WYŻSZA SZKOŁA
INFORMATYKI i ZARZĄDZANIA
z siedzibą w Rzeszowie

LABORATORY

Artificial Intelligence in Games

Date:

- 13.05.2025

Student:

- Necati Sarper Bakır w68754

Teacher:

- mgr inż. Piotr Synoś

WYŻSZA SZKOŁA
INFORMATYKI i ZARZĄDZANIA
z siedzibą w Rzeszowie

1) SUBJECTS

1.1 PATHFINDER

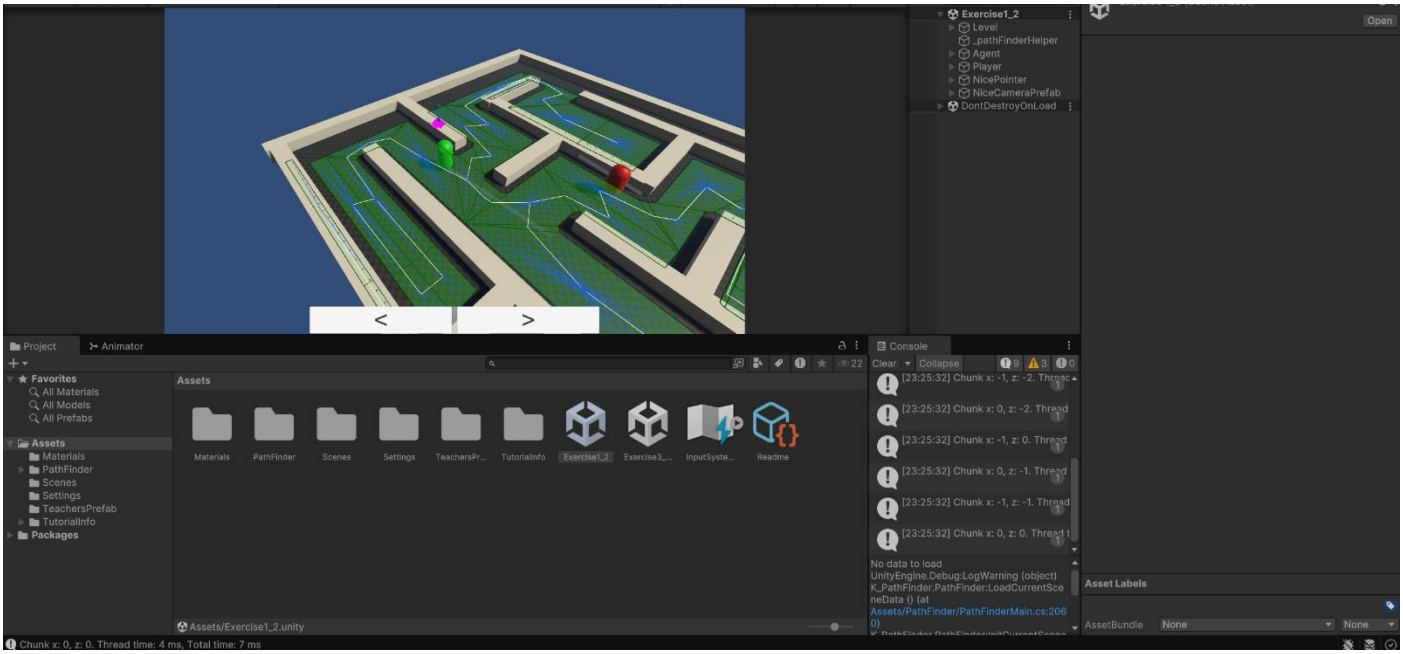
This documentation summarizes the development of basic and advanced enemy AI behaviors using Unity, focusing on patrol and pathfinding systems. The work is based on two laboratory exercises designed to teach the use of Unity's NavMesh system and C# scripting to simulate intelligent agent movement. These sessions cover both foundational mechanics like navigation and waypoints, and advanced behaviors such as player interaction and state management. The purpose is to equip developers with the tools and patterns needed to build dynamic and responsive enemy AI for game environments.

