    }
    #endregion
```

Hình 4.1 Inventory áp dụng Singleton

Để áp dụng mẫu Singleton trong trường hợp này thì ta cần phải khai báo đối tượng Inventory Manager là static. Bình thường, ở mẫu singleton truyền thống, chúng ta sẽ cần phải tạo phương thức khởi tạo là private, sau đó khởi tạo và lấy đối tượng trong phương thức getInstance(). Nhưng với Unity, ta sẽ làm hơi khác vì Unity không cho phép chúng ta định nghĩa phương thức khởi tạo. Thay vào đó chúng ta sẽ định nghĩa trong Awake(). Cụ thể chúng ta sẽ hủy đối tượng(unity mặc định tạo đối tượng, sau đó chúng ta sẽ hủy đi) nếu như đã có đối tượng InventoryManager.

4.2.2 Mẫu Observer Pattern

Trong hệ thống Inventory, mỗi khi người chơi nhặt vật phẩm, sử dụng vật phẩm, xóa vật phẩm hoặc thay đổi ô đang chọn, giao diện người dùng phải được cập nhật ngay lập tức. Việc để UI liên tục kiểm tra dữ liệu (polling) sẽ gây tốn tài nguyên và khó bảo trì. Vì vậy, nhóm áp dụng **Observer Pattern** nhằm tách biệt phần quản lý dữ liệu và phần hiển thị giao diện.

Cụ thể, ba class phối hợp với nhau như sau:

InventoryManager đóng vai trò **Subject**, quản lý dữ liệu vật phẩm và phát ra hai loại sự kiện:

onInventoryChanged: kích hoạt khi có thay đổi số lượng vật phẩm (thêm/xóa).

onSelectionChanged: kích hoạt khi thay đổi ô đang được chọn.

Mỗi thay đổi về dữ liệu đều được InventoryManager gửi đi dưới dạng sự kiện.

InventoryUI đóng vai trò **Observer**, đăng ký lắng nghe hai sự kiện này. Khi nhận được thông báo từ InventoryManager, lớp InventoryUI tự động cập nhật giao diện, bao gồm:

làm mới icon của các vật phẩm trong từng slot,

thay đổi màu viền để thể hiện slot đang được chọn.

InventorySlot là các phần tử UI con. Chúng không lắng nghe sự kiện trực tiếp, mà được InventoryUI điều khiển để hiển thị trạng thái mới (icon, màu viền, slot trống…).

Nhờ ứng dụng Observer Pattern, UI luôn cập nhật đúng thời điểm mà không cần kiểm tra liên tục. Hệ thống giảm phụ thuộc giữa các lớp, tránh việc gọi chéo phức tạp, và dễ dàng mở rộng hoặc sửa đổi mà không ảnh hưởng đến cấu trúc tổng thể.

```
public class InventoryManager : MonoBehaviour
{
    Singleton

    public List<ItemData> items = new List<ItemData>();
    public int space = 4;

    public int currentSelectedSlot = 0;

    // TÁCH THÀNH 2 SỰ KIỆN RIÊNG BIỆT
    // 1. Chỉ gọi khi thêm hoặc xóa vật phẩm
    public UnityEvent onInventoryChanged;
    // 2. Chỉ gọi khi thay đổi ô lựa chọn
    public UnityEvent onSelectionChanged;

    Unity Message | 0 references
    private void Start()
    {
        // Kích hoạt cả hai event lúc đầu để UI hiển thị đúng
        onInventoryChanged.Invoke();
        onSelectionChanged.Invoke();
    }

    4 references
    public bool Add(ItemData item)
    {
        if (items.Count >= space)
        {
            Debug.Log("Not enough room in inventory.");
            return false;
        }

        if (item == null)
        {
            Debug.LogError("Trying to add NULL item to inventory!");
            return false;
        }

        bool wasEmpty = items.Count == 0;
        items.Add(item);
```

Hình 4.2 InventoryManager áp dụng Observer Pattern

```
public class InventoryUI : MonoBehaviour
{
    [Header("UI References")]
    public Transform itemsParent;
    // selectionBorder không cần nữa vì dùng màu border của slot

    private InventoryManager inventory;
    private InventorySlot[] slots;

    Unity Message | 0 references
    void Start()
    {
        // Kiểm tra null để tránh crash
        if (InventoryManager.instance == null)
        {
            Debug.LogError("InventoryManager.instance is null! Make sure there's an InventoryManager in the scene.", this);
            return;
        }

        inventory = InventoryManager.instance;

