Informe

Repositorio

Video partes 1 y 2

Video parte 3

Controles

- Desplazar la cámara con WASD
- Orientar la cámara moviendo el ratón
- Zoom con la ruedecita del ratón

Apartados

Boids

Los he implementado en base al código de ejemplo sin añadir mucha cosa más. Algo de ruido al movimiento, ratios de actualización para reducir el coste computacional y una forma más óptima de calcular los vecinos.

Código Boids

Boid

```
/// <summary>
/// Controlador de cada boid individual
/// </summary>
public class Boid : MonoBehaviour
{
    /// <summary>
    /// Su manager
    /// </summary>
    public BoidManager manager;
    /// <summary>
    /// Cuanto queda para actualizar el movimiento del boid
    /// </summary>
    public float updateTimer;

    /// <summary>
    /// Dirección de movimiento actual del boid
    /// </summary>
    public Vector3 direction;

    public float speed =1f;

    /// <summary>
    /// Array de boids cercanos que pueden ser vecinos
    /// </summary>
    public Boid[] boidsNearby;


    // Start is called before the first frame update
    void Start()
    {
        direction = transform.forward * speed;
        boidsNearby = manager.allBoids; ///Por defecto tiene en cuenta a todos los
boids
    }

    // Update is called once per frame
    void Update()
    {

        transform.rotation = Quaternion.Slerp(transform.rotation,
Quaternion.LookRotation(direction),
            manager.rotationSpeed * Time.deltaTime);
        transform.Translate(0.0f, 0.0f, Time.deltaTime * speed);

        updateTimer += Time.deltaTime ;
        if(updateTimer >= manager.updateRate){ //Actualiza la velocidad cada
cierto tiempo
            calculateSpeed();
            updateTimer = 0;
        }

    }
```

```
    /// <summary>
    /// Fórmula del cálculo de la separación
    /// </summary>
    void forceSeparation(Boid go, ref Vector3  separation, float distance){

        var dsp = (distance);
        if (distance < manager.tooCloseDistance)
            dsp *= (distance/manager.tooCloseDistance);
        var res = manager.avoidance*(transform.position - go.transform.position) /
                                (dsp);
        separation += res;
    }

    /// <summary>
    /// Calcula la velocidad en base a los vecinos
    /// </summary>
    void calculateSpeed(){
        Vector3 cohesion, align, separation;
        cohesion = align = separation = Vector3.zero; //Para debuggear el resto
del código

        int num = 0;
        foreach (Boid go in boidsNearby) //Tener en cuenta solo boids "cercanos"
        {
            float distance = Vector3.Distance(go.transform.position,
transform.position);
            if (go != this) { //Ignore self
                if (distance <= manager.neighbourDistance)
                {
                    cohesion += go.transform.position; //Mantener cohesión
                    forceSeparation(go, ref separation, distance); //Mantener
separación
                    align += go.direction; //Alinear con vecinos
                    num++;
                }

            }



        }
        align += direction; //Sigue también su propio alineamiento (evita bugs)
        //Divide by boids
        align /= num+1;
        //align += Random.insideUnitSphere * manager.directionNoise;
        speed = Mathf.Clamp(align.magnitude, manager.minSpeed, manager.maxSpeed);


        if (num > 0)
            cohesion = (cohesion / num - transform.position).normalized * speed;
```

```
            direction = (cohesion + align + separation).normalized * speed
                + Random.insideUnitSphere * manager.directionNoise;
            stayWithinBounds();
        }

        /// <summary>
        /// Altera la dirección de los boids para que vuelvan al interior de los
límites si se salen
        /// </summary>
        void stayWithinBounds(){
            var nextpos = direction + transform.position;
            if (manager.boundsBox.Contains(nextpos))
                return;

            // nextpos = manager.boundsBox.ClosestPoint(nextpos);
            // direction = nextpos - transform.position;
            var strength = manager.boundsBox.SqrDistance(nextpos);
            var goBack = manager.transform.position - transform.position;

            direction += goBack * strength;
        }
    }
```