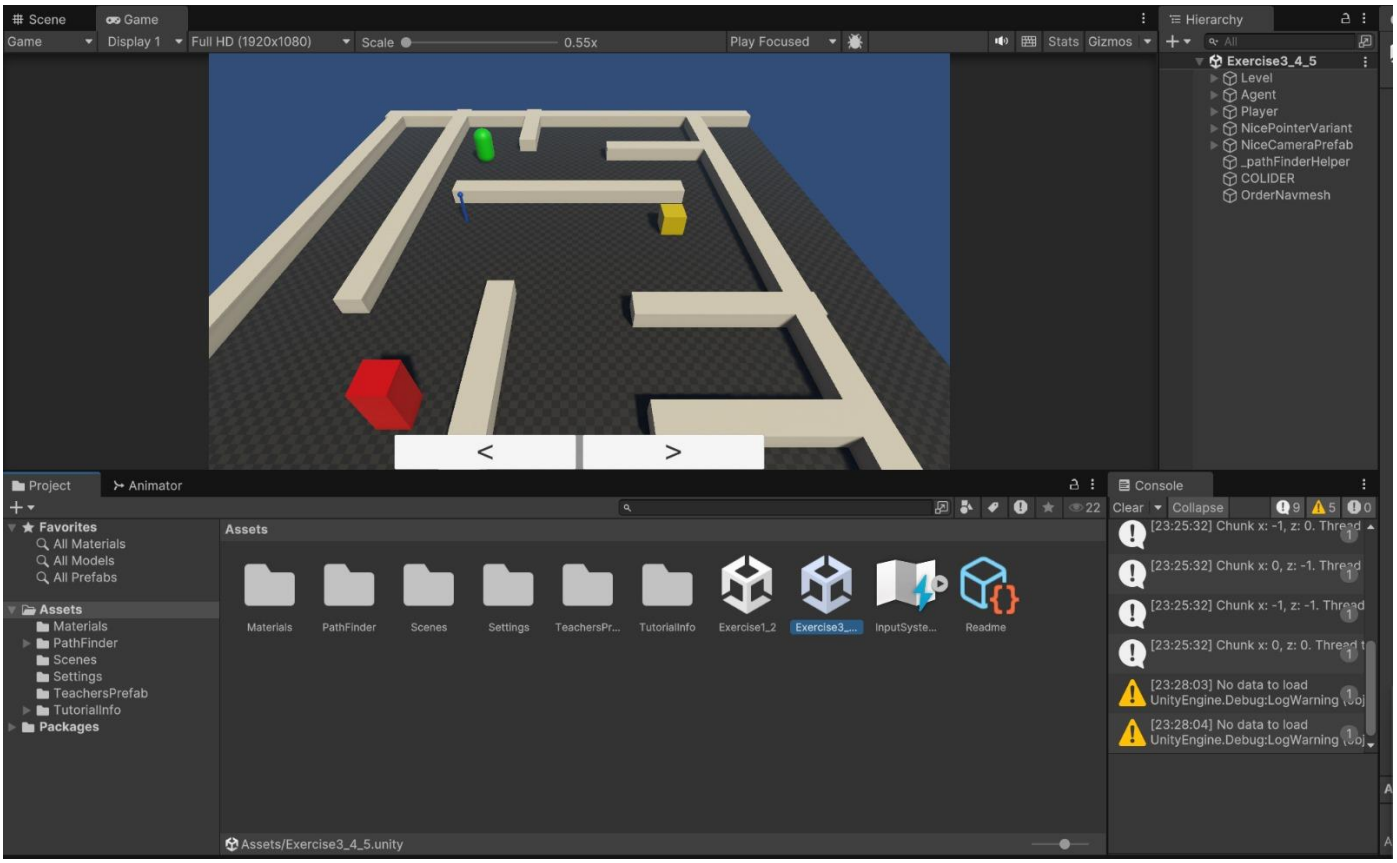
In the first part of the lab, the core of AI movement is established through two main scripts: `PathFinderAgent.cs` and `SimplePatrolPath.cs`. The `PathFinderAgent.cs` script handles agent movement along a predefined path using Unity's `NavMeshAgent`, while `SimplePatrolPath.cs` defines a series of waypoints that the enemy will patrol. The patrol behavior includes options to loop or reverse the path, control wait times at waypoints, and adjust movement speed. This modular approach allows designers to modify patrol patterns without altering the movement logic directly.

