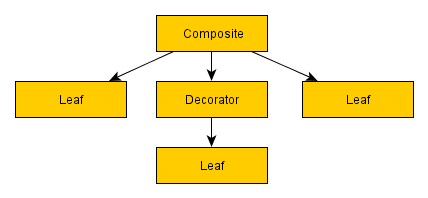
Task 1 – Behaviour Trees

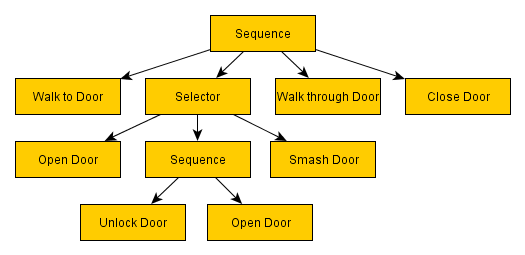
# What are Behaviour Trees?

Behaviour trees define a way to intelligently control characters in video games. Behaviour trees are simple to define; they are scalable to exhibit complex AI and modular for reusability. This type of AI behaviour is goal oriented with each tree attempting to achieve multiple high level goals. Behaviour trees can also be linked together allowing for the implementation of complex behaviours by first defining smaller sub-behaviours. Behaviour trees are similar to hierarchical state machines, but the main difference is that the building blocks for behaviours use tasks rather than states.

A key feature of a behaviour tree is its flow; the flow is the core functionality of the tree. Every node of the tree can return one of three statuses; Success, Failure or Running. An example could be an NPC’s task is to walk to a set location, it would return success if the NPC’s position is equal to the destined locations position, it would return failure if the pathfinding failed or the if the NPC is stopped on the walk to the set location and it would return running if the NPC is currently carrying out the task. Other nodes will wait for this task to return success or failure before they can be carried out.

A behaviour tree diagram consists of three main examples of nodes which are composite, decorator and leaf. A composite node is usually a high priority in the hierarchy of the diagram this is because it can have one or more children, they will process one or more child node in sequence or at random. A composite nodes status is determined by its child’s statuses; for example whilst the child’s tasks are being executed the composite node will return running. A decorator node is like a composite node but instead can only have one child node. The decorator nodes function is either to transform their child’s status (inverter), to terminate the child (succeeder) or to iterate the child node (repeater or repeat until fail). A use for the decorator node is to invert the child’s status like a NOT logic gate. The final node example is a leaf which cannot have a child and is the lowest level node type in a diagram. Leaf nodes are the most powerful nodes because they are implemented into the video game as an AI action.

In relation to game developers behaviour trees are rather popular, they’re easy to understand and are less error prone. There is also a straight translation that can be drawn between a behaviour tree diagram and code for a game; this is because composite and decorator nodes can be demonstrated as functions, language constructs such as if statements can define the flow of your code and leaf nodes act as function calls that allow the AI character to perform a task.

An example of a behaviour tree is illustrated below; this diagram demonstrates the actions an AI character would need to make when trying to pass through a door. 

