Carrera: Ingeniería de Software

Materia: Videojuegos II

Alumno: Emanuel Rendon Veloz

Profesor: Nicolas Arrioja Landa Cosio

12/03/2024

Introducción

Este juego tiene como protagonista a un pollo y se desarrolla en una pequeña granja que sirve como mapa principal. El objetivo es que el pollo llegue a un tomate viviente antes de que lo haga un perro hambriento, ya que si el perro llega primero, el jugador perderá. Sin embargo, no será tan sencillo: en el camino se encuentran los "Tres Mosqueteros", un grupo de gatos que patrullan la zona. Si el pollo es descubierto por ellos, será devorado y también perderá el juego.

Para el desarrollo de este juego se implementaron algoritmos de inteligencia artificial que permiten que el pollo se mueva libremente por el mapa, mientras que los personajes no jugables (el perro y los gatos) tienen comportamientos dinámicos basados en varios estados, lo que agrega un nivel adicional de desafío y diversión. Todo está diseñado para brindar una experiencia entretenida y agradable al jugador.

Antecedentes

El desarrollo de este juego se basó en una variedad de algoritmos aprendidos en clase, asignando a cada objeto un comportamiento único que reacciona de forma diferente según sus mecánicas. La inteligencia artificial (IA) juega un papel fundamental al permitir que los enemigos interactúen con el jugador, generando un desafío dinámico. Este enfoque convierte al juego en una experiencia competitiva, ya que implementa un Modelo de Estados Finitos (MEF) para los enemigos. Cada estado aporta un comportamiento específico, como atacar al jugador o vigilar activamente, lo que genera situaciones impredecibles que desafían la capacidad del jugador para adaptarse y tomar decisiones estratégicas en tiempo real.

Descripción de problema

El juego está diseñado para quienes buscan una experiencia única en juegos de carrera, ofreciendo más que simplemente "correr hacia el tomate". Combina velocidad, reflejos y estrategia, creando una presión constante al evitar ser atrapado o devorado por los gatos. Este diseño pone a prueba la habilidad del jugador para gestionar riesgos y aprovechar las mecánicas del juego, elevando la experiencia al hacer que cada acción tenga consecuencias inmediatas.

Objetivo general

El objetivo principal del juego es guiar al pollo a través de la granja, tomando decisiones estratégicas que pondrán en riesgo su seguridad. Cada elección puede comprometer el desempeño del jugador, transformando el juego en una persecución frenética en la que el pollo deberá huir para sobrevivir. Este entorno dinámico busca mantener a los jugadores emocionados y entretenidos, mientras intentan evitar ser devorados por los enemigos.

Objetivos específicos

Diseñar el mapa de la granja: Incluir bardas que sirvan como coberturas y obstáculos, aumentando la dificultad al limitar los movimientos del pollo.

Establecer un final claro: El juego concluye cuando el pollo llega al tomate o es eliminado por los enemigos, incentivando una experiencia equilibrada entre desafío y recompensa.

Definir comportamientos de los enemigos: Usar estados como vigilancia, descanso o persecución para crear un entorno dinámico y desafiante.

Incrementar la jugabilidad: Integrar elementos como mecánicas de desgaste en los enemigos, añadiendo profundidad estratégica para el jugador.

Alcances

El juego tiene un gran potencial de escalabilidad. Se puede expandir el mapa, agregar más enemigos con comportamientos únicos o incluir elementos personalizables, como elegir entre diferentes tipos de pollos. También se podrían implementar efectos sonoros inmersivos, como el canto del pollo durante las persecuciones. Esto permitiría adaptar la dificultad según las preferencias del jugador, mejorando la rejugabilidad y la experiencia general.

Limitaciones

Actualmente se ve limitado ya que no cuenta con los mejores gráficos visuales para el jugador por temas de presupuesto para hacerlo de una calidad más elevada, otra de las limitaciones del juego es la falta de un modo multijugador ya que si incluyera esto sería un juego con un alcance mayor ya que podríamos pasar el tiempo con nuestro amigos compitiendo viendo quien es mejor donde ellos podrían ser el gato para que eviten que lleguemos a la meta haciendo así un juego mucho más divertido, otra limitación fue el mapa ya no se pudo realizar más grande como originalmente esperaba.