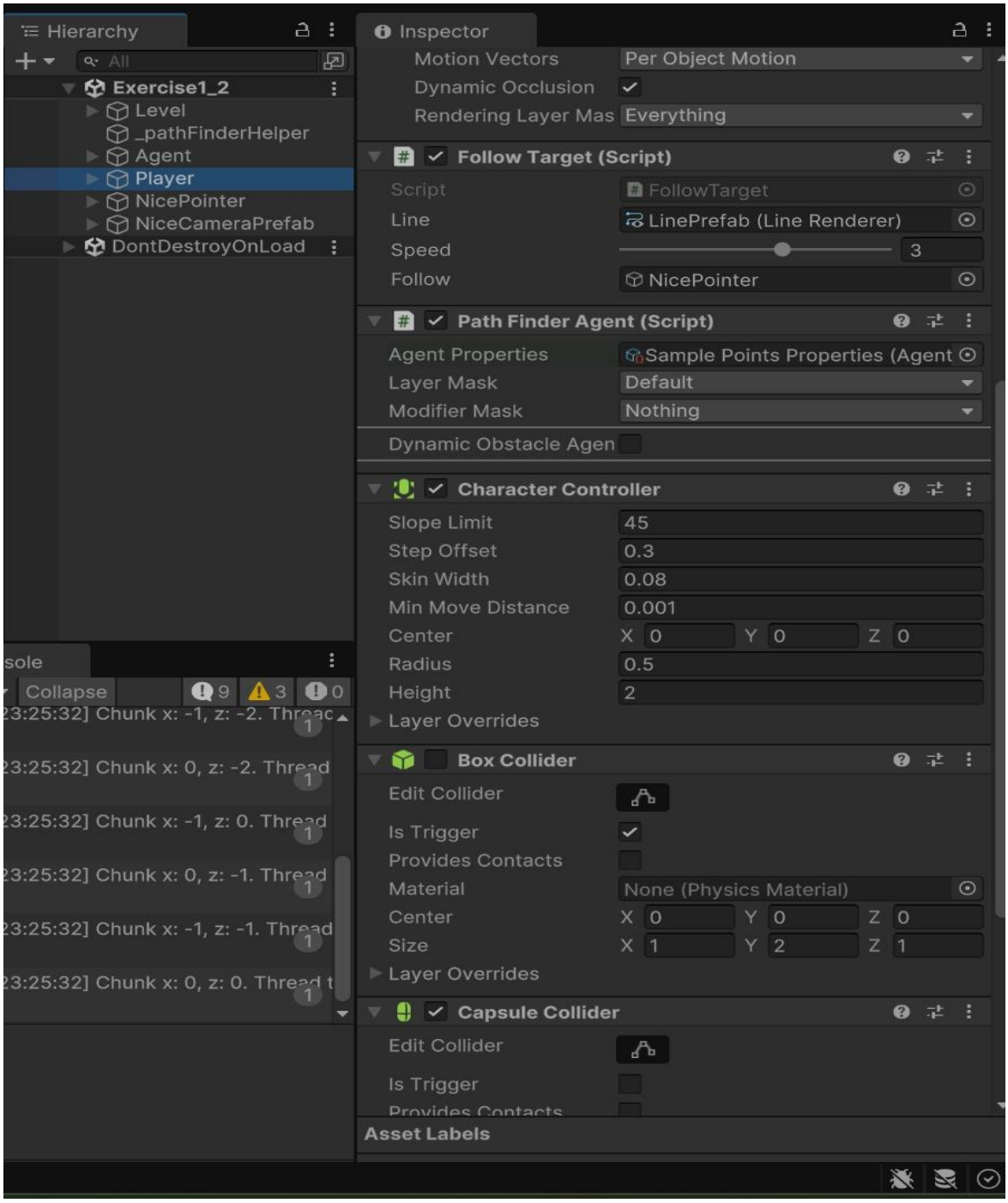
The design also emphasizes practical Unity features such as baking the navigation mesh, serializing variables for Inspector control, and managing behaviors using `Start()` and `Update()` functions. These scripts are built for scalability, allowing designers to assign different paths or modify the patrol strategy easily. In practice, the enemy agent moves smoothly from one waypoint to another and handles conditions like reaching the final point or pausing before changing direction, making it suitable for guard or patrol scenarios in a stealth game.