Boid Manager

```
/// <summary>
/// Controlador del conjunto de Boids
/// </summary>
public class BoidManager : MonoBehaviour
{

    [Header("General")]
    /// <summary>
    /// Lista de todos los boids
    /// </summary>
    public Boid[] allBoids;

    /// <summary>
    /// Número de boids a crear
    /// </summary>
    public RandomBetweenInt numBoids;
    /// <summary>
    /// prefab de los boids
    /// </summary>
    public GameObject boidPrefab;

    /// <summary>
    /// Dimensiones a partir de las que generamos la <see cref="boundsBox"/>
    /// </summary>
    public Vector3 bounds = new Vector3(10, 7, 10);
    [System.NonSerialized]
```

```
        /// <summary>
        /// Límites del area en la que se mueven los boids
        /// </summary>
        public Bounds boundsBox ;

        [Header("Boid Settings")]
        /// <summary>
        /// Velocidad a la que los boids ajustan su dirección
        /// </summary>
        public float rotationSpeed;

        /// <summary>
        /// Velocidad mínima de los Boids
        /// </summary>
        public float minSpeed =0.5f;

        /// <summary>
        /// Velocidad máxima de los Boids
        /// </summary>
        public float maxSpeed = 1.5f;

        /// <summary>
        /// Rango en el que spawnear los boids de forma aleatoria
        /// </summary>
        public float spawnRange = 6;

        /// <summary>
        /// Cada cuanto calcular el comportamiento de los boids
        /// </summary>
        public float updateRate =0.1f;

        /// <summary>
        /// Cada cuanto recalcular los voids cercanos que se tienen en cuenta
        /// </summary>
        public float watchNearbyRate =0.5f;

        /// <summary>
        /// Timer de <see cref="watchNearbyRate"/>
        /// </summary>
        public float watchNearbyTimer = 0;

        /// <summary>
        /// Distancia dentro de la cual los boids se consideran "vecinos"
        /// </summary>
        public float neighbourDistance;

        /// <summary>
        /// Distancia dentro de la cual los boids se consideran demasiado cercanos
        /// </summary>
        public float tooCloseDistance;

        /// <summary>
        /// Ratio de evitación
        /// </summary>
```

```
    public float avoidance;

    /// <summary>
    /// Distancia a partir de la cual no se comprueba si los voids son vecinos
    /// </summary>
    public float nearbyDistance;

    /// <summary>
    /// Aleatoriedad que añadir al movimiento
    /// </summary>
    public float directionNoise = 0.05f;

    // Start is called before the first frame update
    void Start()
    {
        initBoids();
        boundsBox = new Bounds(transform.position, bounds);
    }

    // Update is called once per frame
    void Update()
    {
        watchNearbyTimer += Time.deltaTime;
        if (watchNearbyTimer >= watchNearbyRate || Input.GetKeyDown(KeyCode.C)){
            calculateNearby();
            watchNearbyTimer = 0;
        }
    }

    /// <summary>
    /// Crea e inicializa los boids
    /// </summary>
    void initBoids(){
        allBoids = new Boid[numBoids];
        for (int i = 0; i < numBoids; ++i) {
            Vector3 pos = this.transform.position + Random.insideUnitSphere *
spawnRange; // random position
            Vector3 randomize = Random.insideUnitSphere.normalized; // random
vector direction
            var newBoid = (GameObject)Instantiate(boidPrefab, pos,
Quaternion.LookRotation(randomize));
            allBoids[i] = newBoid.GetComponent<Boid>();
            allBoids[i].manager = this;
        }
    }

    /// <summary>
    /// Calcula los boids "cercanos" de cada boid
    /// (los que es posible que sean sus vecinos ahora o en el futuro cercano).
    /// </summary>
    void calculateNearby(){
        var nearbyDict = new Dictionary<Boid, Queue<Boid>>();
        int num = 1;
        foreach (Boid boid in allBoids){
```

```
            nearbyDict.Add(boid, new Queue<Boid>());
        }
        foreach (Boid boid in allBoids){ //Itera todos los boids
            for(var i = num; i<allBoids.Length; i++){ //Evita comparar con si
mismo y boids ya iterados
                float distance = Vector3.Distance(boid.transform.position,
allBoids[i].transform.position);

                if(distance <= nearbyDistance){
                    nearbyDict[boid].Enqueue(allBoids[i]);
                    nearbyDict[allBoids[i]].Enqueue(boid);
                }
            }
            num++;
        }

        foreach (Boid boid in allBoids){
            boid.boidsNearby = nearbyDict.TryGetValue(boid, out var queue)?
queue.ToArray() : new Boid[0];

        }
    }


    void OnDrawGizmos()
    {
        Gizmos.color = Color.cyan;
        //Dibujar gizmos de los límites
        Gizmos.DrawWireCube(transform.position, bounds);
    }
}
```