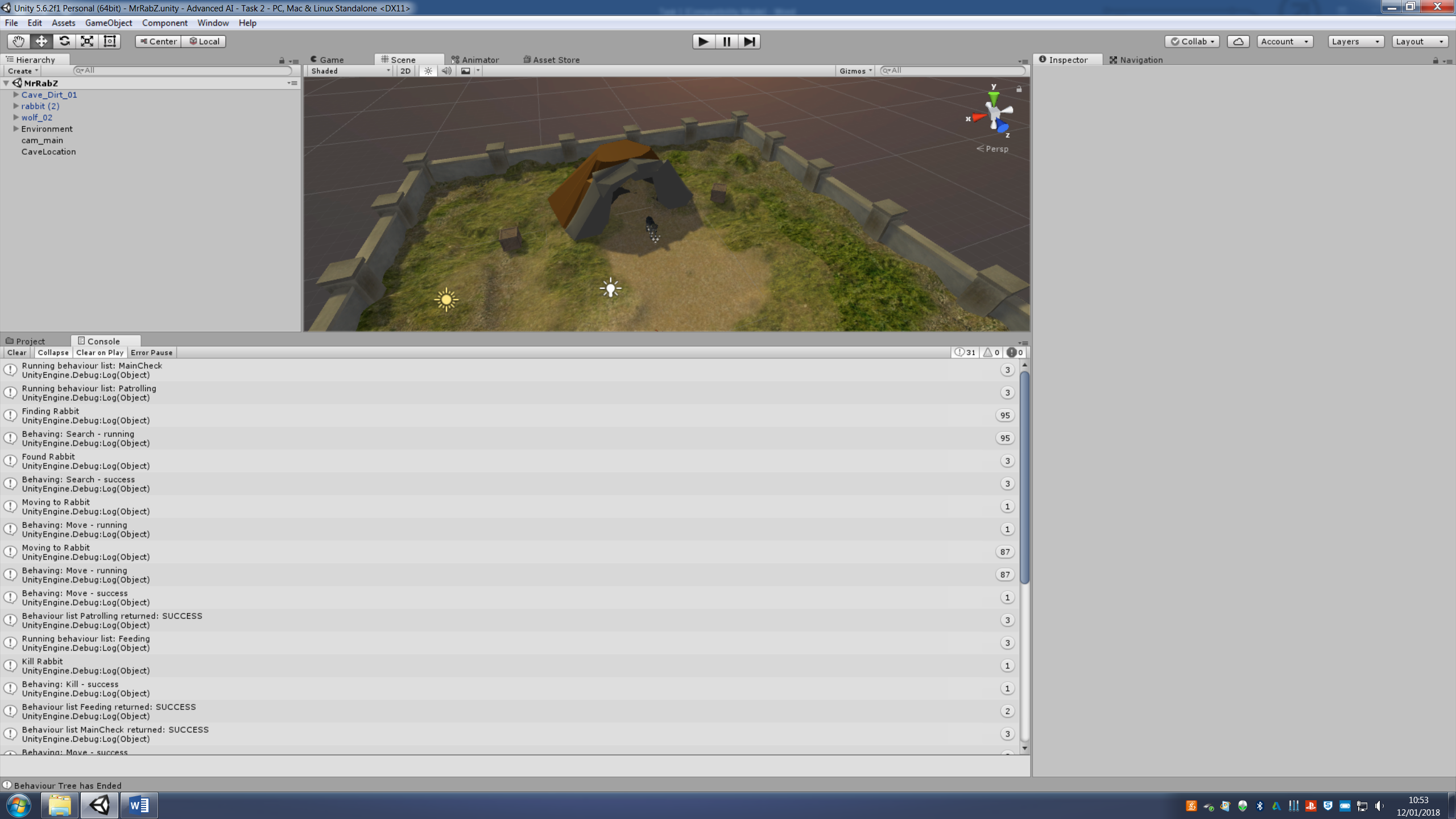
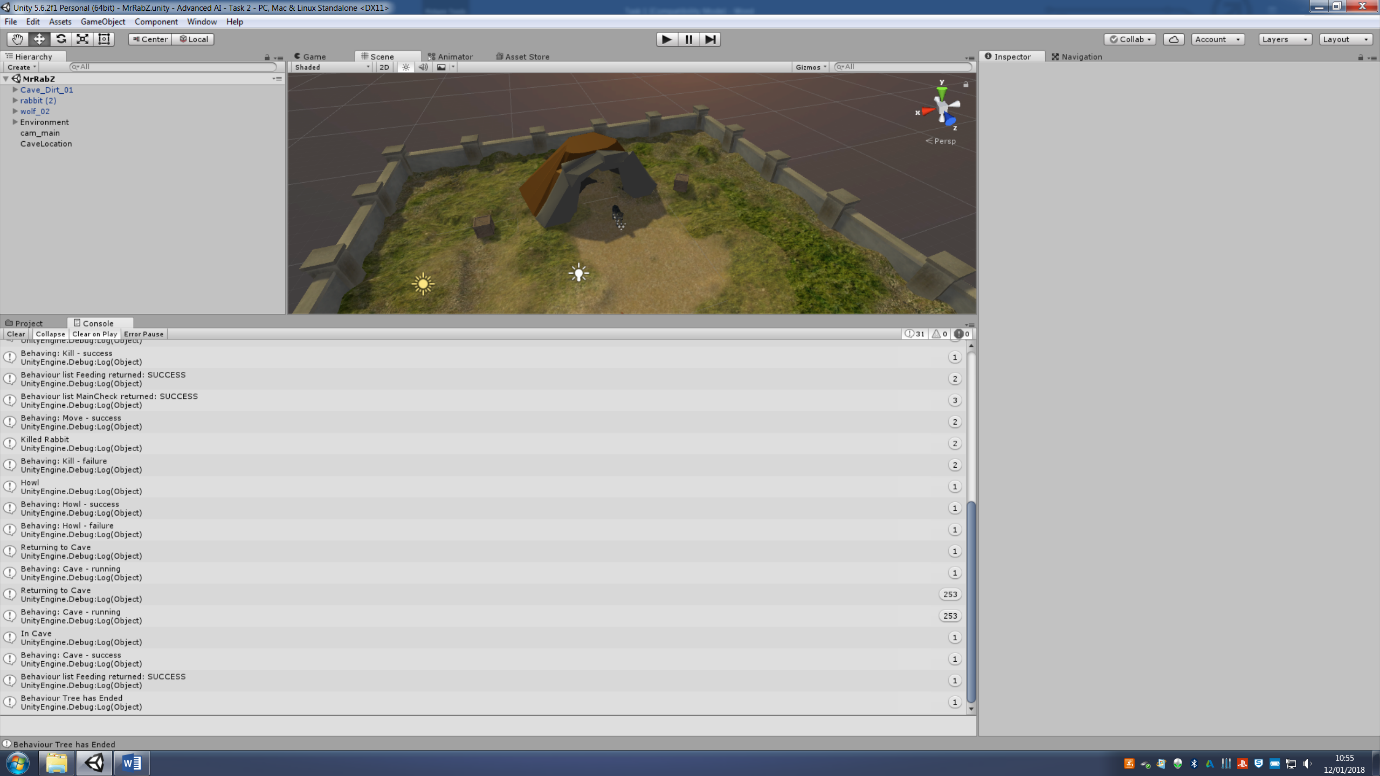
Analysing the diagram you can see a range of leaf nodes (AI tasks) but also sequence and selector nodes. A sequence node is another representation for an AND logic gate, all children of the sequence must return a success in order for the parents status to also be a success, if just one child returns fail then the sequence node will also return fail. A sequence node will visit each child in order from left to right, a child must be ticked as a success or failure before the sequence node can move on. A selector node is another representation for an OR logic gate, if a child’s status fails the selector node will carry on visiting the other children until it has a final status on each child. Only one child for a selector node needs to succeed for the parent to also return a success.