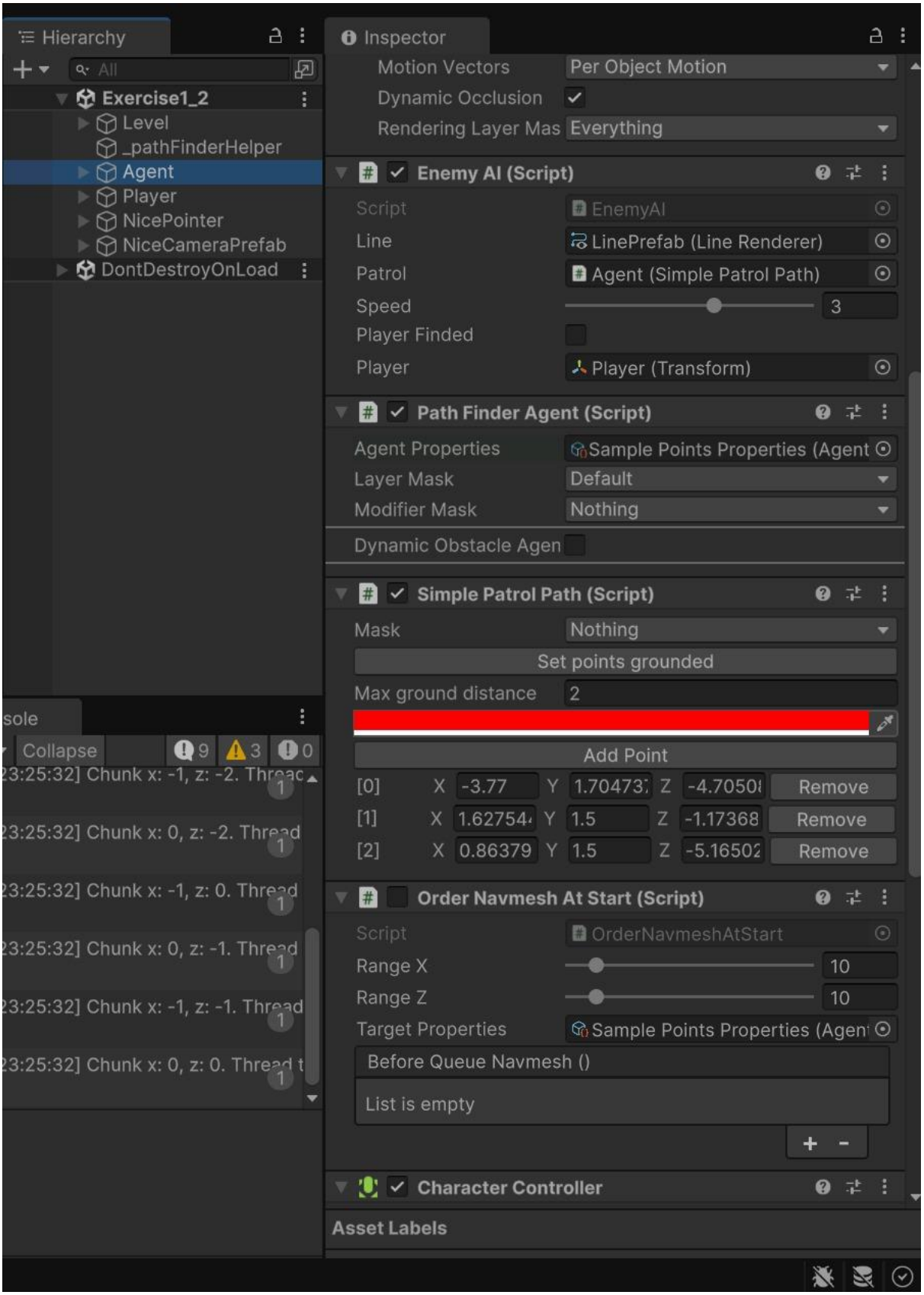
The second part of the lab introduces more sophisticated behaviors by integrating collision detection and scene-specific prefabs. The `OnCollisionEnter()` function detects contact with the player character and allows for triggering responses such as chasing or attacking. Developers are encouraged to build a basic state machine using enumerations like `Patrolling`, `Chasing`, and `Returning` to control enemy logic based on game conditions. These states help in organizing the AI's reactions, making it more intelligent and believable within the game world.

