BÁO CÁO DỰ ÁN AI GAME: ASTAR GAME

# Phân Công:

|  |  |  |
| --- | --- | --- |
| Mã Sinh Viên | Tên Sinh Viên | Công Việc |
| N22DCCN139 | Văn Duy Khánh | - Lên ý tưởng.  - Tìm tài nguyên.  - Hỗ trợ tìm thư viện hỗ trợ: DOTWeen, Input.GetKewDown(),… - Hỗ trợ Code: Astar. - Làm báo cáo. |
| N22DCCN142 | Vũ Đình Khoa | - Lên ý tưởng.  - Tìm tài nguyên, thư viện. - Code - chính:   + Thuật toán Astar.  + Các hàm Manager.  + Các hàm Controller.  + Các hàm Model.  + Các hàm View.  + Logic game (Win/Lose).  + Animation (DOTWeen). - Chỉnh sửa UI trên Unity. - Làm báo cáo. |
| N22DCCN160 | Phạm Văn Phú | - Lên ý tưởng.  - Tìm tài nguyên.  - Hỗ trợ tìm thư viện hỗ trợ: DOTWeen, Input.GetKewDown(),… - Hỗ trợ Code: Astar. |
| N22DCCN180 | Phạm Hùng Thiên | - Lên ý tưởng.  - Tìm tài nguyên.  - Hỗ trợ tìm thư viện hỗ trợ: DOTWeen, Input.GetKewDown(),… - Hỗ trợ Code: Astar, Player (MVC), hiển thị PlayerView. |

# Giới thiệu tổng quan

Tên game: AStar Game.  
Công cụ phát triển: Unity (C#).  
Mục tiêu của game:  
Người chơi điều khiển nhân vật chính vượt mê cung để đến được điểm đích. Trong khi đó, kẻ địch (Enemy) sẽ liên tục đuổi theo người chơi bằng cách sử dụng thuật toán tìm đường A\*. Nếu người chơi đến được điểm đích → Win game. Nếu bị enemy bắt → Lose game.

# Gameplay và luật chơi

- Player:  
 + Điều khiển bằng phím (WASD + phím điều hướng: lên, xuống, trái và phải).  
 + Mỗi bước di chuyển được tính trên hệ lưới (matrix).   
 + Lưu vị trí vào Model của Player sau mỗi bước di chuyển.  
- Enemy (AI):  
 + Tự động di chuyển từng bước về phía Player.  
 + Tìm đường ngắn nhất dựa trên thuật toán A\*.  
 + Lưu vị trí vào Model của Enemy sau mỗi bước di chuyển.  
- Luật thắng/thua:  
 + Win: Player chạm đến vị trí đích.  
 + Lose: Enemy chạm vào vị trí của Player.

# 4. Kỹ thuật và công nghệ sử dụng

## 4.1. Mô hình MVC (Model - View - Controller)

**Giải thích ngắn gọn:** **MVC** là viết tắt của **Model - View - Controller**, một mô hình thiết kế phần mềm giúp **tách biệt ba thành phần chính trong một ứng dụng**: dữ liệu, giao diện và xử lý logic. Mô hình này thường được dùng trong phát triển game, ứng dụng web, và phần mềm có giao diện người dung:

* Model: Quản lý dữ liệu.
* View: Giao diện – hiện thị game (sprite, animation).
* Controller: Điều khiển logic.

**Luồng xử lý cơ bản diễn ra như sau:** Người dùng thực hiện hành động (input – Controller tiếp nhận hành động) 🡪 Controller phân tích hành động, sau đó cập nhật dữ liệu trong Model 🡪 **Model** thay đổi dữ liệu bên trong 🡪 View sẽ **lấy dữ liệu từ Model** để cập nhật giao diện.