Referencias

Technologies, U. (s. f.). Unity - Scripting API: Quaternion.Slerp. *https://docs.unity3d.com/ScriptReference/Quaternion.Slerp.html*

nicosiored. (2019k, junio 20). Pathfinding con grafos II - 21 - Inteligencia Artificial Videojuegos [Vídeo]. YouTube. *https://www.youtube.com/watch?v=dGn0pN98AC0*

nicosiored. (2019e, abril 11). Máquina de estados finitos II - 11 - Inteligencia Artificial Videojuegos [Vídeo]. YouTube. *https://www.youtube.com/watch?v=inDwZMW5CGw*

José Núñez. (2022, 2 septiembre). Maquina de estados finitos - UNITY [Vídeo]. YouTube. *https://www.youtube.com/watch?v=keM5wu8Gt1o*

Código

```
//Autor : Emanuel Rendon Veloz
//Fecha : 12/03/2024
//Version : 1.0
//Esta clase maneja el movimiento de un objeto a través de los nodos de un grafo

public class GrafoMove : MonoBehaviour
{
    // Referencia al mapa de nodos
    public MapaGrafo mapa;
    private int nodoActual = 0;
    private bool enObjetivo = false;

    // Este método se llama una vez por cada fotograma. Mueve el objeto desde el
nodo actual
    void Update()
    {
        if (enObjetivo == false)
        {
            //Verifacamos si hemos llegado al nodo
            Vector3 nodoObjetivo = new
Vector3(mapa.nodosCoords[mapa.ruta[nodoActual]].x + 0.5f, 0.5f,
mapa.nodosCoords[mapa.ruta[nodoActual]].z + 0.5f);

            if (Vector3.Magnitude(transform.position - nodoObjetivo) < 0.1)
            {
                nodoActual++;
                if (nodoActual == mapa.ruta.Count)
                    enObjetivo = true;
            }


            transform.LookAt(nodoObjetivo);
            transform.Translate(Vector3.forward * Time.deltaTime * 5);
        }
    }
}
```

```
//Autor : Emanuel Rendon Veloz
//Fecha : 12/03/2024
//Version : 1.0
//Estructura que representa un nodo en el grafo con sus coordenadas (x, z).
public struct nodoGrafo
{
    public float x;
    public float z;

    // Constructor de la estructura nodoGrafo
    public nodoGrafo(float pX, float pZ)
    {
        x = pX;
        z = pZ;
    }
}

// Clase que representa el mapa como un grafo de nodos conectados, donde se
puede calcular una ruta entre dos nodos utilizando el algoritmo de búsqueda de
camino
public class MapaGrafo : MonoBehaviour
{
    public int inicio = 0;
    public int final = 6;
    public List<int> ruta;
    public GameObject personaje;

    private int[,] mAdyacencia;
    private int[] indegree;
    private int cantNodos = 10;  // Cambiado a 10 nodos
    public nodoGrafo[] nodosCoords;
    private bool inicializado = false;

    // Inicializa los elementos del grafo y establece las coordenadas y
conexiones entre nodos y tambien establece la posición del personaje en el nodo
de inicio.
    void Start()
    {
        mAdyacencia = new int[cantNodos, cantNodos];

        // Instanciamos arreglo de ingree
        indegree = new int[cantNodos];

        // Instanciamos el arreglo con las coordenadas de nodos
        nodosCoords = new nodoGrafo[cantNodos];

        ruta = new List<int>();

        // Definimos las aristas
        AdicionaArista(0, 1);
        AdicionaArista(0, 3);
```

```
        AdicionaArista(1, 4);
        AdicionaArista(2, 5);
        AdicionaArista(3, 2);
        AdicionaArista(3, 4);
        AdicionaArista(4, 6);
        AdicionaArista(6, 5);
        AdicionaArista(0, 2);
        AdicionaArista(1, 5);
        AdicionaArista(2, 4);
        AdicionaArista(4, 5);
        AdicionaArista(5, 3);
        AdicionaArista(7, 0);
        AdicionaArista(8, 7);
        AdicionaArista(9, 8);
        AdicionaArista(5, 9);
        AdicionaArista(3, 9);

        //Cordenadas fijas
        AdicionaCoords(0, -35, 10);
        AdicionaCoords(1, 20, -25);
        AdicionaCoords(2, -10, -8);
        AdicionaCoords(3, 25, 15);
        AdicionaCoords(4, 10, -30);
        AdicionaCoords(6, 30, 0);
        AdicionaCoords(7, 0, -20);
        AdicionaCoords(8, 15, 5);
        AdicionaCoords(9, -5, 30);

        inicializado = true;

        // Establecemos la posición del personaje en el nodo de inicio
        personaje.transform.position = new Vector3(nodosCoords[inicio].x, 0.5f,
nodosCoords[inicio].z);

        CalculaRuta();
    }

    // Método que desactiva el mapa y establece la bandera de inicialización en
falso.
    private void OnDisable()
    {
        inicializado = false;
    }

    // Añade una arista entre dos nodos en el grafo
    public void AdicionaArista(int pNodoInicio, int pNodoFinal)
    {
        mAdyacencia[pNodoInicio, pNodoFinal] = 1;

        mAdyacencia[pNodoFinal, pNodoInicio] = 1;
    }
```

```
    public void AdicionaCoords(int pNodo, float pX, float pZ)