Behavior Trees

No voy a incluir el código en este apartado porque ocuparía una cantidad absurda de páginas. Esto se debe al funcionamiento del plugin que me ha obligado a crear multitud de scripts para realizar tareas básicas como asignar un valor o instanciar un prefab en una posición relativa al objeto actual ya que no venían de forma base en el plugin. Aviso también de que la documentación y comentarios de estos scripts no están muy elaborados, de nuevo, porque son demasiados y muchos de ellos hacen cosas básicas o han sido modificados una y otra vez por bugs con el plugin. Aún así, intentaré hacer un resumen breve de todos ellos.

Custom Bricks

Se encuentran todos en la carpeta Scripts/AI/Behaviors, en la subcarpeta correspondiente a su tipo

Actions

- **Blackboard actions**: Son usadas para solucionar el problema de que Behavior Bricks no te deja acceder a la blackboard desde fuera del behavior tree mediante una blackboard propia.

    - **SetMyBlackboard**: Permite asignar valores a las variables de MyBlackboard desde el Behavior Tree
    - **AccessMyBlackboard**: Permite extraer valores de las variables de MyBlackboard para usarlos en el Behavior Tree
- **Speech actions**: Acciones usadas para controlar los bocadillos de diálogo
    - **SpeakAction**: Pone un Sprite concreto en el diálogo permite configurarlo para que dure una cantidad limitada de tiempo, o incluso que se ejecute en paralelo y se detenga en cuanto la otra tarea finalize.
    - **ShutUpAction**: Vacia el diálogo
- **Basic Actions**: acciones muy fundamentales que no entiendo como no vienen ya por defecto dentro del plugin
    - **IncreaseFloat**: Permite sumar un valor a una variable float de la blackboard.
    - **InstantiateRelative**: Versión modificada de la acción Instantiate que permite instanciar objetos con posición y rotación relativas al objeto que ejecuta el BehaviorTree.
    - **SetAgentSpeed**: Permite cambiar la velocidad del navmesh agent.
- **FindTrashCan**: Obtiene la ubicación de la papelera más cercana
- **SetFormationMode**: Cambia la forma de la formación actual (solo puede ejecutarla el líder)

Conditions

- **CheckMyBlackboard**: Compara un valor con una variable de la blackboard y devuelve el resultado.
- **Basic Chechs**: Condiciones muy fundamentales que no entiendo como no vienen ya por defecto dentro del plugin
    - **CheckFloat**: Compara el valor de un float
    - **CheckIfNull**: Compureba si una variable es null
- **Environment Checks**: Comprueban ciertas cosas del entorno.
    - **CanPoop**: Comprueba si el perro cumple las condiciones necesarias para hacer caca.
    - **IsDogNearby**: Comprueba si hay algún perro cerca y guarda el más cercano. Se puede hacer que filtre solo los que están haciendo caca.
    - **IsObjectNearby**: Función genérica para detectar objetos mediante tags. Al final no la uso.
    - **IsPoopNearby**: Comprueba si hay una caca cerca y guarda la más cercana.