**Áp dụng trong game:**

|  |  |  |
| --- | --- | --- |
| Thành phần | Vai trò | Ví dụ cụ thể |
| PlayerModel, EnemyModel,  MatrixElementModel | Model | Lưu vị trí hiện tại của Player/Enemy/Matrix. Lưu thành phần View hiển thị mà bản thân quản lý. Lưu loại Matrix tại vị trí (x, y) trên ma trận:  - 0: Có thể đi. - 1: Tường không thể đi. |
| PlayerView, EnemyView,  HomeUIView,  WinUIView,  LoseUIView | View | Hiển thị sprite. Hiển thị vị trí đang đứng trên màn hình.  Hiển thị animation (Idle, Run), Lưu hàm Move dùng DOTWeen tạo Animation di chuyển mượt mà. |
| PlayerController, EnemyController,  MatrixController,  HomeUIController | Controller | Lưu List các Model Player/Enemy 🡪 Đảm bảo chỉ có duy nhất 1 PlayerController/EnemyController.  Spawn Player/Enemy/Matrix: - Player vị trí bắt đầu là [0, 0]. - Enemy vị trí ngẫu nhiên trong Matrix. - Matrix (Size đã được lưu trong data, ví trí tường và ô có thể đi cũng được lưu trong data).  Nhận Input từ người chơi:  - Player: Input.GetKeyDown() cho phép nhận biết bàn phím đang nhấn nút nào.  Xử lý Logic di chuyển theo Direction:Tác động đến Model để thay đổi Position, Model tác động vào View để hiện thị lên màn hình (gọi thêm hàm Move để di chuyển mượt hơn bằng DOTWeen):  - Player: di chuyển sau khi nhấn nút điều khiển.  - Enemy: di chuyển sau khi Player nhấn nút, Gọi AStar cập nhật đường đi cho các Enemy.  HomeUIController: Lưu View với trường hợp đặc biệt, thực thể không cần hiển Model. |

## 4.2. Design Pattern: Singleton

**Khái quát: Singleton** là một **mẫu thiết kế (Design Pattern)** thuộc nhóm **Creational Patterns** (mẫu khởi tạo), với mục tiêu:

🔹 **Đảm bảo rằng một lớp chỉ có duy nhất một thể hiện (instance)**  
🔹 **Cung cấp một điểm truy cập toàn cục (global access point)** đến thể hiện đó.  
  
**Áp dụng trong game:**

| **Lớp (Manager / Controller)** | **Vai trò chính** | **Lý do dùng Singleton** |
| --- | --- | --- |
| **HomeUIController** | - Quản lý giao diện màn hình chính (Home Menu).  - Gọi LoadLevel theo Index. | Chỉ cần **một thể hiện duy nhất** để điều khiển UI chính. |
| **UIManager** | - Quản lý giao diện tổng thể trong game (Win, Lose, Pause...). | Là **trung tâm điều phối UI**, cần truy cập toàn cục dễ dàng qua UIManager.Instance. |
| **MatrixController** | - Quản lý bản đồ mê cung dạng ma trận. | Là **nguồn dữ liệu bản đồ duy nhất**, dùng chung cho tất cả Enemy và Player. |
| **PlayerController** | - Quản lý tất cả model Player trong game.  - Xử lý logic tổng thể cho Player khi nhận Input.  - Cập nhật vị trí cho Model Player. | Dùng Singleton đảm bảo chỉ có một nơi quản lý dữ liệu Player. |
| **EnemyController** | - Quản lý toàn bộ Enemy model.  - Gọi thuật toán A\* và xử lý logic AI cho danh sách Enemy. - Cập nhật vị trí cho danh sách các Model Enemy. | Singleton gom tất cả Enemy lại một chỗ để dễ kiểm soát và đồng bộ hóa AI. |

## 4.3. Thuật toán AStar (A\*)

Thuật toán A\* là một trong những thuật toán tìm kiếm đường đi ngắn nhất phổ biến nhất và hiệu quả nhất trong lĩnh vực trí tuệ nhân tạo và khoa học máy tính. Thuật toán này kết hợp các ưu điểm của tìm kiếm ưu tiên theo độ sâu (Depth-First Search) và tìm kiếm theo chiều rộng (Breadth-First Search) để tìm ra đường đi tối ưu từ điểm bắt đầu đến điểm đích trong một không gian tìm kiếm.

- Cơ sở lý thuyết: Thuật toán A\* sử dụng hai hàm chi phí để đánh giá các bước đi trong không gian tìm kiếm:

+ g(n): Chi phí từ điểm bắt đầu đến đỉnh n. + h(n): Ước lượng chi phí từ đỉnh n đến đích. Hàm này thường được gọi là hàm heuristic. - Tổng chi phí của một đỉnh n được tính bằng hàm f(n):

f(n) = g(n) + h(n)

Trong đó:

- g(n) là chi phí thực tế để đi từ điểm bắt đầu đến đỉnh n. - h(n) là chi phí ước lượng từ đỉnh n đến đích, dựa trên một hàm heuristic. Để đảm bảothuật toán A\* tìm được đường đi ngắn nhất, hàm heuristic phải chấp nhận được (admissible), tức là nó không bao giờ được đánh giá cao hơn chi phí thực tế.