    {
        nodosCoords[pNodo] = new nodoGrafo(pX, pZ);
    }

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

    }

    // Método que dibuja el grafo en la escena usando Gizmos.
    private void OnDrawGizmos()
    {
        if (inicializado)
        {
            foreach (nodoGrafo miNodo in nodosCoords)
            {
                Gizmos.color = Color.green;
                Gizmos.DrawSphere(new Vector3(miNodo.x, 0, miNodo.z), 0.5f);
            }

            int n = 0;
            int m = 0;

            // Dibujamos las conexiones entre los nodos
            for (n = 0; n < cantNodos; n++)
                for (m = 0; m < cantNodos; m++)
                {
                    if (mAdyacencia[n, m] != 0)
                    {
                        Gizmos.color = Color.yellow;
                        Gizmos.DrawLine(new Vector3(nodosCoords[n].x, 0,
nodosCoords[n].z), new Vector3(nodosCoords[m].x, 0, nodosCoords[m].z));
                    }
                }
        }
    }

    public int ObtenAdyacencia(int pFila, int pColumna)
    {
        return mAdyacencia[pFila, pColumna];
    }

    // Encuentra el primer nodo con indegree igual a 0
    public void CalcularIndegree()
    {
        int n = 0;
        int m = 0;
```

```
        for (n = 0; n < cantNodos; n++)
        {
            for (m = 0; m < cantNodos; m++)
            {
                if (mAdyacencia[m, n] == 1)
                    indegree[n]++;
            }
        }
    }

    public int EncuentraI0()
    {
        bool encontrado = false;
        int n = 0;

        for (n = 0; n < cantNodos; n++)
        {
            if (indegree[n] == 0)
            {
                encontrado = true;
                break;
            }
        }
        if (encontrado)
            return n;
        else
            return -1;
    }

    public void DecrementaIndegree(int pNodo)
    {
        indegree[pNodo] = -1;

        int n = 0;

        for (n = 0; n < cantNodos; n++)
        {
            if (mAdyacencia[pNodo, n] == 1)
                indegree[n]--;
        }
    }

    // Método que calcula la ruta más corta entre el nodo de inicio y el nodo
final utilizando el algoritmo adecuado.
    public void CalculaRuta()
    {
        int[,] tabla = new int[cantNodos, 3];

        int n = 0;
        int distancia = 0;
        int m = 0;
```

```
        // Inicializamos la tabla
        for (n = 0; n < cantNodos; n++)
        {
            tabla[n, 0] = 0;
            tabla[n, 1] = int.MaxValue;
            tabla[n, 2] = 0;
        }
        tabla[inicio, 1] = 0;

        for (distancia = 0; distancia < cantNodos; distancia++)
        {
            for (n = 0; n < cantNodos; n++)
            {
                if ((tabla[n, 0] == 0) && (tabla[n, 1] == distancia))
                {
                    tabla[n, 0] = 1;
                    for (m = 0; m < cantNodos; m++)
                    {
                        // Verificamos que el nodo adyacente
                        if (ObtenAdyacencia(n, m) == 1)
                        {
                            if (tabla[m, 1] == int.MaxValue)
                            {
                                tabla[m, 1] = distancia + 1;
                                tabla[m, 2] = n;
                            }
                        }
                    }
                }
            }
        }

        ruta.Clear();

        int nodo = final;

        while (nodo != inicio)
        {
            ruta.Add(nodo);
            nodo = tabla[nodo, 2];
        }
        ruta.Add(inicio);

        ruta.Reverse();
    }
}
```

```
//Autor : Emanuel Rendon Veloz
//Fecha : 12/03/2024
enum estados { Patrullar, Perseguir, Huir, Atacar, Dormir, Cansado } //
Agregamos Cansado

public class MEF : MonoBehaviour
{
    private int estado = (int)estados.Patrullar;

    public float velocidad = 5;

    private int cantidadPasos;
    private List<Vector3> ruta = new List<Vector3>();
    private int indice = 0;
    private Vector3 objetivo;

    public Transform jugador;
    public float rangoVision = 25;
    public float rangoFov = 30;

    private Vector3 JugadorDesdeIA;
    float distanciaAJugador = 0;
    float angulo = 0;

    private float tiempoCansado = 0f;
    public float tiempoDeCansancio = 5f;

    // Start is called before the first frame update
    void Start()
    {
        // Creamos la ruta que usaremos en el patrullaje
        ruta.Add(new Vector3(-4, 0.69f, 25));
        ruta.Add(new Vector3(-12, 0.69f, 16));
        ruta.Add(new Vector3(-29, 0.69f, 11));
        ruta.Add(new Vector3(-19, 0.69f, 23));