        // Đăng ký lắng nghe cho cả hai sự kiện
        inventory.onInventoryChanged.AddListener(UpdateInventoryUI);
        inventory.onSelectionChanged.AddListener(UpdateSelectionUI);

        Debug.Log("InventoryUI: Event listeners registered successfully!");

        slots = itemsParent.GetComponentsInChildren<InventorySlot>();
        Debug.Log("Inventory slots found: " + slots.Length);

        // Kiểm tra xem có tìm thấy slots không
        if (slots.Length == 0)
        {
            Debug.LogError("No InventorySlot components found! Check itemsParent reference.", this);
            return;
```

Hình 4.3 InventoryUI áp dụng Observer Pattern

```
public class InventorySlot : MonoBehaviour
{
    public Image icon; // Kéo ItemIcon vào đây
    private ItemData item;
    private Image slotBackground; // Background/border của chính sl

    Unity Message | 0 references
    private void Awake()
    {
        // Lấy Image component của chính object này (là background/
        slotBackground = GetComponent<Image>();

        // Đặt màu mặc định cho slot trống
        if (slotBackground != null)
        {
            slotBackground.color = new Color32(128, 128, 128, 255);
        }
    }

    1 reference
    public void AddItem(ItemData newItem)
    {
        item = newItem;
        icon.sprite = item.icon;
        icon.enabled = true;
    }

