Milkid Animal Agent Artificial Intelligence Documentation

Milkid Animal AI is an all-in-one solution for realistic mood-driven animal agent behaviour in Unity.
This package uses finite state machines to mimic animalistic behaviour depending on mood, which is driven by vital and environmental factors. This asset comes with a built in pooling / spawning solution.

How it works:

Milkid Animal Agent AI uses interfaces as behaviour states. I've almost completely avoided the usage of colliders for distance checks as Unity colliders always check the entire scene. I've tested out quite a lot of different methods to reach as little as possible computational effort when using it. I've decided to completely rely on mathematical distance checks that are running in a different thread instead of using the physics engine for that. Therefore if you want your agents to attack Players, you'll have to use the TrackableObjects class - this will be explained very detailed later in this documentation. This asset also comes with a built-in pooling and spawning solution, which is easy to adjust to your likings. I highly suggest to read the entire "Setup" section of this documentation in order to set the system up correctly.

Behaviour Pattern:

To achieve realistic behaviour I had to make sure that there are no long periods in which the agent stands still and does nothing, that's how I came up with the idea of having a base cycle of behaviours which extends itself upon possibility.
Here, the base cycle is built from the wander and idle state. This means: If the agent is currently in its idle state and no other behaviour is triggered by the given conditions and a certain amount of time has passed the agent will switch into its wander state, where it remains for however long it takes to pass the given amount of randomly generated waypoints. And again, if that behaviour isn't interrupted by any other triggered behaviours , the cycle will go back to its idle state.
However, if f.E. the agent got hungry and found some food while wandering, it will exit wandering and switch to its eat state and as soon as this action is finished, it will go back to its usual base cycle.

Threaded Agent Awareness:

As mentioned earlier - I tried to stay away from excessive collider distance checks as they suck up performance. Instead I decided to make use of an observer pattern and make every agent trackable. The tracking actually stores the current position of the objects, it doesnt update itself, instead the agent updates his position in the tracking whenever it moves. The fact that I basically made up my own trackable object opened the door for multi-threading,

as you can't use unity engine variables for threading. What happens under the hood is that the agents request to look for a target, those requests get queued and then processed after each other in another thread and return either a target or null to the requesting agent.

Possible States and their corresponding actions:

MOOD STATES:	
Satisfied	The agent is currently satisfied, meaning it is not hungry, nor tired or hurt.
Tired	The agent is currently tired, if this is the case - then the option of going into a rest state is added to the base cycle and the agent will go into said rest state if no other, more important behavior is triggered.
Hungry	The agent is currently hungry, if this is the case - then the option of going into a eat state is added to the base cycle and the agent will go into said state if no other, more important behavior is triggered and the agents search for food was successful.
Hurt	The agent is currently hurt, if this is the case - then the agent will start to regenerate its health over timer.
BEHAVIOR STATES:	
Idle	The agent is currently remaining in one spot and playing it's idle animation. It will remain in this state for a set period of time or until another behavior is triggered.
Rest	The agent is currently remaining in one spot and playing it's rest animation. It will remain in this state for a set period of time or until another behavior is triggered. Its regeneration energy in this state.
Eat	The agent found food and is now remaining in one spot and playing it's eat animation. It will remain in this state for a set period of time or until another behavior is triggered. Its regenerating satisfaction in this state.
Wander	The agent is currently wandering around and playing its walking animation. It will remain in this state for however long it takes to pass a set amount of automatically generated waypoints or until another behavior is triggered. If the agent is hungry while wandering, it will look for food in this state.
Flee	The agent is currently fleeing from another object, it will return to its idle state if the other object is far away enough or after a set

	period of time.
Chase	The agent is currently chasing another object, it will return to its attack state if it is close enough to his target or return to its old position and its idle state if a set amount of time has passed.
Attack	The agent is currently close enough to a target to attack it, it will play its attack animation. This state can only be interrupted the targets distance or death.
Dead	The agent is dead. Its death animation is played once and the dead agents body remains as an active game object in the scene for a set period of time. It will get deactivated and added back to the pool to be respawned again later.

Setup:

Pooling: This asset comes with a built-in pooling system for your AI agents. In order to set it up you have to place the Pooling prefab in the scene. Then in its inspector you'll have to create as many pools as you need. Please be aware that the pool name as to be the same name as the agent type in order to work properly. Pooled object needs a prefab of your ai agent. The pool size evaluates how many of your agents will be generated and stored in the pool at runtime.

Spawner: The prefab comes with two spawners attached as child objects, if you only have one you can delete the second one, if you need more you can add as many as you need. The spawner will be in charge of spawning a set amount of game objects and respawning new once if there aren't enough active in the current scene. The spawn radius variable describes in which radius, around the spawners position, the agents will spawn. The object radius describes the distance between two spawning objects. Unwalkable layers evaluate if it is possible to spawn at a generated position - I highly suggest to have your agents in one layer and then add this layer to unwalkable, this will ensure that agents can't spawn on top of each other.

Tracking: The Tracker Prefab comes already attached with the TrackRequestManager as a child. You have to make sure that the Tracker is tagged as tracker. There are no other values you need to change.