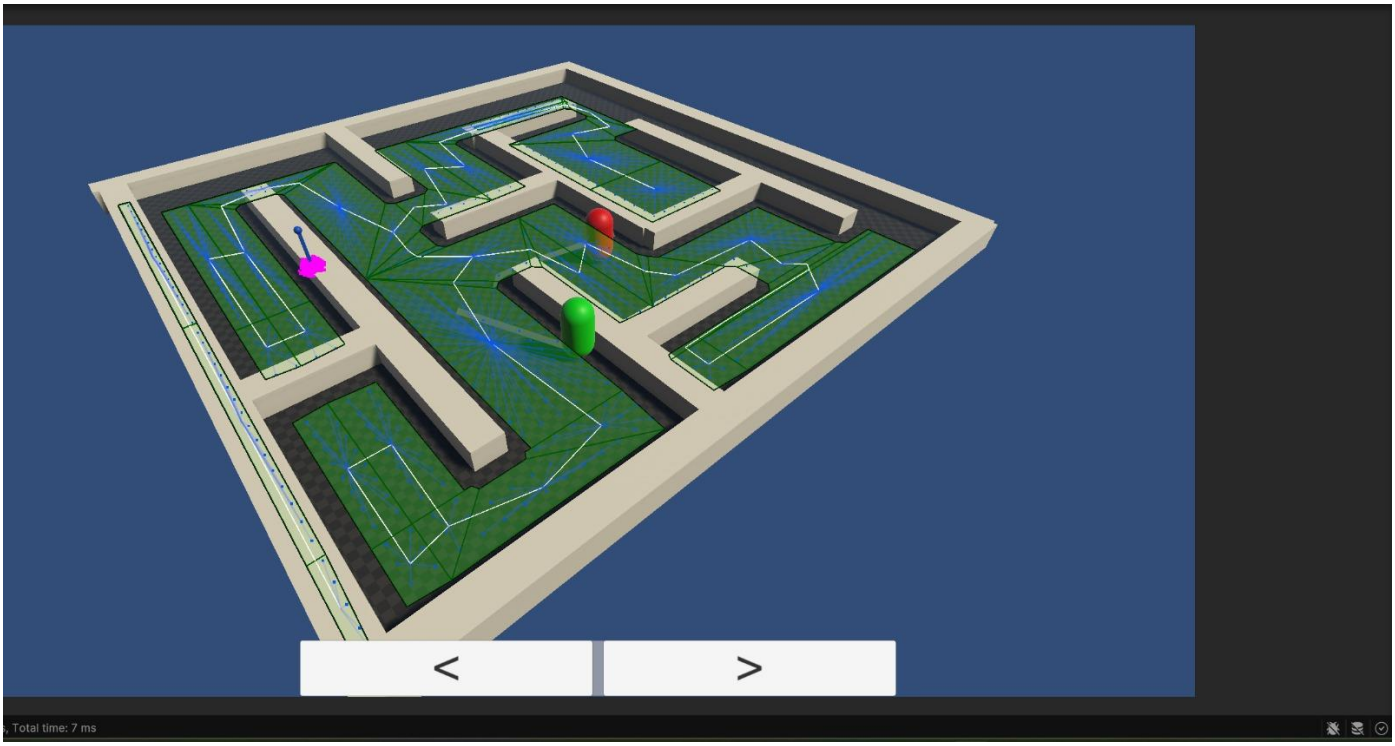
Finally, the use of prefabs like `OrderNavmeshAtStart` simplifies scene setup by ensuring that the navigation system is correctly initialized. By combining reusable components and clear scripting practices, this system forms a strong foundation for developing more advanced AI features in future iterations. Overall, this approach encourages modular development and separation of concerns, making AI behaviors easier to maintain, expand, and integrate into larger game systems.