Árboles Creados

Los dos árboles de comportamiento principales son los del limpiador y del perro. Son bastante complejos y constan de varios sub árboles pero a continuación voy a resumir su funcionamiento y la intención de mi diseño.

Como utilizo cápsulas en vez de modelos, he decidido representar las acciones que toman como bocadillos de diálogo que se muestran sobre sus cabezas. Durante el tiempo que dura la acción.

Limpiador

La idea principal es que actue como el "poli" de la plaza. Moviéndose por ella y espantando a los perros cuando toque. Pero al mismo tiempo ha de emular el comportamiento de un limpiador.

Para lograrlo, he hecho que cada cierto tiempo se le "llene la bolsa" y tenga que ir a "tirar la basura" a una de las papeleras que hay repartidas por el mapa para poder continuar limpiando.

Para emular lo del comportamiento de Poli vs Ladrón he hecho que cuando ve a un perro haciendo caca (tras un momento de sorpresa) vaya a asustarlo para hacer que se vaya. Y si por algún motivo llega tarde y se encuentra el regalito, pues simplemente lo limpia.

Perro

El perro funciona de forma similar. Cada cierto tiempo tiene que hacer sus necesidades, así que en cuanto se encuentra en un sitio válido, se parará a hacerlas. Una vez termine, el ciclo se resetea.

Si el limpiador interrumpe al perro, este se irá asustado. Corriendo más rápido de lo normal por la plaza. Y no se planteará volver a pararse a hacer sus necesidades hasta que pase un tiempo y se relaje.

Extras

Desde el principio tenía pensado hacer que el tema "poli vs ladron" fuera en realidad "perro vs limpiador" y que las cacas sirvieran como elemento de unión del primer y segundo apartado, pero quería una forma de poder influenciar el comportamiento de los otros agentes (que evitaran las cacas) que no requiriera ser hardcodeado en cada uno de los agentes (especialmente los de la práctica anterior), así que lo que he hecho es que las cacas modifiquen la navmesh.

Al principio intenté usar NavmeshObstacles pero eso generaba problemas con el que el limpiador limpiara las cacas, así que en vez de eso busqué una forma de modificar la navmesh en tiempo real y encontré una bastante simple y super eficiente, así que decidí copiarla y adaptarla un poco.

Al final las cacas han acabado siendo un Navmesh obstacle con un cilindro invisible debajo cuyo navmesh area es de tipo "Poop". Lo del cilindro es un apaño porque el navmeshModifierVolume solo permite rectángulos y no quería pelearme para ver cómo hacer literalmente la cuadratura del círculo con eso.

Group Behavior

Para este apartado lo que he hecho es crear dos clases nuevas: un "controlador de formación" y un "controlador de miembro".

La idea es que el controlador de formación se encarga de decir a cada miembro cual es su posición asignada y luego cada miembro se encarga individualmente de ir a donde le corresponde.

Algoritmos de formación

En vez de crear posiciones preasignadas o algo por el estilo, he decidido hacer un sistema "escalable" que acepta cualquier cantidad de miembros y simplemente "calcula" cual sería su posición relativa en base al número que les corresponde dentro de la lista de miembros.

Una vez calculada, la posición asignada no se vuelve a recalcular a no ser que sea necesario debido a un cambio de la formación (como que se decida cambiar de forma). En vez de hacer eso, se aplica la posición asignada como un "offset" respecto a la del líder, y así obtenemos la posición en coordenadas reales.

