Actividad 1: NavMesh y Steering Behaviors

Para la implementación del vagabundeo se han generado puntos aleatorios en cada frame, dentro de una esfera de tamaño “wanderRadius”. Una vez generado el punto, comprobamos que existe el agente al que aplicar la destinación y si aún no ha llegado a la distancia de parada. Si estas condiciones se cumplen, asignamos el punto generado aleatoriamente a la destinación de nuestro NavMeshAgent para que se dirija él.

A continuación el código correspondiente a la actividad:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI;

public class A1_Logic : MonoBehaviour
{
    private NavMeshAgent agent;
    private float speed = 3;
    public static float wanderRadius = 20;

    private void Start()
    {
        agent = GetComponent<NavMeshAgent>();
        if(agent != null)
        {
            agent.speed = speed;
        }
    }

    void Update()
    {
        Vector3 randPos = Random.insideUnitSphere * wanderRadius;

        if (agent != null && agent.remainingDistance <= agent.stoppingDistance)
        {
            agent.destination = randPos;
        }
    }
}
```

Actividad 2: Percepción

Para la creación del sistema de percepción implementado en esta actividad se ha simulado una linterna que se usa para mostrar el campo de visión del nuevo agente. Para ello, se ha generado una mesh con un ángulo de 30º ($\pm$ 15) a partir de triangulos que parten desde el centro del agente.

Para cada iteración, calculamos el ángulo basado en la rotación del agente sobre el eje y. Dado este ángulo, se usa la función *GetVectorFromAngle* para devolver el vector correspondiente dentro del círculo unitario y obtener las direcciones para cada rayo lanzado desde el origen. A continuación, mediante Raycasting, se comprueba si alguno de los rayos lanzados está chocando contra algún objeto y se utiliza el punto exacto de choque (o bien la posición en la distancia máxima si no lo hay) para almacenar esta información en una lista de puntos. A partir de aquí, usamos dicha lista para calcular el número de vertices que tendrá la mesh y el número de triangulos a trazar según los vertices. Por último, sólo tenemos que asignar estos valores a nuestra mesh actual y recalcular las normales para dibujar nuestra linterna.

Se ha implementado un extra con tal de afinar el trazado de los rayos en los ejes. Dada la resolución limitada de la mesh y la implementación por triangulos, una vez pasamos de largo un objeto, el rayo que parte del origen siempre va a parar al extremo del triángulo (equivalenta al punto máximo, dado que sólo tenemos dos destinaciones desde el origen). Con tal de precisar aún más la información recogida tras pasar los bordes, se calcula un punto intermedio entre el máximo y el mínimo para asignar el rayo a este, en vez de la esquina del triángulo. Estos nuevos puntos se añadirán a nuestra lista actual para realizar el cálculo en cada frame.