Exercise1_2
Level
_pathFinderHelper
Agent
Player
NicePointer
NiceCameraPrefab
DontDestroyOnLoad
Open
<
>
Project
Animator
Console
Favorites
All Materials
All Models
All Prefabs
Assets
Materials
PathFinder
Scenes
Settings
TeachersPrefab
TutorialInfo
Packages
Readme
Clear
Collapse
[23:25:32] Chunk x: -1, z: -2. Thread
[23:25:32] Chunk x: 0, z: -2. Thread
[23:25:32] Chunk x: -1, z: 0. Thread
[23:25:32] Chunk x: 0, z: -1. Thread
[23:25:32] Chunk x: -1, z: -1. Thread
[23:25:32] Chunk x: 0, z: 0. Thread
No data to load
UnityEngine.Debug:LogWarning (object)
Asset Labels
AssetBundle
None
Assets/Exercise1_2.unity
Chunk x: 0, z: 0. Thread time: 4 ms, Total time: 7 ms


Scene
Game
Display 1
Full HD (1920x1080)
Scale
0.55x
Play Focused
Stats
Gizmos
Hierarchy
Exercise3_4_5
Level
Agent
Player
NicePointerVariant
NiceCameraPrefab
_pathFinderHelper
COLIDER
OrderNavmesh
<
>
Project
Animator
Console
Favorites
All Materials
All Models
All Prefabs
Assets
Materials
PathFinder
Scenes
Settings
TeachersPrefab
TutorialInfo
Packages
Exercise1_2
Exercise3_...
InputSyste...
Readme
Clear
Collapse
[23:25:32] Chunk x: -1, z: 0. Thread
[23:25:32] Chunk x: 0, z: -1. Thread
[23:25:32] Chunk x: -1, z: -1. Thread
[23:25:32] Chunk x: 0, z: 0. Thread
[23:28:03] No data to load
UnityEngine.Debug:LogWarning
[23:28:04] No data to load
UnityEngine.Debug:LogWarning
Assets/Exercise3_4_5.unity


Hierarchy
All
Exercise1_2
Level
_pathFinderHelper
Agent
Player
NicePointer
NiceCameraPrefab
DontDestroyOnLoad
sole
Collapse
9
3
0
23:25:32] Chunk x: -1, z: -2. Threac
23:25:32] Chunk x: 0, z: -2. Thread
23:25:32] Chunk x: -1, z: 0. Thread
23:25:32] Chunk x: 0, z: -1. Thread
23:25:32] Chunk x: -1, z: -1. Thread
23:25:32] Chunk x: 0, z: 0. Thread
Inspector
Motion Vectors
Per Object Motion
Dynamic Occlusion
Rendering Layer Mas
Everything
Follow Target (Script)
Script
FollowTarget
Line
LinePrefab (Line Renderer)
Speed
3
Follow
NicePointer
Path Finder Agent (Script)
Agent Properties
Sample Points Properties (Agent
Layer Mask
Default
Modifier Mask
Nothing
Dynamic Obstacle Agen
Character Controller
Slope Limit
45
Step Offset
0.3
Skin Width
0.08
Min Move Distance
0.001
Center
X 0 Y 0 Z 0
Radius
0.5
Height
2
Layer Overrides
Box Collider
Edit Collider
Is Trigger
Provides Contacts
Material
None (Physics Material)
Center
X 0 Y 0 Z 0
Size
X 1 Y 2 Z 1
Layer Overrides
Capsule Collider
Edit Collider
Is Trigger
Asset Labels