# Behaviour Trees Compared to HTN and GOAP

Looking at BTs, HTN and GOAP in their design you can see that they all follow a plan. All these methods look at creating a diagram; this is to help the AI decide the best approach to completing a general task, which is composed of simpler more specific tasks. These behaviours work in hierarchy allowing the tasks to be prioritised and integrate with each other as a parent or a child. But this is where you can start to see some key differences in their design, for example a GOAP design only allows a parent to have one child, the parent world state is the world state of the previously executed Action after it has applied its effects. Whereas with BTs and HTNs most parent nodes or actions can have multiple children, this allows the AI to be more diverse with their approach. Also the behaviours all work in sequence to complete a task but in different ways, for HTN the sequence of execution is based on what conditions the AI has met. For GOAP the world state uses a spanning tree to determine the most efficient sequence to complete the tasks in, whereas BTs are simpler, the sequence just reads from left to right. BTs are also more advanced in some ways because they allow the user to use a selector instead of a sequence if desired.

Looking at implementation, BTs are simple to implement, the BT diagrams allow for a straight conversion to code. But with HTN it is more complex to implement this is because there are different types of tasks to implement, as well these tasks need to communicate correctly with the world state. Implementing HTN is very similar to GOAP, except GOAP uses a more complex world state in using atoms to guess how close the Node is to the goal.

Moving onto application of AI behaviours, the best scenarios for a BT to be used would be when an AI is performing a general task with a few variables to consider, whereas a scenario for HTN is applied like a BT but when the NPC needs to reason about its approach. GOAP is best applicable when there are a lot of pathways to take in order to reach a certain goal. In regards to these behaviours performance GOAP would be the most efficient because of its ability to find the quickest route to achieve a goal. HTN on the other hand would perform better when there are multiple AI’s using the same behaviour because of the way the world state works allowing for fewer resources to be required. Finally behaviour trees perform best when an NPC is left in a unique situation; this allows developers a lot more freedom to create a better playing experience for the user.

Task 2 – Screenshots

Here are two screenshots of the console within Unity. In this console it displays each selector, sequence and leaf. For each node it will display whether the action has returned a success, failure or is currently running.

Task 2 – Testing

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected Outcome** | **Actual Outcome** |
| Patrolling | In the sequence node it is expected to return successful after the wolf has searched and moved to a rabbit using a perception system. | The sequence node returns successful after both leaves return successful but not using the perception system correctly. |
| Feeding | In this selector node it should at least return one leaf as successful in each BT tick. | It returns successful or running at least once in each tick. |
| Search | Uses a perception system to locate and face towards a rabbit. | Wolf faces towards player and uses senses but line isn’t drawn to the rabbit. |
| Move | Wolf moves towards located rabbit. | The wolf did move towards the rabbit’s location but not with an offset. |
| Kill | The wolf destroys the rabbit’s game object when they’re positions equal the same. | This test met its expected outcome. |
| Howl | When there are no rabbits left to kill then the wolf howls once. | Once kill returns failure then the howl leaf begins and will only howl once based on the howl count. |
| Return to cave | Once the wolf has finished howling he then will move back to his cave and then the behaviour tree will end. | This test met its expected outcome. |

Task 2 – Scripts

# BTController

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.AI;

public class BTController : MonoBehaviour {

Node behaviourTree;

Context behaviourState;

private BTController btcontroller;

public Transform Rabbit1;

public Transform cave;

public GameObject Rabbit01;

public float speed = 5.0f;

public float move;

public float frontRange = 10.0f;

public float frontAngle = 30.0f; // if angle between forward and los is less than this, the target is at front

public float closeRange = 2.0f;

void Awake()

{

behaviourTree = CreateBehaviourTree ();

behaviourState = new Context ();

move = speed \* Time.deltaTime;

}

void Start()

{

btcontroller = GetComponent<BTController> ();

}

public void IsPlayerWithinPerceptionRange()

{

RaycastHit hit;

if(Physics.Linecast(transform.position, Rabbit1.position, out hit))

{

if(hit.transform == Rabbit1.transform)

{

if(hit.distance <= closeRange)

{

Debug.DrawLine(transform.position, hit.transform.position, Color.black);

Rabbit1 = hit.transform.gameObject.GetComponent<BTController>().transform;

Rabbit01 = hit.transform.gameObject;

}

else if(hit.distance <= frontRange)

{

if(Vector3.Angle(transform.forward,hit.transform.position - transform.position) <= frontAngle)

{

Debug.DrawLine(transform.position, hit.transform.position, Color.black);

Rabbit1 = hit.transform.gameObject.GetComponent<BTController>().transform;

Rabbit01 = hit.transform.gameObject;

}

}

}

}

}

void Update()

{

behaviourState.RabbitCount = GameObject.FindGameObjectsWithTag ("Rabbit").Length;

if (Rabbit1 != null)