Animator: Milkid Ai works with Unity's mechanim system. To activate your animation you simply have to set the useAnimator bool to true in the agents inspector but before it will actually work you'll have to set up an animator controller component, this needs to be added to the part of your gameobject that carries the model.

STEP - BY - STEP:

1. Create a new animator controller
2. Open it up and drag all your animations into it
3. Set up transitions between your states
4. Create parameter to trigger the transitions. The parameters have to be the following bools: “isIdle”, “isResting”, “isEating”,”isWalking”,”isRunning”, “isAttacking”,”isDead”.
5. Make sure to untick “Has Exit Time” for each transition.

Agent:

This is the actual agent script which is going to be attached to every agent. You’ll have to setup your agent here - so I’ll explain every variable in its inspector here in detail.

General	
Agent Type	Describes the type of enemy - has to be the same as the corresponding pool name.
useAnimator	Activates mechanim animations, in order to use this you’ll have to set up an animator controller - this will be explained under “Animations” in this documentation.
Mode	The mode determines how your animals will react in general. You can choose from three different modes here: AGGRESSIVE - The agent will automatically chase/attack every other object that is close enough and considered to be an enemy of the agent. DEFENSIVE - The agent will only attack/chase another object if it got attacked first. PASSIVE - The agent will never attack or chase any other object, it will always try to flee.
States	
Enable Rest	Enables the rest state and allows the agent to get tired.
Enable Eat	Enables the eat state and allows the agent to get hungry.
Enable Chase	Enables the chase state.
Enable Flee	Enables the flee state.
Enable Attack	Enables the attack state.

Enable Dead	Enables the dead state and allows the agent to be mortal.
Vitality	
Max Health	Defines the maximum amount of health value.
Health Tick Rate	Defines how many health points will get regenerated per tick.
Health Tick Time	Defines the time between health ticks.
Max Energy	Defines the maximum amount of the health value.
Energy Tick Rate	Defines how much energy gets reduced with each tick.
Energy Tick Time	Defines the time between energy ticks.
MaxSatisfaction	Defines the maximum value of satisfaction.
Hunger Tick Rate	Defines how much satisfaction gets reduced with each tick
Hunger Tick Time	Defines the time between hunger ticks.
Food	
Carnivore	Determines if the agent will eat dead players/agents.
Natural Enemies	
Enemies	Is an array of all the string types of agents/objects that are considered to be enemies.
Damage	
Max Damage	Defines the maximum Damage the agent can deal.
Min Damage	Defines the minimum Damage the agent can deal.
Aggro Time	Is only relevant for defensive agents, it defines the time for how long they will be aggressive after they have been attacked by another agent/player.
Movement	
Walkspeed	Defines the agent's speed when walking.
Runspeed	Defines the agent's speed when running.

Sight	
Sight Range	Defines how far the agent can see/be aware of other agents.
Wander	
Dynamic Wandering	Determines if the agents waypoints will be generated dynamic from its current position or static from its actual spawnpoint.
Wanderrange	Defines the radius within the agent will wander around.
Max Waypoints	Defines the maximum amount of waypoints the agent will pass while it is in its wander state.
Range	
Chaserange	Defines the maximum distance from the agent to another object for it to chase the other object.
Fleerange	Defines the maximum distance from the agent to another object for it to be fleeing from the other object.
Attackrange	Defines the maximum distance from the agent to another object for it to attack the other object.
Eatrange	Defines the maximum distance from the agent to another object for it to eat the other object.
Timing	
Idletime	Defines how long the agent can remain in its idle state at max.
Resttime	Defines how long the agent can remain in its rest state at max.
Eattime	Defines how long the agent will remain in its eat state if its not interrupted.
Chasetime	Defines how long the agent will chase another object at max before it gives up and returns to its old position.
Fleetime	Defines how long the agent will flee from another object at max.
Decaytime	Defines how long a dead agents body will remain active in the scene before being added back to the pooling system.
Debugging	
Show Logs	This activates console logs - be aware that many debug logs can drain performance, so really only use it for debugging and deactivate it in your game.

Agent Setup - STEP BY STEP:

1. Drag your agent model in the scene
2. In the hierarchy add an empty gameobject and drag the model onto it to make the empty game object the parent. Now, if your model happens to have messed up axis you can simply adjust its rotation now - this makes sure that the agent won't walk backwards.
3. Add a navmeshagent component to the parent game object.
4. Add a rigidbody component, that is set to be kinematic to the parent gameobject
5. Add a animator controller component to the actual model
6. Add the Agent script to the model
7. Add a convex mesh collider to your model to prohibit ghost walking.
   Tipp: For most models you have to attach the collider to the first child of the model. You'll see the collider showing up in a wrong rotation if you are trying to attach it to the wrong part.
8. Attach a simply shaped collider to your actual model. We will use this to detect damage input. We are using a simply shaped collider to reduce the computational effort. This collider should be set to trigger.
9. Open your models rigging in the hierarchy and ad spherical colliders to the parts of your model that deal damage. It is necessary to attach them to the bones in order to allow them to move with the animation.
10. Attach the Damage script to those gameobjects and tag them "Damage Output"