Tower Defence Notes : Enemy Ai

23 June 2026 10:13

Enemy AI System

Overview

The Enemy AI system is responsible for controlling enemy behaviour throughout the game. It handles movement, path navigation, combat interactions, animation updates, and communication with other gameplay systems.
One of the main goals behind this implementation was to create a reusable framework that could support multiple enemy types without requiring separate AI scripts. To achieve this, enemy behaviour and enemy configuration are kept separate. The AI controls how enemies behave, while a dedicated data asset defines their individual characteristics.

Key Features

- Data-driven enemy configuration using Scriptable Objects
- State-based behaviour management
- Multi-path waypoint navigation
- Turret detection and combat behaviour
- Event-based enemy registration
- Animation synchronization
- Runtime movement speed modification support

Architecture Overview

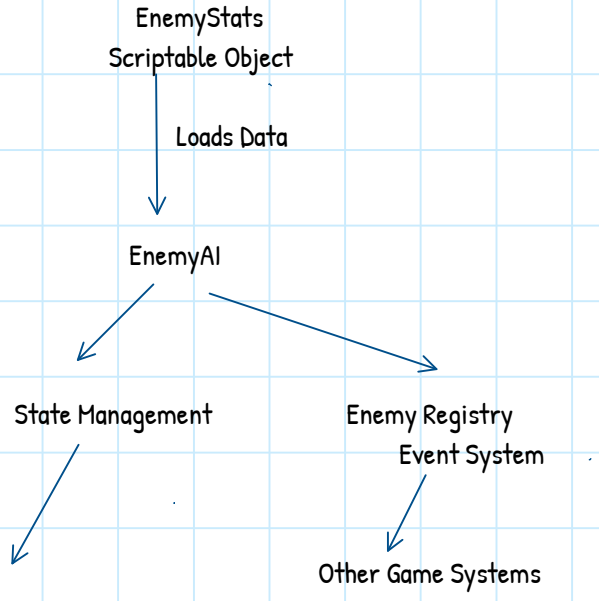
The system is built around two primary components.

EnemyStats

Stores gameplay data such as health, movement speed, damage, rewards, and attack speed.

EnemyAI

Handles runtime behaviour including movement, combat, state management, animation updates, and communication with other systems.
The following diagram illustrates how the different parts of the system interact with one another.



Walking
Attacking
Idle

- Targeting
- Wave Manager
- Analytics

Path Navigation
CheckPoints

Animation Layer

This separation keeps gameplay data, enemy behaviour, visual feedback, and external communication independent from one another.

Enemy Configuration

Enemy-specific values are stored inside a Scriptable Object called EnemyStats.

```
[CreateAssetMenu]
public class EnemyStats : ScriptableObject
{
    public int Hp;
    public float defaultMoveSpeed;
    public int baseDmg;
    public float attackPace;
    public int threatningLevel;
}
```

When an enemy is spawned, the AI loads the assigned configuration and initializes its runtime values.

```
private void OnSetInitialValue()
{
    Hp = stats.Hp;
    defaultMoveSpeed = stats.defaultMoveSpeed;
    reward = stats.reward;
    baseDmg = stats.baseDmg;
    attackPace = stats.attackPace;
}
```

By storing configuration outside the AI script, new enemy variations can be created without modifying behaviour code.

State Management

The AI uses a small state system to separate movement and combat behaviour.

```
protected enum EnemyState
{
    Idle,
    Walking,
    Attacking
}
```

The active state determines which behaviour should run during the current frame.

```
private void Update()
{
   if(enemyState == EnemyState.Attacking)
   {
      return;
   }
OnMove();
}
```
This keeps the update loop simple and prevents movement and combat from running at the same time.

Movement & Path Navigation

Enemies move using predefined waypoint paths.
When an enemy is spawned, one of the available paths is selected randomly.

```
protected void SetWay()
{
   int rnInt =
      UnityEngine.Random.Range(
      0,
      SourcewayPointStorage.wayPoints.Length);
currentSelectedPathIndex = rnInt;
   UpdateWaypointStructure();
}
```
Each enemy keeps its own checkpoint progress, allowing multiple enemies to use the same route independently.

Movement Flow

Movement itself is handled by calculating a direction toward the current checkpoint.

```
Vector3 direction =
(target.position - transform.position)
.normalized;
transform.Translate(
direction * currentMoveSpeed *
Time.deltaTime,
Space.World);
```
When a checkpoint is reached, the next checkpoint becomes the active target.

```
if(distance <= reachedDistance)
{
   selectedPath.checkPoints
   [currentSelectedCheckpointIndex]
   .tick = true;
ChangeCheckPointIndex();
}
```

Combat Behaviour

Enemies can attack player-built turrets.
When a turret enters the enemy's trigger range, the AI transitions into the Attacking state.

```
private void OnTriggerEnter(Collider other)
{
   CheckObjectTag(
      other,
```

```
      EnemyState.Attacking);
}
```

The attack loop is handled using a coroutine.

```
while(enemyState ==
    EnemyState.Attacking &&
    !turrenthealth.IsDead())
{
  turrenthealth.OnDamage(baseDmg);
yield return new WaitForSeconds(
      attackPace);
}
```

Using a coroutine keeps attack timing simple and easy to control.

Animation Integration

The animation system reacts to the current AI state.

```
private void UpdateAnimationCallBack()
{
  animationManager
  ?.SetCurrentAnimName(
     (int)enemyState);
}
```

This allows gameplay logic and visual presentation to remain separate while staying synchronized.

Enemy Registration

Each enemy registers itself when it is spawned.

```
enemyDataPacket packet =
new enemyDataPacket(
  totalEnemyCount,
  stats.threatningLevel,
  gameObject,
  enemyAi,
  enemyHealth,
  id,
  currentSelectedPathIndex,
  transform);
```

The information is then broadcast through an event.

```
onSpawnEnemy?.Invoke(packet);
```

This allows systems such as targeting, wave management, analytics, and UI tracking to receive enemy information without directly depending on the AI component.

Design Decisions

Separating Data from Behaviour

Enemy statistics are stored independently from AI logic, making balancing and content creation easier.

State-Based Logic

States provide a clear separation between movement and combat behaviour.

Event-Based Communication

Systems can exchange information without creating unnecessary dependencies.

Independent Path Tracking

Each enemy maintains its own navigation progress while sharing the same route structure.

Future Improvements

Potential future additions include:

- Status effects such as slow, stun, or burn.
- Additional enemy states.
- Dynamic path selection.
- Boss-specific behaviour.
- Formation movement.
- Advanced decision-making systems.

Conclusion

The Enemy AI system provides a flexible foundation for enemy behaviour within the project. By separating configuration, movement, combat, animation, and communication into distinct responsibilities, the system remains easy to understand, maintain, and extend.

The result is a reusable framework capable of supporting multiple enemy variations while keeping the overall implementation clean and practical for future development.