{

IsPlayerWithinPerceptionRange ();

// Searches for Rabbit

if (behaviourState.FindRabbit == false)

{

transform.rotation = Quaternion.RotateTowards (transform.rotation, Rabbit1.rotation, 0.4f);

}

// Moves to Rabbit

if (behaviourState.MovetoRabbit == false && behaviourState.FindRabbit == true)

{

transform.position = Vector3.MoveTowards (transform.position, Rabbit1.position, move);

}

}

if (behaviourState.RabbitCount >= 1)

{

if (transform.position == Rabbit1.position)

{

behaviourState.MovetoRabbit = true;

transform.position = transform.position;

Destroy (Rabbit01);

}

if (transform.rotation == Rabbit1.rotation)

{

behaviourState.FindRabbit = true;

}

}

// Kill Rabbit

if (behaviourState.RabbitCount == 1)

{

behaviourState.KillRabbit = true;

}

else

{

behaviourState.KillRabbit = false;

behaviourState.howl = true;

}

if (behaviourState.howlCount == 1)

{

behaviourState.howl = false;

transform.rotation = Quaternion.RotateTowards (transform.rotation, cave.rotation, 0.9f);

if (transform.rotation == cave.rotation)

{

transform.position = Vector3.MoveTowards (transform.position, cave.position, move);

}

}

if (transform.position == cave.position)

{

behaviourState.ReturntoCave = true;

}

if (behaviourState.BTEnd == true)

{

Debug.Log ("Behaviour Tree has Ended");

btcontroller.enabled = false;

}

}

void FixedUpdate()

{

behaviourTree.Behave(behaviourState);

}

Node CreateBehaviourTree()

{

Sequence Patrolling = new Sequence ("Patrolling", new Search(), new Move());

Selector Feeding = new Selector ("Feeding", new Kill(), new Howl (), new Cave ());

Sequence MainCheck = new Sequence ("MainCheck", Patrolling, Feeding);

Repeater repeater = new Repeater (MainCheck);

return repeater;

}

}

# Context

using UnityEngine;

using System.Collections;

public class Context : BehaviourState

{

public bool FindRabbit = false;

public bool MovetoRabbit = false;

public bool KillRabbit = false;

public bool howl = false;

public bool ReturntoCave = false;

public float howlCount = 0;

public float RabbitCount = 1;

public bool BTEnd = false;

}

# Search

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Search : Leaf {

public override NodeStatus OnBehave(BehaviourState state)

{

Context context = (Context)state;

if (context.FindRabbit == false)

{

Debug.Log ("Finding Rabbit");

return NodeStatus.RUNNING;

}

else if (context.FindRabbit == true)

{

Debug.Log ("Found Rabbit");

return NodeStatus.SUCCESS;

}

else

{

return NodeStatus.FAILURE;

}

}

public override void OnReset()

{

}

}

# Move

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Move : Leaf {

public override NodeStatus OnBehave(BehaviourState state)

{

Context context = (Context)state;

if (context.MovetoRabbit == false)

{

Debug.Log ("Moving to Rabbit");

return NodeStatus.RUNNING;

}

else if (context.MovetoRabbit == true)

{

return NodeStatus.SUCCESS;

}

else

{

return NodeStatus.FAILURE;

}

}

public override void OnReset()

{

}

}

# Kill

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Kill : Leaf {

public override NodeStatus OnBehave(BehaviourState state)

{

Context context = (Context)state;

if (context.KillRabbit == true)

{

Debug.Log ("Kill Rabbit");

return NodeStatus.SUCCESS;

}

else

{

Debug.Log ("Killed Rabbit");

return NodeStatus.FAILURE;

}

}

public override void OnReset()

{

}

}

# Howl

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Howl : Leaf {

public override NodeStatus OnBehave(BehaviourState state)

{

Context context = (Context)state;

if (context.howl == true)

{

Debug.Log("Howl");

context.howlCount = 1;

return NodeStatus.SUCCESS;