Por último, para la detección del otro agente, se ha asignado un material a la mesh que cambia de color cuando este entra en el campo de visión. Para ello, la función *DetectAgent* se encarga de comprobar si hay algún target dentro del angulo especificado visión. Como sólo tenemos obstáculos y target en la escena, si *Physics.Raycast()* detecta una colisión con algo que no sea un obstáculo, asumimos que se trata del otro agente y seteamos el booleano *detected* a true y cambiamos el color de la linterna al rojo.

A continuación el código de la implementación de esta actvidad:

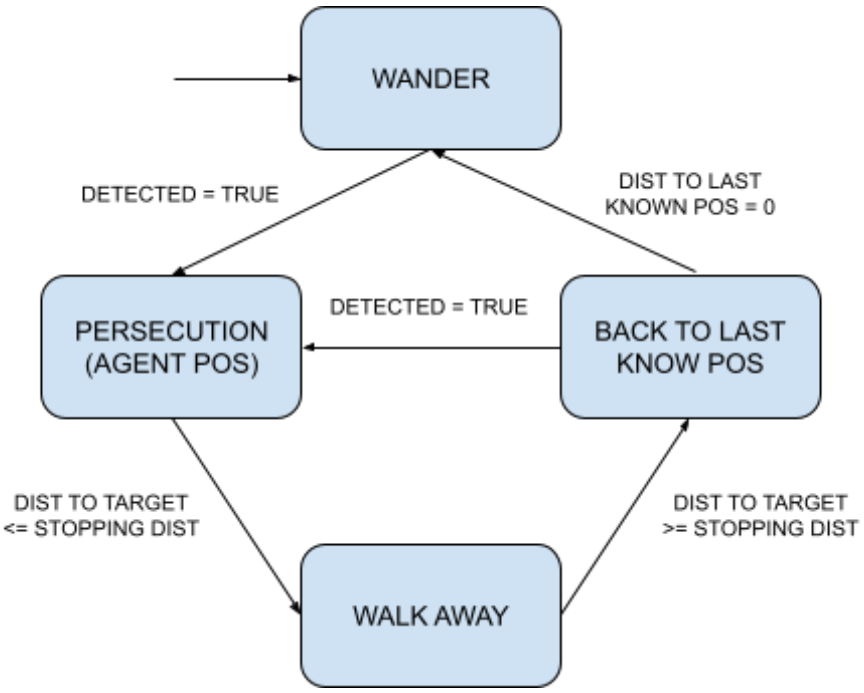
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class A2_FOV : MonoBehaviour
{
    [SerializeField] LayerMask obstacleMask;
    [SerializeField] LayerMask agentMask;

    private int view_angle = 30;
    private int view_dist = 10;
    private int edge_iter = 4;
    private float edge_dist_thrs = 0.5f;

    public MeshFilter mesh_filter;
    private Mesh view_mesh;
    [SerializeField] private GameObject mesh;

    [SerializeField] private Material flashlight_material;
```

```
public static bool detected;

private struct ViewCastInfo
{
    public bool hit;
    public Vector3 hit_point;
    public float ray_distance;
    public float ray_angle;

    public ViewCastInfo(bool _hit, Vector3 _hit_point, float _ray_distance, float _ray_angle)
    {
        hit = _hit;
        hit_point = _hit_point;
        ray_distance = _ray_distance;
        ray_angle = _ray_angle;
    }
}

private struct EdgeInfo
{
    public Vector3 point_a;
    public Vector3 point_b;

    public EdgeInfo(Vector3 _point_a, Vector3 _point_b)
    {
        point_a = _point_a;
        point_b = _point_b;
    }
}

private void Start()
{
    view_mesh = new Mesh();
    view_mesh.name = "View Mesh";
    mesh_filter.mesh = view_mesh;
    flashlight_material.color = new Color32(254, 224, 0, 140);
}

private void LateUpdate()
{
    DrawFov();
    DetectAgent();
}

private void DrawFov()
{
    List<Vector3> view_points = new List<Vector3>();
    ViewCastInfo old_viewcast = new ViewCastInfo();

    for (int i = 0; i <= view_angle; i++)
    {
        float angle = mesh.transform.eulerAngles.y - view_angle/2 + i;

        ViewCastInfo viewcast = ViewCast(angle);

        if (i > 0)
        {
            bool threshold_exceeded = Mathf.Abs(old_viewcast.ray_distance -
```

```
viewcast.ray_distance) > edge_dist_thrs;

            if (old_viewcast.hit != viewcast.hit || (old_viewcast.hit && viewcast.hit &&
threshold_exceeded))
            {
                EdgeInfo edge = GetEdge(old_viewcast, viewcast);
                if(edge.point_a != Vector3.zero)
                {
                    view_points.Add(edge.point_a);
                }
                if (edge.point_b != Vector3.zero)
                {
                    view_points.Add(edge.point_b);
                }
            }