Trình bày thuật toán A\*:

Thuật toán A\* có thể được trình bày theo các bước sau:

- Khởi tạo: Đặt đỉnh bắt đầu vào danh sách mở (open list), là một hàng đợi ưu tiên dựa trên

hàm f(n). Đặt chi phí ban đầu của đỉnh này là 0 và ước lượng chi phí đến đích bằng hàm

heuristic.

- Duyệt đỉnh: Trong khi danh sách mở không rỗng:

+ Lấy đỉnh n có giá trị f(n) thấp nhất ra khỏi danh sách mở.

+ Nếu n là đích, thuật toán kết thúc và đường đi tối ưu đã được tìm thấy.

+ Di chuyển đỉnh n vào danh sách đóng (closed list) để đánh dấu rằng nó đã được duyệt. + Mở rộng đỉnh: Đối với mỗi đỉnh kề m của n:

- Tính giá trị g(m) bằng g(n) cộng với chi phí từ n đến m.

+ Nếu m nằm trong danh sách đóng và giá trị g(m) mới thấp hơn giá trị g(m) hiện tại, bỏ qua đỉnh m.

+ Nếu m không nằm trong danh sách mở hoặc giá trị g(m) mới thấp hơn giá trị g(m) hiện tại:

+ Cập nhật g(m) và tính lại f(m).

+ Đặt n là cha của m.

+ Nếu m chưa có trong danh sách mở, thêm m vào danh sách mở.

-Thuật toán kết thúc: Khi đỉnh đích được lấy ra từ danh sách mở, đường đi tối ưu đã được xác định bằng cách truy vết từ đỉnh đích về điểm bắt đầu thông qua các đỉnh cha.

Ưu và nhược điểm của thuật toán A\*

- Ưu điểm:

Một thuật giải linh động, tổng quát, trong đó hàm chứa cả tìm kiếm chiều sâu, tìm kiếm chiều rộng và những nguyên lý Heuristic khác. Nhanh chóng tìm đến lời giải với sự định hướng của hàm Heuristic. Chính vì thế mà người ta thường nói A\* chính là thuật giải tiêu biểu cho Heuristic. - Nhược điểm:

+ Bộ nhớ: A\* có thể tiêu tốn nhiều bộ nhớ vì phải lưu trữ nhiều đỉnh trong danh sách mở và danh sách đóng. + Hiệu suất: Nếu hàm heuristic không được chọn đúng cách, thuật toán có thể trở nên không hiệu quả. Như vậy, thuật toán A\* là một công cụ mạnh mẽ cho việc tìm kiếm đường đi trong không gian tìm kiếm, đặc biệt hữu ích trong các ứng dụng như lâp kế hoạch robot, trò chơi điện tử và các hệ thống GIS.

**Mục tiêu**: Tìm đường đi ngắn nhất từ start đến goal trên lưới bằng thuật toán A\*.

**Lưới kích thước**: 16 hàng × 32 cột.

**Dữ liệu sử dụng**:

* gScore: Chi phí nhỏ nhất từ điểm bắt đầu đến một điểm bất kỳ.
* cameFrom: Lưu cha của mỗi ô để truy vết đường đi.
* openList: Tập các điểm đang xét.
* closedList: Tập các điểm đã xét xong.

**1. Khởi tạo:**

float[,] gScore = new float[16, 32];

Vector2Int[,] cameFrom = new Vector2Int[16, 32];

List<OpenListElement> openList = new List<OpenListElement>();

HashSet<Vector2Int> closedList = new HashSet<Vector2Int>();

- gScore[x, y]: chi phí đi từ start đến ô (x, y).

- cameFrom[x, y]: lưu điểm cha của ô (x, y) trên đường đi tốt nhất hiện tại.

- openList: danh sách các điểm đang được xét (dựa theo chi phí nhỏ nhất).

- closedList: danh sách các điểm đã xét.

for (int i = 0; i < 16; i++)

for (int j = 0; j < 32; j++)

{

gScore[i, j] = float.MaxValue;

cameFrom[i, j] = new Vector2Int(-1, -1);

}

- Khởi tạo gScore tất cả điểm là vô cùng (chưa biết chi phí tốt nhất).

- cameFrom mặc định là (-1, -1) để nhận biết điểm không có cha.

gScore[start.x, start.y] = 0;

openList.Add(new OpenListElement

{

GCost = 0,

FCost = 0,

Current = start,

Parent = new Vector2Int(-1, -1)

});

- Gán chi phí bắt đầu là 0.

- Thêm start vào openList.