}

else

{

return NodeStatus.FAILURE;

}

}

public override void OnReset()

{

}

}

# Cave

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Cave : Leaf {

public override NodeStatus OnBehave(BehaviourState state)

{

Context context = (Context)state;

if (context.ReturntoCave == true)

{

Debug.Log("In Cave");

context.BTEnd = true;

return NodeStatus.SUCCESS;

}

else if (context.ReturntoCave == false)

{

Debug.Log("Returning to Cave");

return NodeStatus.RUNNING;

}

else

{

return NodeStatus.FAILURE;

}

}

public override void OnReset()

{

}

}

# PerceptionManager

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

namespace Perception

{

public class PerceptionManager : MonoBehaviour

{

private List<GameObject> Wolves;

private List<PerceptionStimulus> StimBuffer;

private List<GameObject> Rabbits;

void Awake()

{

Wolves = new List<GameObject>();

StimBuffer = new List<PerceptionStimulus>();

Rabbits = GetRabbits();

}

void Start ()

{

StartCoroutine(StimulusCheck());

}

private List<GameObject> GetRabbits()

{

GameObject[] tempAI = GameObject.FindGameObjectsWithTag("Rabbit");

List<GameObject> chars = new List<GameObject>(tempAI);

return chars;

}

public void Register(GameObject registry)

{

Wolves.Add(registry);

}

public void AcceptStimulus(PerceptionStimulus stimulus)

{

StimBuffer.Add(stimulus);

}

IEnumerator StimulusCheck()

{

yield return new WaitForSeconds(0.1f);

GenerateVisualStims();

ProcessStimBuffer();

StartCoroutine(StimulusCheck());

}

private void ProcessStimBuffer()

{

PerceptionTracker perceptionTracker;

foreach (GameObject W in Wolves)

{

perceptionTracker = W.GetComponent<PerceptionTracker>();

foreach (PerceptionStimulus s in StimBuffer)

{

if (Filter(perceptionTracker, W, s))

{

if(s.StimType == PerceptionStimulus.StimulusTypes.AudioMovement||s.StimType == PerceptionStimulus.StimulusTypes.AudioAttack)

{

float distance = Mathf.Abs((s.stimLoc - W.transform.position).magnitude);

if(distance < s.stimRadius)

{

perceptionTracker.FilteredStimulus(s);

}

}

else if( s.StimType == PerceptionStimulus.StimulusTypes.VisualCanSee && s.stimSecondary == W)

{

perceptionTracker.FilteredStimulus(s);

}

}

}

}

StimBuffer.Clear();

}

private void GenerateVisualStims()

{

Vector3 dir;

foreach (GameObject w in Wolves)

{

foreach (GameObject r in Rabbits)

{

if (CanSeeWolves(w.transform, r.transform, out dir) && w != r)

{

StimBuffer.Add(new PerceptionStimulus(PerceptionStimulus.StimulusTypes.VisualCanSee, r, r.transform.position, dir, 0f, w));

w.GetComponent<BTController>().Rabbit01 = r;

}

}

}

}

private static bool CanSeeWolves(Transform Reg, Transform Wol, out Vector3 Dir)

{

float Height = Reg.GetComponent<CharacterController>().height;

WolfSenses wolfSenses = Reg.GetComponent<WolfSenses>();

Dir = Wol.position - Reg.position;

float DistanceToRabbit = Vector3.Distance(Reg.position, Wol.position);

if(wolfSenses.sightRange > DistanceToRabbit)

{

float angle = Vector3.Angle(Dir, Reg.forward);

angle = System.Math.Abs(angle);

if (angle < (wolfSenses.viewAngle / 2))

{

RaycastHit hitData;

LayerMask playerMask = 1 << 9;

LayerMask coverMask = 1 << 8;

LayerMask aiMask = 1 << 10;

LayerMask mask = coverMask | playerMask | aiMask;

float targetHeight = (Wol.GetComponent<CharacterController>().height / 1.25f);