CONTINUAR

Código Grupos

Formation

```
/// <summary>
/// Gestiona los movimientos en formación
/// </summary>
public class Formation : MonoBehaviour
{

    public List<FormationMember> members = new List<FormationMember>();

    public NavMeshAgent agent;

    /// <summary>
    /// Forma actual de la formación
    /// </summary>
    public FormationShape shape;

    /// <summary>
    /// Distancia de separación entre los miembros de la formación
    /// </summary>
    public float spread = 2f;

    /// <summary>
    /// Tolerancia en la comprobación de que cada miembro esté en su sitio, ver
    /// <see cref="FormationMember.isInPlace"/>
    /// </summary>
    public float tolerance = 1f;

    /// <summary>
    /// Mensaje que mostrar al cambiar a este modo de formación
    /// </summary>
    public Sprite[] shapeMessages;

    /// <summary>
    /// "Bocadillo" que usar para reproducir mensajes
    /// </summary>
    private SpeechBubble speechBubble;


    /// <summary>
    /// Ha habido cambios desde el último updates
    /// </summary>
    public bool isDirty;

    /// <summary>
    /// Indica que todos los miembros de la formación están en su lugar
    /// </summary>
    public bool allInPlace{
        get {
            return members.All(m => m.isInPlace);
        }
    }
```

```
    // Start is called before the first frame update
    void Start()
    {
        agent = GetComponent<NavMeshAgent>();
        speechBubble = GetComponentInChildren<SpeechBubble>();
    }

    // Update is called once per frame
    void Update()
    {
        if (isDirty)
            applyChanges();
    }

    /// <summary>
    /// Aplica los cambios a los miembros de la formación
    /// </summary>
    public void applyChanges(){
        var num = 0;
        foreach( var member in members){
            (member.formPosition, member.formRotation) = getPosition(num);
            num++;
        }
        isDirty = false;
        speechBubble.say(shapeMessages[(int)shape]);
    }

    /// <summary>
    /// Obtiane la posición que le corresponde a un miembro de la formación
    /// </summary>
    /// <param name="memberNum">nº de miembro cuya posición queremos saber</param>
    /// <returns>Tupla con la posición y rotación correspondientes (null si no
hay)</returns>
    public (Vector3?, Quaternion?) getPosition(int memberNum){

        if(shape == FormationShape.circle){ //Formación en círculo
            int ringNum=1, levelPos =memberNum, ringPos ;

            //Encontrar en qué nivel del anillo le toca ponerse
            while(levelPos >= ( ringPos = ringPositions(ringNum))){
                levelPos -= ringPos;
                ringNum++;
            }
            var offset =  360 * levelPos / ringPos;
            var rotation = Quaternion.AngleAxis(offset, Vector3.up);
            var pos = (Vector3.forward )* ringRadius(ringNum);
            if (ringNum % 2 == 0)
                pos *= -1; //Alternar el lado en el que empiezan los círculos
            pos = rotation * pos;
            return (pos, rotation);

        }else if(shape == FormationShape.square){ //Formación cuadrada
```

```
            var offset = squarePosition(memberNum);

            var xPos = offset.x % 2 == 0 ? Vector3.right : Vector3.left;
            xPos *= (offset.x / 2 + 0.5f) * spread;
            var yPos = Vector3.forward * spread * offset.y;


            return (xPos + yPos, Quaternion.identity);
        }
        return (null, null);
    }

    /// <summary>
    /// Calcula cuantas posiciones posibles hay dentro de un nivel de la formación
de anillo
    /// </summary>
    /// <param name="ringNum"></param>
    /// <returns></returns>
    public int ringPositions(int ringNum){

        return (int)(ringCircumference(ringNum) / spread);
    }

    /// <summary>
    /// Calcula la circumferencia de un nivel de los anillos
    /// </summary>
    /// <param name="ringNum">nº de anillo (desde dentro)</param>
    /// <returns></returns>
    public float ringCircumference(int ringNum){
        return Mathf.PI * 2 * ringRadius(ringNum);
    }

    /// <summary>
    /// Calcula el radio de un anilllo de la formación en círculo
    /// </summary>
    /// <param name="ringNum">nº de anillo (desde dentro)</param>
    /// <returns></returns>
    public float ringRadius(int ringNum){
        return ringNum  * spread * 0.8f - (ringNum-1)*0.5f ;
    }


    /// <summary>
    /// Calcula la forma de la formación cuadrada y obtiene las coordenadas del
miembro indicado
    /// </summary>
    /// <param name="memberNum">número del miembro cuya posición queremos
saber</param>
    /// <returns>"Coordenadas" del miembro (ALERTA! hay que convertirlas a
distancias)</returns>
    public Vector2Int squarePosition(int memberNum){
        //Calcular medidas de la formación
        var count = members.Count;
        var twice = count*2;
```

```
        int sizex = Mathf.CeilToInt(Mathf.Sqrt(twice))/2*2;

        //Calcular coordenadas dentro de esas medidas
        int x=0,y=1, n = 0;
        while (n < memberNum){
            n++;
            x++;
            if (x >= sizex) { x = 0; y++; }
        }

        return new Vector2Int(x,y);

    }


}
```