        indice = 0;
        objetivo = ruta[indice];
        cantidadPasos = ruta.Count;
    }

    // Update is called once per frame
    void Update()
    {
        switch (estado)
        {
            case (int)estados.Patrullar:
                velocidad = 5;
                if ((transform.position - ruta[indice]).magnitude < 1)
```

```
                {
                    indice++;
                    if (indice >= cantidadPasos)
                        indice = 0;

                    objetivo = ruta[indice];
                }
                transform.LookAt(objetivo);
                transform.Translate(Vector3.forward * velocidad *
Time.deltaTime);

                // Verificamos si es necesario cambiar de estado
                if (VeJugador() == true)
                    estado = (int)estados.Perseguir;
                break;

            case (int)estados.Perseguir:
                velocidad = 10;
                transform.LookAt(jugador.position);
                transform.Translate(Vector3.forward * velocidad *
Time.deltaTime);

                // Verificamos si es necesario cambiar de estado
                if ((transform.position - jugador.position).magnitude < 1)
                    estado = (int)estados.Atacar; // Cambio al estado de Atacar
                break;

            case (int)estados.Huir:
                velocidad = 15;
                Vector3 punto = jugador.position - transform.position;
                Vector3 vision = transform.position - punto;

                vision.y = jugador.position.y;
                transform.LookAt(vision);
                transform.Translate(Vector3.forward * velocidad *
Time.deltaTime);

                // Verificamos si es necesario cambiar de estado
                if ((transform.position - jugador.position).magnitude > 50)
                    estado = (int)estados.Patrullar;

                if (Random.Range(1, 100) == 5 && (transform.position -
jugador.position).magnitude > 10)
                    estado = (int)estados.Perseguir;
                break;

            case (int)estados.Atacar:
                velocidad = 0;
                // Lógica de ataque, como animaciones o infligir daño
                Debug.Log("Atacando al jugador");
```

```
                // Aquí la lógica para matar al jugador
                MatarJugador();

                // Después de atacar, regresa a patrullar o fin del juego
                estado = (int)estados.Patrullar; // O puedes terminar el juego
directamente aquí
                break;

            case (int)estados.Dormir:
                velocidad = 0;
                // Lógica de dormir, como esperar un tiempo antes de volver a
patrullar
                Debug.Log("Durmiendo...");
                // Puede cambiar a otro estado tras un tiempo
                if (Random.Range(0, 100) < 5)
                    estado = (int)estados.Patrullar;
                break;
            // Agregamos la lógica para el estado Cansado
            case (int)estados.Cansado:
                velocidad = 0;
                Debug.Log("Cansado... descansando");
                tiempoCansado += Time.deltaTime;

                if (tiempoCansado >= tiempoDeCansancio)
                {
                    // Reiniciamos el tiempo de cansancio
                    tiempoCansado = 0f;
                    // Volver a patrullar después de descansar
                    estado = (int)estados.Patrullar;
                }
                break;
        }
    }

    // Método para verificar si el jugador está en el rango de visión
    public bool VeJugador()
    {
        bool visto = false;

        // Calculamos la distancia cuadrada
        distanciaAJugador = Vector3.SqrMagnitude(transform.position -
jugador.position);

        // Verificamos si está dentro del rango de visión
        if (distanciaAJugador <= (rangoVision * rangoVision))
        {
            // Vector de la IA al jugador
            JugadorDesdeIA = jugador.position - transform.position;

            // Calculamos el ángulo
            angulo = Vector3.Angle(transform.forward, JugadorDesdeIA);
```

```
            // Verificamos si está en el ángulo de visión
            if (angulo <= rangoFov)
            {
                visto = true;
            }
        }
        return visto;
    }

    // Método para matar al jugador y terminar el juego
    private void MatarJugador()
    {
        Destroy(jugador.gameObject);

        // Fin del juego
        Time.timeScale = 0;

    }
}
```

```
//Autor : Emanuel Rendon Veloz
//Fecha : 12/03/2024