Hierarchy
All
Exercise1_2
Level
_pathFinderHelper
Agent
Player
NicePointer
NiceCameraPrefab
DontDestroyOnLoad
sole
Collapse
9
3
0
23:25:32] Chunk x: -1, z: -2. Thread
23:25:32] Chunk x: 0, z: -2. Thread
23:25:32] Chunk x: -1, z: 0. Thread
23:25:32] Chunk x: 0, z: -1. Thread
23:25:32] Chunk x: -1, z: -1. Thread
23:25:32] Chunk x: 0, z: 0. Thread
Inspector
Motion Vectors
Per Object Motion
Dynamic Occlusion
Rendering Layer Mas
Everything
Enemy AI (Script)
Script
EnemyAI
Line
LinePrefab (Line Renderer)
Patrol
Agent (Simple Patrol Path)
Speed
3
Player Finded
Player
Player (Transform)
Path Finder Agent (Script)
Agent Properties
Sample Points Properties (Agent
Layer Mask
Default
Modifier Mask
Nothing
Dynamic Obstacle Agen
Simple Patrol Path (Script)
Mask
Nothing
Set points grounded
Max ground distance
2
Add Point
[0] X -3.77 Y 1.70473 Z -4.7050 Remove
[1] X 1.62754 Y 1.5 Z -1.17368 Remove
[2] X 0.86379 Y 1.5 Z -5.16502 Remove
Order Navmesh At Start (Script)
Script
OrderNavmeshAtStart
Range X
10
Range Z
10
Target Properties
Sample Points Properties (Agen
Before Queue Navmesh ()
List is empty
Character Controller
Asset Labels



1.2 SWORN

AI-Based Player-Following Flocking Behavior – Detailed Documentation

This system offers a straightforward yet effective method for developers aiming to implement AI-driven flocking behavior in the Unity game engine. Based on the information presented in the referenced video, it draws inspiration from the classical Boids algorithm, enabling agents within a flock to move cohesively while also following a player. This behavior can be applied in games to simulate natural and interactive movement for enemy groups, friendly units, or animal herds.

Detailed Project Setup

The first step involves setting up the Unity scene, including the camera, the player object, and several test objects. A prefab called “Agent” is then created to represent each individual entity within the flock. This prefab includes a 3D object (such as a sphere or capsule), a Rigidbody component, and the necessary scripts.

Core AI Behaviors and Logic

The behavior of each agent is governed by three primary rules derived from the classical flocking algorithm:

Separation:
Agents attempt to maintain a certain distance from one another to avoid crowding and collisions.

Each agent evaluates its proximity to neighboring agents, and if the distance falls below a defined threshold, a repelling vector is calculated and applied in the opposite direction.

Alignment:
Agents align their movement direction with that of their nearby neighbors, promoting smooth and cohesive group movement. This involves calculating the average heading of nearby agents and gradually adjusting the agent's own direction accordingly.

Cohesion:
Agents move towards the average position of nearby flockmates, helping maintain the overall integrity of the group. This behavior prevents agents from drifting too far from the flock.

Player Tracking:
As an enhancement to the base flocking algorithm, each agent is also influenced by the player's position. This additional steering force ensures the flock not only moves cohesively but also orients itself toward the player, creating dynamic, target-aware movement.

An agent's final movement vector is computed as a weighted sum of the four forces: separation, alignment, cohesion, and player tracking. By adjusting the weights, developers can fine-tune the behavior to create tighter, looser, or more aggressive flock dynamics.