**2. Vòng lặp chính A**\*:

while (openList.Count > 0)

Lặp khi còn điểm để xét.

a. **Chọn điểm tốt nhất**

OpenListElement current = openList.OrderBy(e => e.FCost).First();

openList.Remove(current);

closedList.Add(current.Current);

 Chọn phần tử có FCost (g = đường đi + h = dự đoán) nhỏ nhất.

 Bỏ ra khỏi openList và thêm vào closedList.

b. **Nếu đạt đến goal**

if (current.Current == goal)

Đã đến đích → truy vết đường đi.

List<Vector2Int> path = new List<Vector2Int>();

Vector2Int child = current.Current;

Vector2Int parent = cameFrom[child.x, child.y];

path.Add(goal);

while (parent.x != -1)

{

path.Add(parent);

child = parent;

parent = cameFrom[child.x, child.y];

}

path.Add(child);

return path;

Truy ngược từ goal về start bằng cameFrom để dựng lại đường đi.

c. **Duyệt các hàng xóm**

for (int i = 1; i <= 8; ++i)

8 hướng đi (giả sử Direction là enum: trái, phải, lên, xuống, chéo...).

Vector2Int neighbor = this.Actions(dir, current.Current);

if (neighbor.x == -1 || closedList.Contains(neighbor)) continue;

Bỏ qua nếu không hợp lệ (đi ra ngoài biên) hoặc đã duyệt.

float tentativeG = gScore[current.Current.x, current.Current.y] + 1;

bool inOpenList = openList.Any(e => e.Current == neighbor);

tentativeG: chi phí giả định từ start đến neighbor.

if (!inOpenList || tentativeG < gScore[neighbor.x, neighbor.y])

Nếu chưa có trong openList, hoặc có đường mới tốt hơn:

gScore[neighbor.x, neighbor.y] = tentativeG;

Vector2 pos1 = new Vector2(current.Current.x, current.Current.y);

Vector2 pos2 = new Vector2(neighbor.x, neighbor.y);

float fScore = tentativeG + this.EuclideanDistance(pos1, pos2);

cameFrom[neighbor.x, neighbor.y] = current.Current;

 Tính fScore = g + h (trong đó h là khoảng cách Euclid).

 Cập nhật đường đi tốt nhất đến neighbor.

openList.Add(new OpenListElement

{

GCost = tentativeG,

FCost = fScore,

Current = neighbor,

Parent = current.Current

});

Thêm neighbor vào openList để xử lý ở bước sau.

3. **Không tìm được đường đi**

return null;

## 4.4 DOTween – Animation

**DOTween** là một thư viện tweening (chuyển động nội suy) **miễn phí và rất mạnh mẽ** dành cho Unity, giúp **tạo hoạt ảnh cho các thuộc tính** như vị trí, màu sắc, xoay, phóng to/thu nhỏ... một cách dễ dàng và hiệu quả.

* Website: <http://dotween.demigiant.com>
* Nhà phát triển: **Demigiant**
* DOTween là hậu duệ của **HOTween**, được tối ưu tốt hơn và dễ dùng hơn

**Áp dụng trong Game:**

* DOLocalJump tạo hiệu ứng **nhảy tới vị trí mới** → thay vì di chuyển trơn, nhân vật sẽ **bật nhẹ lên rồi rơi xuống ô mới**, sinh động và hấp dẫn hơn.
* **Enemy**: Khi AI tìm được đường mới → mỗi bước đi sẽ **"bật nhảy" theo đường dẫn A\***

# 5. Xử lý logic gameplay

## 5.1 Quản lý bản đồ bằng ma trận

- Dùng mảng 2 chiều int[,].  
- Giá trị ô:  
 + 0: Ô trống có thể đi.  
 + 1: Tường (không thể đi).

## 5.2 Lưu vị trí trong Model

- Mỗi nhân vật (Player, Enemy) có x, y hiện tại trong mô hình lưới.  
- View sẽ dịch vị trí gridPosition sang toạ độ trong Canvas của Unity.  
- View sẽ gọi hàm di chuyển để hiển thị ở vị trí mới trên map.

## 5.3 Xử lý thắng thua:

- Khi nhân vật tới đích gọi hiển thị WinUIView.  
- Khi nhân vật bị Enemy chạm vào thì gọi hiển thị LoseUIView.  
- Khi người chơi bấm nút Pause, gán Time.deltaTime = 0 (dừng Game), bấm lần nữa thì gán Time.deltaTime = 1.