        }

        view_points.Add(viewcast.hit_point);
        old_viewcast = viewcast;
    }

    int vertex_count = view_points.Count + 1;
    Vector3[] vertices = new Vector3[vertex_count];
    int[] triangles = new int[(vertex_count - 2) * 3];

    vertices[0] = Vector3.zero;
    for(int i = 0; i < vertex_count - 1; i++)
    {
        vertices[i + 1] = transform.InverseTransformPoint(view_points[i]);

        if (i < vertex_count - 2)
        {
            triangles[i * 3] = 0;
            triangles[i * 3 + 1] = i + 1;
            triangles[i * 3 + 2] = i + 2;
        }
    }

    view_mesh.Clear();
    view_mesh.vertices = vertices;
    view_mesh.triangles = triangles;
    view_mesh.RecalculateNormals();
  }

  private void DetectAgent()
  {
    flashlight_material.color = new Color32(254, 224, 0, 140);
    Collider[] visibleTarget = Physics.OverlapSphere(transform.position, view_dist,
agentMask);

    for(int i = 0; i < visibleTarget.Length; i++)
    {
        Transform target = visibleTarget[i].transform;
        Vector3 dirToTarget = (target.position - transform.position).normalized;

        if(Vector3.Angle(transform.forward, dirToTarget) < view_angle / 2)
        {
            if (!Physics.Raycast(mesh.transform.position, dirToTarget, view_dist,
obstacleMask) && A3_WalkAway.isWalkingAway == false)
```

```
            {
                detected = true;
                flashlight_material.color = new Color32(255, 0, 0, 140);
            }
        }
    }
}

private Vector3 GetVectorFromAngle(float angle)
{
    return new Vector3(Mathf.Sin(angle * Mathf.Deg2Rad), 0, Mathf.Cos(angle *
Mathf.Deg2Rad));
}

private ViewCastInfo ViewCast(float global_angle)
{
    Vector3 dir = GetVectorFromAngle(global_angle);
    RaycastHit hit;

    if (Physics.Raycast(mesh.transform.position, dir, out hit, view_dist, obstacleMask))
    {
        return new ViewCastInfo(true, hit.point, hit.distance, global_angle);
    }
    else
    {
        return new ViewCastInfo(false, mesh.transform.position + dir * view_dist, view_dist,
global_angle);
    }
}

private EdgeInfo GetEdge(ViewCastInfo min_viewcast, ViewCastInfo max_viewcast)
{
    float min_angle = min_viewcast.ray_angle;
    float max_angle = max_viewcast.ray_angle;
    Vector3 min_point = Vector3.zero;
    Vector3 max_point = Vector3.zero;

    for(int i = 0; i < edge_iter; i++)
    {
        float angle = (min_angle + max_angle) / 2;
        ViewCastInfo viewcast = ViewCast(angle);
        bool threshold_exceeded = Mathf.Abs(min_viewcast.ray_distance -
viewcast.ray_distance) > edge_dist_thrs;

        if (viewcast.hit == min_viewcast.hit && !threshold_exceeded)
        {
            min_angle = angle;
            min_point = viewcast.hit_point;
        }
        else
        {
            max_angle = angle;
            max_point = viewcast.hit_point;
        }
    }

    return new EdgeInfo(min_point, max_point);
}
}
```