Coding Details

The system is implemented in C#. It consists of a `FlockAgent` class attached to each agent prefab and a `FlockManager` class that governs the entire flock. The `FlockManager` keeps track of all agents in the scene and calculates behavior vectors by identifying each agent's neighbors. Neighbor detection is performed using `Physics.OverlapSphere`.

Each agent is moved using its own Rigidbody component. Velocity and direction are computed as vectors and applied through `rigidbody.velocity`.

Performance and Optimization Considerations

As the flock size increases, so does computational load. In the video, Stephen Barr mentions that large numbers of agents can impact performance. To mitigate this, the radius used for neighbor detection should be carefully chosen, and unnecessary calculations should be avoided. For advanced optimization, techniques such as spatial partitioning (e.g., using octrees or grid-based systems) are recommended.

Use Cases

This system can be applied in various game scenarios, such as enemy swarms in action games, fish or bird flocks in simulation games, unit formations in strategy games, or aggressive AI groups in survival games.

Conclusion

This system provides a comprehensive foundation for developers looking to design real-time AI-driven flocking behaviors in Unity. By turning the foundational flocking algorithm into a practical implementation, it enables the creation of dynamic and natural-looking group movements in games. It is especially useful for those new to AI systems and for developers seeking to bring more organic behavior to their game environments.

1.3 MACHINE LEARNING

Fundamentals of Machine Learning – Summary Document

What is Machine Learning?

Machine learning is a subset of artificial intelligence that enables computers to learn from data without being explicitly programmed. Unlike traditional software systems, where logic is manually coded, machine learning systems build models from data to make predictions or decisions. These models identify patterns and relationships within the data, allowing them to generalize and make informed decisions on new, unseen data.

Types of Machine Learning

Machine learning is typically categorized into three main types:

Supervised Learning: In this approach, the algorithm is trained on labeled data—input data paired with the correct output. The goal is to learn a mapping from inputs to outputs. Common applications include email spam detection, credit scoring, and medical diagnosis.

Unsupervised Learning: This method deals with unlabeled data. The system attempts to uncover hidden patterns or groupings without predefined categories. It is often used in market segmentation and anomaly detection.

Reinforcement Learning: Involves an agent that learns to make decisions by interacting with an environment and receiving feedback through rewards or penalties. It is widely applied in robotics, game playing, and autonomous systems.

Regression

Regression is a supervised learning technique used to predict continuous numerical values. For example, predicting house prices based on features like size, location, and number of rooms. Linear regression is the most basic form, but more complex models like decision trees and neural networks can also be used for regression tasks.

Binary Classification

Binary classification refers to tasks where the output variable has only two possible categories. The goal is to assign new data points to one of the two classes. For instance, determining whether an email is “spam” or “not spam,” or whether a transaction is “fraudulent” or “legitimate.”

Multiclass Classification

Multiclass classification involves problems with more than two outcome classes. The model must learn to distinguish between three or more categories. An example is image classification where a model must label images as “dog,” “cat,” or “bird.” Algorithms like decision trees, support vector machines, and neural networks can handle these tasks.

Clustering

Clustering is an unsupervised learning method used to group similar data points together. The system identifies natural clusters within data based on similarities. A common example is customer segmentation—grouping customers by purchasing behavior or demographic characteristics without predefined labels.

Deep Learning

Deep learning is a specialized branch of machine learning based on artificial neural networks with multiple layers (hence “deep”). These models are especially powerful for handling large-scale and complex data such as images, audio, and natural language. Applications include facial recognition, voice assistants, and autonomous vehicles.

Transformers

Transformers are advanced deep learning architectures that have revolutionized the field of natural language processing. Unlike traditional sequential models (like RNNs), transformers process input data in parallel and use mechanisms like attention to model relationships between words regardless of their position. Models such as BERT, GPT, and T5 are based on transformer architectures and are capable of tasks like translation, summarization, and question answering with remarkable accuracy.

GITHUB LINK: https://github.com/Sarper35