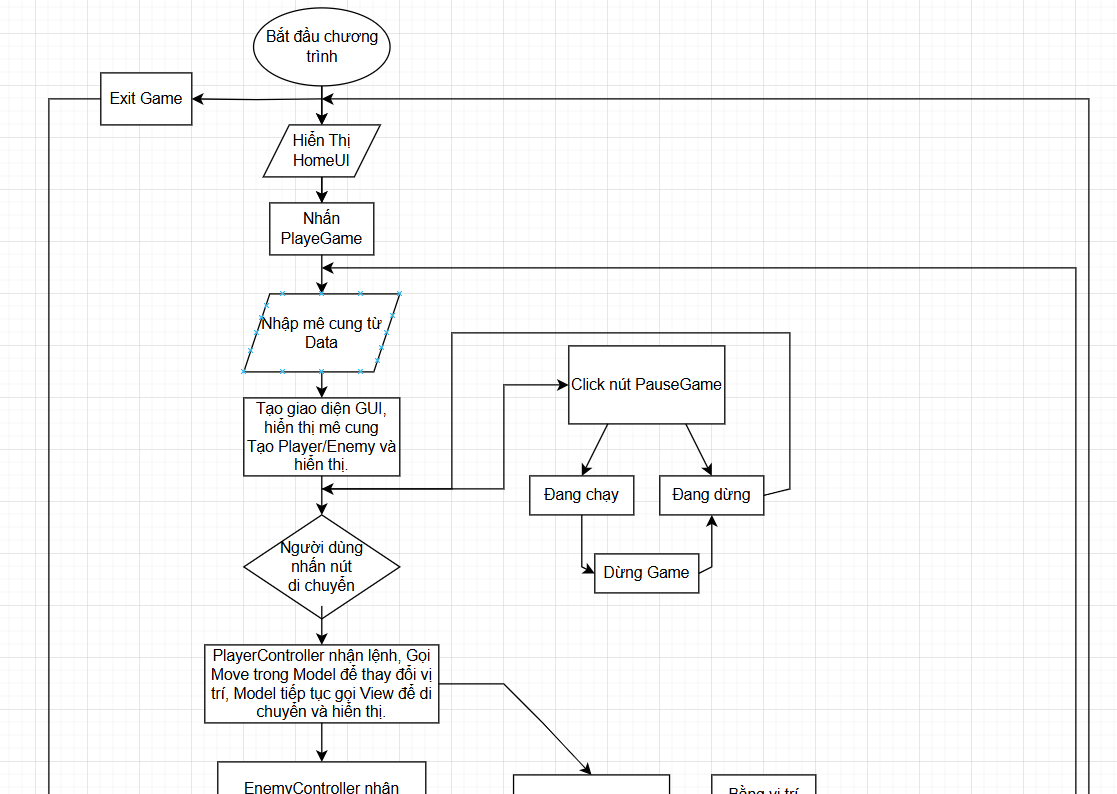
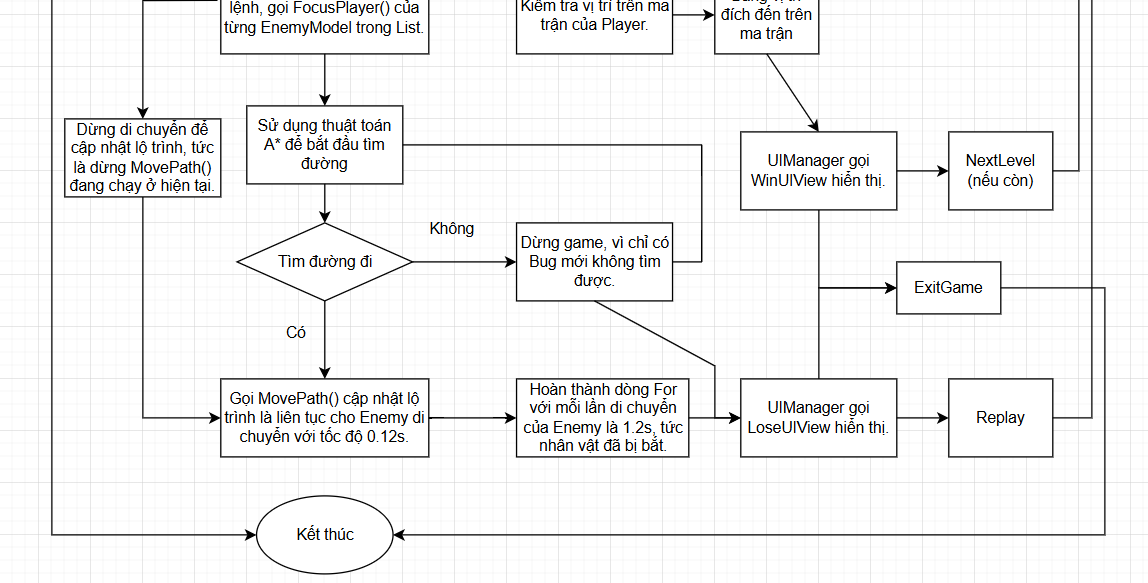
# 6. Tổng kết