Actividad 3: máquina de estados (FSM)




```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI;

public class A3_FSM : MonoBehaviour
{
    [HideInInspector] public A3_Interface currentState;
    [HideInInspector] public A3_Wander a3_Wander;
    [HideInInspector] public A3_Persecution a3_Persecution;
    [HideInInspector] public A3_WalkAway a3_WalkAway;

    [HideInInspector] public NavMeshAgent navMeshAgent;
    [HideInInspector] public NavMeshAgent target;

    [HideInInspector] public float speed = 3;

    private void Awake()
    {
        a3_Wander = new A3_Wander(this);
        a3_Persecution = new A3_Persecution(this);
        a3_WalkAway = new A3_WalkAway(this);

        navMeshAgent = GetComponent<NavMeshAgent>();
        target = GameObject.Find("Agent").GetComponent<NavMeshAgent>();
    }

    private void Start()
    {
```

```
        Debug.Log("WANDER STATE");
        currentState = a3_Wander;

        if (navMeshAgent != null)
        {
            navMeshAgent.speed = speed;
        }
    }

    private void Update()
    {
        currentState.UpdateState();
    }
}
```

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI;

public class A3_Wander : A3_Interface
{
    private readonly A3_FSM fsm;

    public A3_Wander(A3_FSM a3_FSM)
    {
        fsm = a3_FSM;
    }

    public void UpdateState()
    {
        Vector3 randPos = Random.insideUnitSphere * A1_Logic.wanderRadius;

        if (fsm.navMeshAgent != null && fsm.navMeshAgent.remainingDistance <=
fsm.navMeshAgent.stoppingDistance)
        {
            fsm.navMeshAgent.destination = randPos;
        }

        if (A2_FOV.detected == true)
        {
            StatePersecution();
        }
    }

    public void StateWander()
    {
        Debug.Log("Already in wander state");
    }

    public void StatePersecution()
    {
        Debug.Log("PERSECUTION STATE");
        fsm.currentState = fsm.a3_Persecution;
    }

    public void StateWalkAway()
    {
```

```
        Debug.Log("WALK AWAY STATE");
        fsm.currentState = fsm.a3_WalkAway;
    }
}
```

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI;

public class A3_Persecution : A3_Interface
{
    private readonly A3_FSM fsm;

    private float stoppingDistance = 2;
    public static Vector3 lastKnownPosition;

    public A3_Persecution(A3_FSM a3_FSM)
    {
        fsm = a3_FSM;
    }

    public void UpdateState()
    {
        fsm.navMeshAgent.transform.LookAt(fsm.target.transform);
        fsm.navMeshAgent.destination = fsm.target.transform.position;

        if(Vector3.Distance(fsm.navMeshAgent.transform.position,
fsm.target.transform.position) <= stoppingDistance)
        {
            fsm.navMeshAgent.destination = -fsm.target.transform.position * stoppingDistance;
            lastKnownPosition = fsm.target.transform.position;
            Debug.Log(lastKnownPosition);
            A3_WalkAway.isWalkingAway = true;
            A2_FOV.detected = false;
            StateWalkAway();
        }
        else if(A2_FOV.detected == false)
        {
            StateWander();
        }
    }

    public void StatePersecution()
    {
        Debug.Log("Already in persecution state");
    }

    public void StateWander()
    {
        Debug.Log("WANDER STATE");
        fsm.currentState = fsm.a3_Wander;
    }

    public void StateWalkAway()
    {
        Debug.Log("WALK AWAY STATE");
        fsm.currentState = fsm.a3_WalkAway;
```

```
    }
}
```

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI;

public class A3_WalkAway : A3_Interface
{
    private readonly A3_FSM fsm;
    private float stoppingDistance = 10;
    public static bool isWalkingAway = false;
    //private Vector3 lastKnownPos;

    public A3_WalkAway(A3_FSM a3_FSM)
    {
        fsm = a3_FSM;
    }

    public void UpdateState()
    {
        if (Vector3.Distance(fsm.navMeshAgent.transform.position,
fsm.target.transform.position) >= stoppingDistance)
        {
            isWalkingAway = false;

            fsm.navMeshAgent.destination = A3_Persecution.lastKnownPosition;
            float remainingDist = fsm.navMeshAgent.remainingDistance;

            if(A2_FOV.detected == true)
            {
                fsm.navMeshAgent.isStopped = true;
                fsm.navMeshAgent.ResetPath();
                StatePersecution();
                Debug.Log("DECTECTED BEFORE LAST KNOWN POS");
            }
            else if(remainingDist != Mathf.Infinity && fsm.navMeshAgent.pathStatus ==
NavMeshPathStatus.PathComplete && fsm.navMeshAgent.remainingDistance == 0)
            {
                Debug.Log("LAST KNOWN POS REACHED");
                Debug.Log(A3_Persecution.lastKnownPosition);
                StateWander();
            }
        }
    }

    public void StateWalkAway()
    {
        Debug.Log("Already in walk away state");
    }

    public void StatePersecution()
    {
        Debug.Log("PERSECUTION STATE");
        fsm.currentState = fsm.a3_Persecution;
    }
```

```
    public void StateWander()
    {
        Debug.Log("WANDER STATE");
        fsm.currentState = fsm.a3_Wander;
    }
}
```