Vector3 registrantEyePosition = new Vector3(Reg.position.x, Reg.position.y + Height, Reg.position.z);

Vector3 targetBodyPosition = new Vector3(Wol.transform.position.x, Wol.transform.position.y + targetHeight, Wol.transform.position.z);

Vector3 rayDirection = (targetBodyPosition - registrantEyePosition).normalized;

bool hit = Physics.Raycast(registrantEyePosition, rayDirection, out hitData, wolfSenses.sightRange, mask.value);

Debug.DrawRay(registrantEyePosition, rayDirection \* wolfSenses.sightRange, Color.red);

if (hit)

{

if (hitData.collider.tag == "Rabbit")

return true;

}

}

}

return false;

}

private bool Filter(PerceptionTracker tracker, GameObject Rabbit, PerceptionStimulus perceptionStimulus)

{

if (perceptionStimulus.stimSource != Rabbit)

{

PerceptionTracker.AgentStatus Type = tracker.agentStatus;

int validAgentTypes = tracker.agentStatusFilter;

PerceptionTracker.AgentStatus sourceAgentType;

if(perceptionStimulus.stimSource.GetComponent<PerceptionTracker>() == null)

{

sourceAgentType = PerceptionTracker.AgentStatus.Wolf;

}else

{

sourceAgentType = perceptionStimulus.stimSource.GetComponent<PerceptionTracker>().agentStatus;

}

if ((sourceAgentType == PerceptionTracker.AgentStatus.Rabbit) && (Type == PerceptionTracker.AgentStatus.Wolf) && ((validAgentTypes & PerceptionTracker.RABBIT) != 0)) return true;

if ((sourceAgentType == PerceptionTracker.AgentStatus.Rabbit) && (Type == PerceptionTracker.AgentStatus.Rabbit) && ((validAgentTypes & PerceptionTracker.WOLF) != 0)) return true;

if ((sourceAgentType == PerceptionTracker.AgentStatus.Rabbit) && (Type == PerceptionTracker.AgentStatus.Neutral) && ((validAgentTypes & PerceptionTracker.NEUTRAL) != 0)) return true;

if ((sourceAgentType == PerceptionTracker.AgentStatus.Wolf) && (Type == PerceptionTracker.AgentStatus.Wolf) && ((validAgentTypes & PerceptionTracker.WOLF) != 0)) return true;

if ((sourceAgentType == PerceptionTracker.AgentStatus.Wolf) && (Type == PerceptionTracker.AgentStatus.Rabbit) && ((validAgentTypes & PerceptionTracker.RABBIT) != 0)) return true;

if ((sourceAgentType == PerceptionTracker.AgentStatus.Wolf) && (Type == PerceptionTracker.AgentStatus.Neutral) && ((validAgentTypes & PerceptionTracker.NEUTRAL) != 0)) return true;

if ((sourceAgentType == PerceptionTracker.AgentStatus.Neutral) && (Type == PerceptionTracker.AgentStatus.Wolf) && ((validAgentTypes & PerceptionTracker.NEUTRAL) != 0)) return true;

if ((sourceAgentType == PerceptionTracker.AgentStatus.Neutral) && (Type == PerceptionTracker.AgentStatus.Rabbit) && ((validAgentTypes & PerceptionTracker.NEUTRAL) != 0)) return true;

if ((sourceAgentType == PerceptionTracker.AgentStatus.Neutral) && (Type == PerceptionTracker.AgentStatus.Neutral) && ((validAgentTypes & PerceptionTracker.NEUTRAL) != 0)) return true;

return false;

}

else { return false; }

}

}

}

# PerceptionTracker

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

namespace Perception

{

public class PerceptionTracker : MonoBehaviour

{

public enum Responses { ResNothing,ResRun,ResAttack}

private const float WaitTime = 0.5f;

public PerceptionManager perceptionManager;

public const int NEUTRAL = 0;

public const int WOLF = 1;

public const int RABBIT = 1;

public enum AgentStatus { Neutral,Wolf,Rabbit};

public AgentStatus agentStatus = AgentStatus.Rabbit;

public int agentStatusFilter;

void Awake()