Dự án “AStar Game” giúp thực hành nhiều kỹ năng lập trình game trong Unity:  
- MVC: Tách logic rõ rang.  
- Singleton: Quản lý global dễ dàng.  
- A\* Algorithm: AI đuổi theo thông minh.  
- DOTween: Chuyển động mượt.  
- Matrix Map: Xử lý đường đi bằng tính toán ma trận, phù hợp với thuật toán Astar.

# Mở rộng trong tương lai

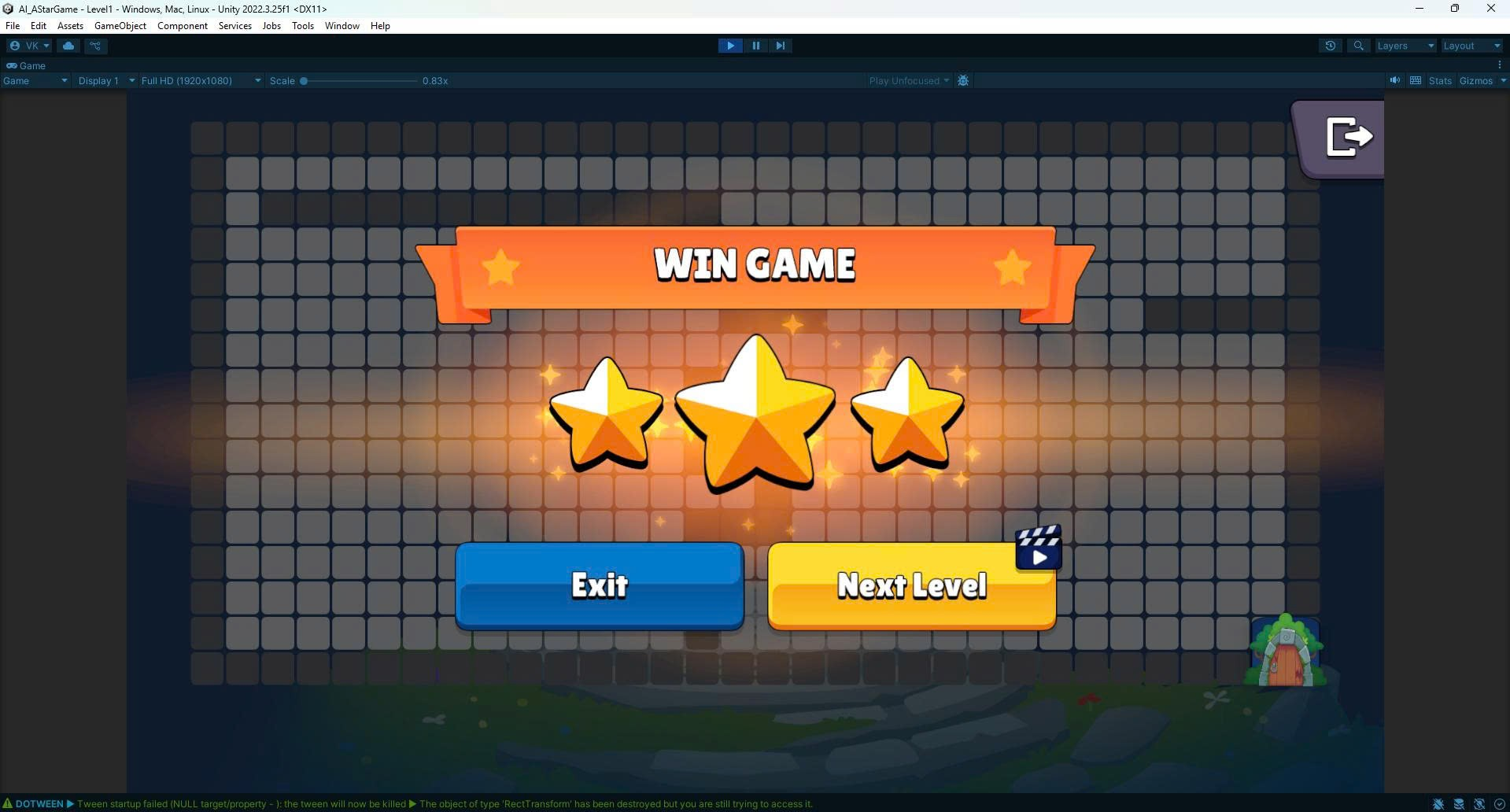
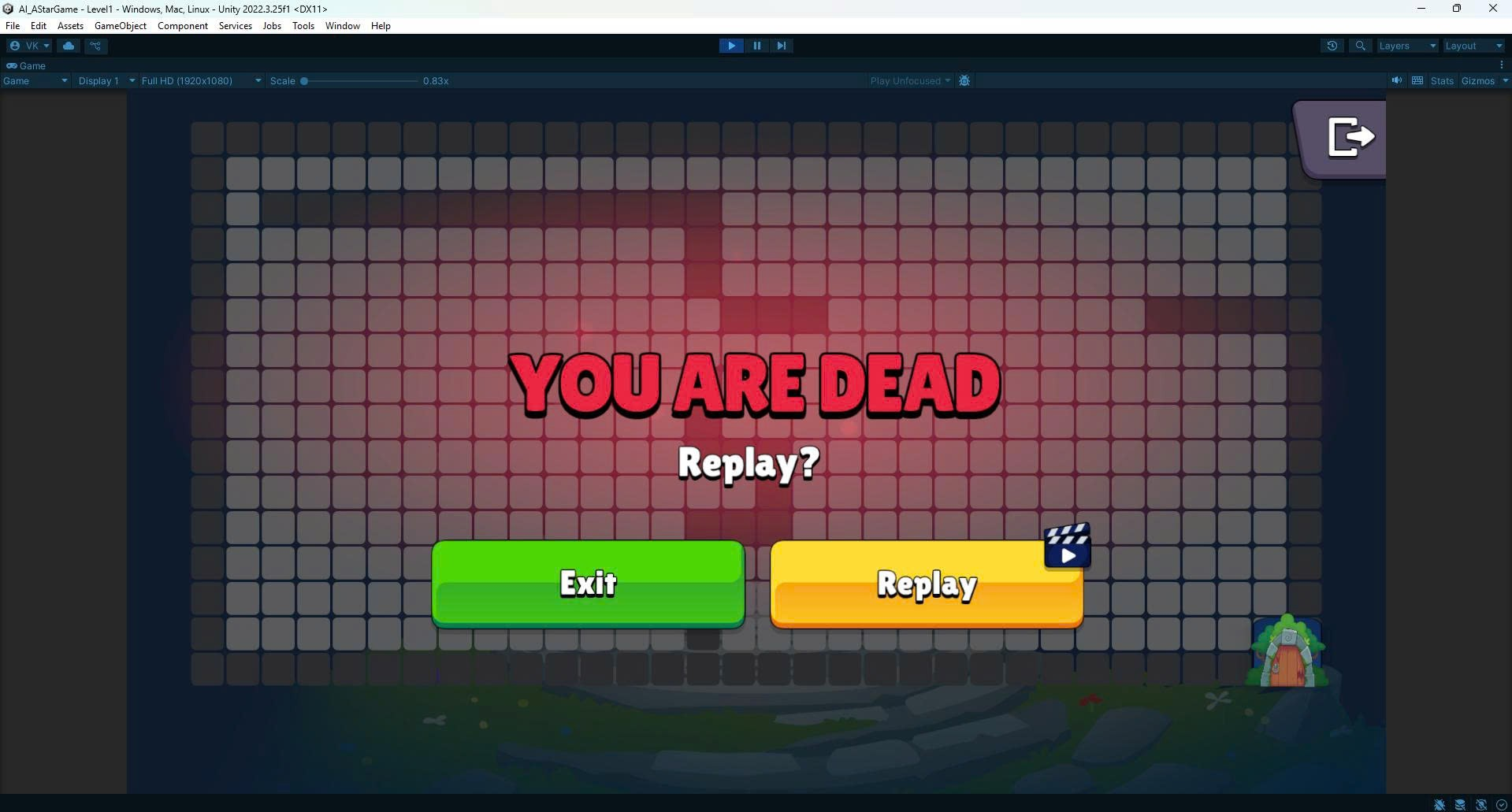
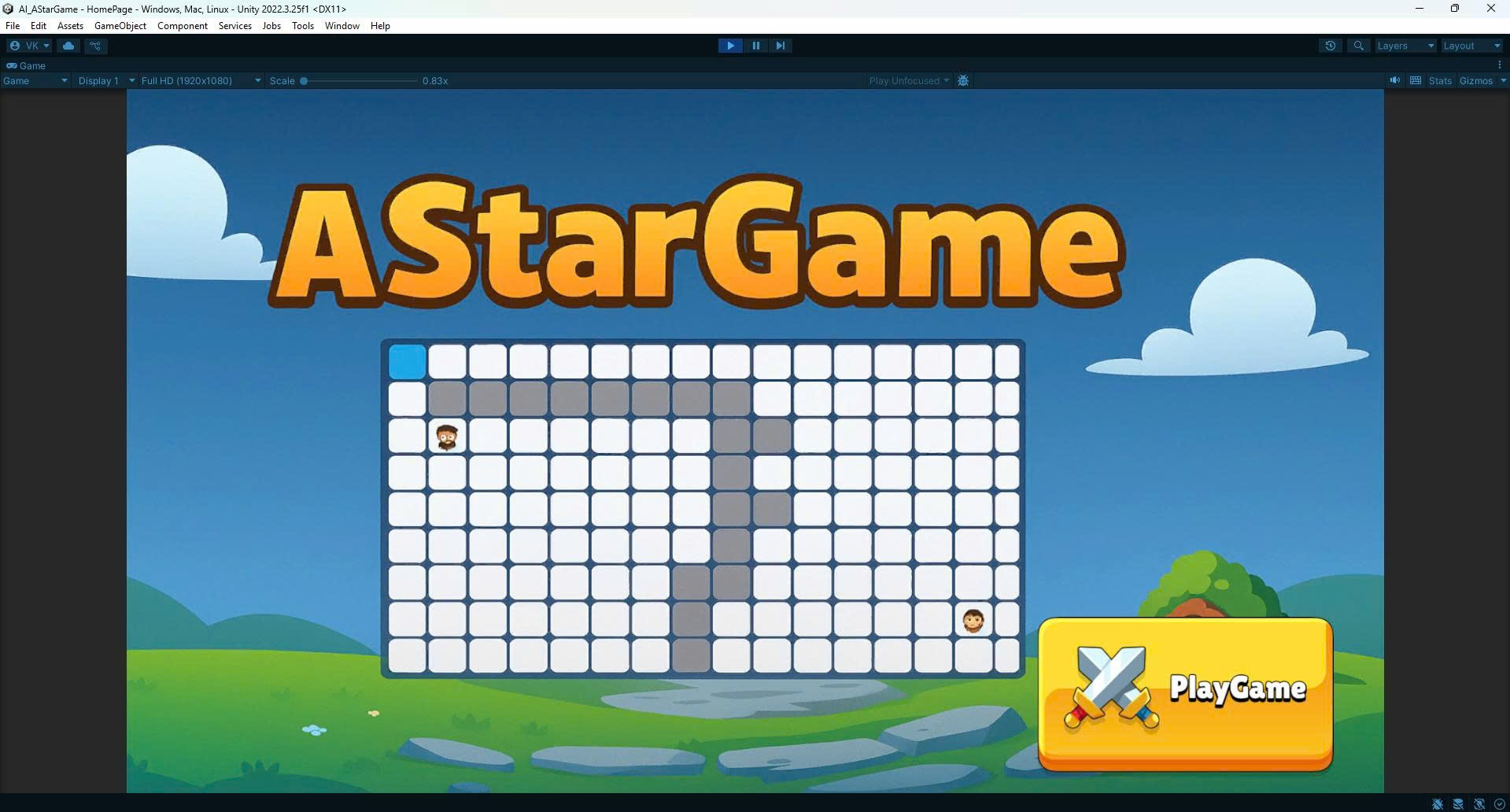
- Thêm nhiều Enemy.  
- Enemy có tốc độ/AI khác nhau.  
- Nâng cấp bản đồ random mỗi level.  
- Tích hợp minimap hoặc hiệu ứng chase camera.

# Lưu đồ thuật toán

Link: <https://app.diagrams.net/#G1muW4v9gE1i8mcyPLWayCkKVGqFPgU5P0#%7B%22pageId%22%3A%22CD5OZfdSSPRqlcloZVhb%22%7D>

# Demo và Github:

**Ảnh Demo:**   


Link Github: <https://github.com/Ptithcm2025/AI_GameAStar.git>