    1 reference
    public void ClearSlot()
    {
        item = null;
        icon.sprite = null;
        icon.enabled = false;
```

Hình 4.4 InventoryUI áp dụng Observer Pattern

CHƯƠNG 5. KẾT QUẢ

5.1 Giao diện hội thoại với nhân vật

Hình 5.1 Giao diện hội thoại với NPC

Hệ thống hội thoại dùng DialogueManager kết hợp ScriptableObject để hiển thị câu thoại và các lựa chọn. Mỗi lựa chọn dẫn đến nhánh hội thoại khác nhau và có thể thay đổi chỉ số Truthfulness.

5.2 Giao diện tương tác với đồ vật

Giao diện tương tác sử dụng cơ chế Trigger + Prompt UI. Khi người chơi bước vào vùng tương tác (OnTriggerEnter), hệ thống hiển thị thông báo như “Nhấn E để tương tác”. Nếu người chơi nhấn E, object gọi hàm Interact() để xử lý: nhặt vật phẩm, mở cửa, kiểm tra vật phẩm yêu cầu hoặc kích hoạt puzzle. Khi rời vùng tương tác (OnTriggerExit), Prompt UI tự động ẩn đi.

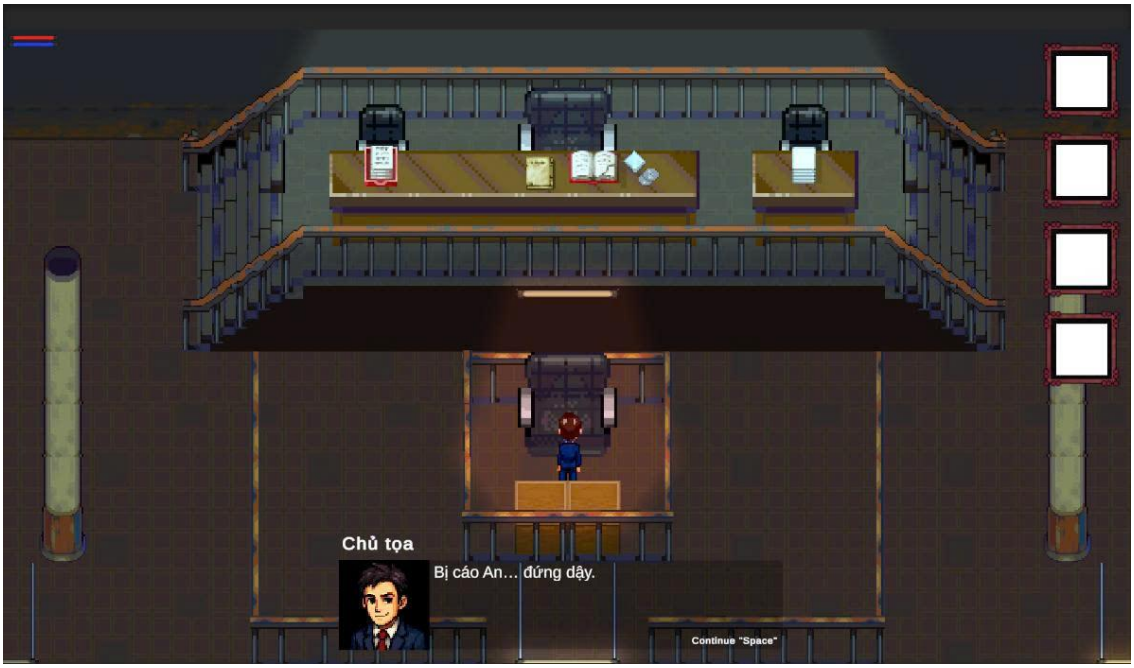
Hình 5.2 Giao diện tương tác với vật phẩm

5.3 Hệ thống AI địch

Hình 5.3 Kẻ địch rượt đuổi nhân vật

AI sử dụng FSM gồm Patrol – Chase – Attack. Khi phát hiện người chơi bằng Raycast, địch chuyển sang trạng thái Chase.

5.4 Cutscene



Hình 5.4 Cutscene hội thoại nhân vật

Giao diện cutscene dùng để hiển thị đoạn phim hoặc cảnh cốt truyện. Khi cutscene bắt đầu, UI gameplay bị khóa và lớp Overlay Cutscene xuất hiện (thường gồm nền tối, phụ đề và nút Skip). Hệ thống sẽ tự chạy animation, thoại hoặc camera timeline. Khi cutscene kết thúc hoặc người chơi nhấn Skip, UI cutscene ẩn đi và trả quyền điều khiển lại cho người chơi.

5.5 Giao diện cài đặt

Cài đặt điều khiển được lấy từ input system, các slider âm lượng, ánh sáng được lưu lại. Khi nhấn vào biểu tượng cài đặt, game được đặt tạm dừng. Sau khi nhấn Apply hoặc Cancel thì game sẽ tiếp tục.

Hình 5.5 Giao diện cài đặt

5.6 Hệ thống Puzzle

Hình 5.6 Puzzle Sokoban

Puzzle Sokoban hoạt động dựa trên lưới grid: người chơi di chuyển và đẩy bình vào đúng vị trí. Khi puzzle hoàn thành, sự kiện OnPuzzleSolved được kích hoạt.

CHƯƠNG 6. KẾT LUẬN

6.1 Kết luận

Đề tài Chuyến Tàu Cuối Cùng đã đạt được các mục tiêu đề ra ban đầu: xây dựng một trò chơi 2D kết hợp narrative – puzzle cùng yếu tố tâm lý, có hệ thống điều khiển, hội thoại, puzzle, AI địch, camera và save/load hoàn chỉnh. Các hệ thống gameplay được triển khai theo mô hình GameObject–Component của Unity, kết hợp các kỹ thuật FSM, Event-driven và kiến trúc module hóa.

Dự án cho thấy khả năng vận dụng hiệu quả kiến thức của môn Phát triển Trò chơi, đặc biệt trong việc:

- Phân tích – thiết kế logic gameplay
- Xây dựng UI và hệ thống hội thoại phân nhánh
- Thiết kế puzzle với nhiều cơ chế khác nhau
- Tổ chức code kiến trúc rõ ràng và tối ưu
- Tạo được một sản phẩm có chiều sâu nội dung và giàu ý nghĩa
- Trò chơi chạy ổn định, các scene hoạt động mượt mà, và các module giao tiếp hiệu quả theo UML đã thiết kế.

6.2 Hướng phát triển

Trong tương lai, trò chơi có thể được nâng cấp theo các hướng sau:

1. Mở rộng nội dung
    - Thêm chapter mới, mở rộng thế giới ký ức của nhân vật An.
    - Tăng số lượng NPC và hội thoại phân nhánh.
    - Xây dựng thêm ending nâng cao (True Ending / Secret Ending).
2. Cải thiện gameplay
    - Bổ sung hệ thống chiến đấu biểu tượng sâu hơn.
    - Tạo thêm các dạng puzzle mới (logic, âm thanh, vật lý…).
    - Thêm minigame theo phong cách quick-time event.
3. Cải thiện âm thanh – đồ họa

- Nâng cấp sprite lên chuẩn 16-bit hoặc 32-bit.
- Thêm hiệu ứng ánh sáng 2D (Unity URP + 2D Light).
- Tăng cường âm thanh ambience, nhạc nền theo cảm xúc.

4. Tối ưu hệ thống lưu tiến trình
    - Lưu đa slot
    - Lưu tự động theo mốc chapter
    - Đồng bộ hóa cloud save
5. Đưa game lên nhiều nền tảng
    - Windows (PC)
    - Android/iOS
    - WebGL

TÀI LIỆU THAM KHẢO

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

Microsoft. (n.d.). Observer Design Pattern. Microsoft Developer Documentation. https://learn.microsoft.com/en-us/dotnet/standard/events/observer-design-pattern