{

perceptionManager = GameObject.Find("wolf\_02").GetComponent<PerceptionManager>();

agentStatusFilter = RABBIT;

}

public void FilteredStimulus(PerceptionStimulus percepStimulus)

{

Debug.Log("Type : " + percepStimulus.StimType + " from " + percepStimulus.stimSource);

}

public Responses GetResponse()

{

return Responses.ResNothing;

}

}

}

# WolfSenses

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

namespace Perception

{

public class WolfSenses : MonoBehaviour

{

public GameObject target;

private CharacterController characterController;

public float viewAngle = 200f;

public float sightRange = 50f;

public float engageTargetAngle = 20f;

public float CorrectionFactor = 20f;

void Start ()

{

characterController = GetComponent<CharacterController>();

}

void Update ()

{

if(target == null)

{

target = GameObject.FindGameObjectWithTag("Rabbit");

TargetSightCheck();

}

}

public bool TargetSightCheck()

{

if (target != null)

{

float DistanceToRabbit = Vector3.Distance(target.transform.position, transform.position);

if (sightRange >= DistanceToRabbit)

{

Vector3 targetDirection = target.transform.position - transform.position;

float targetAngle = Vector3.Angle(targetDirection, transform.forward);

targetAngle = System.Math.Abs(targetAngle);

if (targetAngle < (viewAngle / 2))

{

CharacterController targetCharacterController = target.GetComponent<CharacterController>();

RaycastHit hitData;

LayerMask playerMask = 1 << 9;

LayerMask aiMask = 1 << 10;

LayerMask coverMask = 1 << 8;

LayerMask mask = coverMask | playerMask | aiMask;

float targetHeight = targetCharacterController.height;

float height = characterController.height;

Vector3 eyePos = new Vector3(transform.position.x, transform.position.y + height, transform.position.z);

Vector3 targetPos = new Vector3(target.transform.position.x, target.transform.position.y - (targetHeight / 2.0f), target.transform.position.z);

Vector3 dir = (targetPos - transform.position).normalized;

bool hit = Physics.Raycast(eyePos, dir, out hitData, sightRange, mask.value);

Debug.DrawRay(eyePos, dir \* sightRange, Color.blue);

if (hit)

{

if (hitData.collider.tag == target.gameObject.tag)

{

return true;

}

}

}

}

}

return false;

}

public bool IsBehindCoverCheck(GameObject Wolf, GameObject target)

{

float[] checkY = { 0.1f, 0.5f, 1f };

bool[] covered = { false, false, false };

float range = 3f;

RaycastHit hit;

LayerMask coverMask = 1 << 9;

Vector3 checkPosition;

Vector3 direction = (target.transform.position - Wolf.transform.position).normalized;

for(int n = 0; n < checkY.Length; n++)

{

checkPosition = new Vector3(Wolf.transform.position.x, Wolf.transform.position.y + checkY[n], Wolf.transform.position.z);

covered[n] = Physics.Raycast(checkPosition, direction, out hit, range, coverMask.value);

}

return ((covered[0] || covered[1] && !covered[2]));

}

}

}

# PerceptionStimulus

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

namespace Perception

{

public class PerceptionStimulus : MonoBehaviour

{

public enum StimulusTypes { VisualCanSee,AudioMovement,AudioAttack}

public StimulusTypes StimType;

public GameObject stimSource;

public Vector3 stimLoc;

public Vector3 stimDir;

public float stimRadius;

public GameObject stimSecondary;

public PerceptionStimulus() { }

public PerceptionStimulus(StimulusTypes stim, GameObject source, Vector3 loc, Vector3 dir, float radius, GameObject secondary)

{

this.StimType = stim;

this.stimSource = source;

this.stimLoc = loc;

this.stimDir = dir;

this.stimRadius = radius;

this.stimSecondary = secondary;

}

}

}