FormationMember

```
/// <summary>
/// Script que gestiona a los miembros de una formación
/// </summary>
public class FormationMember : MonoBehaviour
{

    public NavMeshAgent agent;

    /// <summary>
    /// Formación a la que pertenece
    /// </summary>
    public Formation formation;

    /// <summary>
    /// Su posición dentro de la formación (relativa a la del líder)
    /// </summary>
    public Vector3? formPosition;

    /// <summary>
    /// Su rotación dentro de la formación (relativa a la del líder)
    /// </summary>
    public Quaternion? formRotation;

    /// <summary>
    /// Indica si el miembro de la formación está en el lugar que se le ha
asignado
    /// </summary>
    public bool isInPlace {
        get {
            if (formPosition == null)
                return true;
            var worldpos = formationPosToWorld((Vector3)formPosition,
```

```
formation.transform.rotation);
            var dist = Vector3.Scale((Vector3)(agent.nextPosition - worldpos), new
Vector3(1,0,1));
            Debug.Log(dist);
            return  dist.magnitude <= agent.stoppingDistance +
formation?.tolerance;
        }
    }

    void Awake()
    {
        agent = GetComponent<NavMeshAgent>();

        var formations = GameObject.FindObjectsOfType<Formation>();
        var best = formations.Select(t =>
                    new
                    {
                        t,
                        distance = Vector3.Distance(t.transform.position,
transform.position)
                    }).Aggregate(new { t = (Formation)null, distance =
Mathf.Infinity }, (i1, i2) => i1.distance < i2.distance ? i1 : i2);

        joinFormation(best.t);
    }
    // Start is called before the first frame update
    void Start()
    {


    }

    // Update is called once per frame
    void Update()
    {
        followFormation();
    }

    /// <summary>
    /// Hace que el miembro se una a una formación
    /// </summary>
    /// <param name="form"></param>
    public void joinFormation(Formation form){
        var num = form.members.Count;
        form.members.Add(this);
        formation = form;
        formation.isDirty = true;
    }


    /// <summary>
    /// Actualiza el destino del navmesh agent para que siga la formación
    /// </summary>
    public void followFormation(){
```

```
        if (formation == null || formPosition == null || formRotation == null)
            return;

        var addRot = formation.transform.rotation;
        var worldpos = formationPosToWorld((Vector3)formPosition, addRot);
        if (agent.destination != worldpos){
            agent.destination = worldpos; //actualiza la posición

        } else if(!agent.pathPending && agent.remainingDistance <=
agent.stoppingDistance){
            //Girar el agente en la dirección que toca
            transform.rotation = Quaternion.Slerp(transform.rotation,
(Quaternion)formRotation * addRot, Time.deltaTime * agent.angularSpeed);
        }
    }

    /// <summary>
    /// Convierte la posición relativa dentro de la formación en un valor de
posición absoluto
    /// </summary>
    /// <param name="relativePos"></param>
    /// <param name="rotation"></param>
    /// <returns></returns>
    public Vector3 formationPosToWorld(Vector3 relativePos,Quaternion rotation ){
        return (rotation * relativePos) + formation.transform.position;
    }
}
```