enum estados02 { Patrullar, Perseguir, Huir, Investigando, Alertado }

public class MEF02 : MonoBehaviour
{
    private int estado = (int)estados02.Patrullar;

    public float velocidad = 5;

    private int cantidadPasos;
    private List<Vector3> ruta = new List<Vector3>();
    private int indice = 0;
    private Vector3 objetivo;

    public Transform jugador;
    public float rangoVision = 25;
    public float rangoFov = 30;

    private Vector3 JugadorDesdeIA;
    float distanciaAJugador = 0;
    float angulo = 0;

    private Vector3 puntoUltimaPosicionJugador;
    private float tiempoAlertado = 0f;

    // Start is called before the first frame update
    void Start()
    {
        // Creamos la ruta que usaremos en el patrullaje
        ruta.Add(new Vector3(-2, 0.69f, -3));
        ruta.Add(new Vector3(27, 0.69f, 5));
        ruta.Add(new Vector3(3, 0.69f, 13));

        indice = 0;
        objetivo = ruta[indice];
        cantidadPasos = ruta.Count;
    }

    // Update is called once per frame
    void Update()
    {
        switch (estado)
        {
            case (int)estados02.Patrullar:
                velocidad = 5;
                if ((transform.position - ruta[indice]).magnitude < 1)
```

```
                {
                    indice++;
                    if (indice >= cantidadPasos)
                        indice = 0;

                    objetivo = ruta[indice];
                }
                transform.LookAt(objetivo);
                transform.Translate(Vector3.forward * velocidad *
Time.deltaTime);

                // Verificamos si es necesario cambiar de estado
                if (VeJugador() == true)
                    estado = (int)estados02.Perseguir;
                break;

            case (int)estados02.Perseguir:
                velocidad = 10;
                transform.LookAt(jugador.position);
                transform.Translate(Vector3.forward * velocidad *
Time.deltaTime);

                // Verificamos si hay transiciones
                if ((transform.position - jugador.position).magnitude < 1)
                    estado = (int)estados02.Huir;
                break;

            case (int)estados02.Huir:
                velocidad = 15;
                Vector3 punto = jugador.position - transform.position;
                Vector3 vision = transform.position - punto;

                vision.y = jugador.position.y;
                transform.LookAt(vision);
                transform.Translate(Vector3.forward * velocidad *
Time.deltaTime);

                // Verificamos si hay transiciones
                if ((transform.position - jugador.position).magnitude > 50)
                    estado = (int)estados02.Patrullar;

                if (Random.Range(1, 100) == 5 && (transform.position -
jugador.position).magnitude > 10)
                    estado = (int)estados02.Perseguir;
                break;

            case (int)estados02.Investigando:
                velocidad = 3;
                Debug.Log("Investigando la zona...");

                // La IA se dirige a la última posición conocida del jugador
```

```
                transform.LookAt(puntoUltimaPosicionJugador);
                transform.Translate(Vector3.forward * velocidad *
Time.deltaTime);

                // Si la IA llega a la última posición conocida, regresa a
patrullar
                if ((transform.position - puntoUltimaPosicionJugador).magnitude
< 1)
                {
                    estado = (int)estados02.Patrullar;
                }
                break;

            case (int)estados02.Alertado:
                velocidad = 7;
                tiempoAlertado += Time.deltaTime;
                Debug.Log("Alertado, buscando al jugador...");

                // La IA está más atenta y busca al jugador de manera más
eficiente
                if (VeJugador())
                {
                    estado = (int)estados02.Perseguir;
                }

                // Si pasa un tiempo sin encontrar al jugador, vuelve a
patrullar
                if (tiempoAlertado > 5f) // Se mantiene alertada por 5 segundos
                {
                    estado = (int)estados02.Patrullar;
                }
                break;
        }
    }

    // Método para verificar si el jugador está en el rango de visión
    public bool VeJugador()
    {
        bool visto = false;

        distanciaAJugador = Vector3.SqrMagnitude(transform.position -
jugador.position);

        if (distanciaAJugador <= (rangoVision * rangoVision))
        {
            JugadorDesdeIA = jugador.position - transform.position;
            angulo = Vector3.Angle(transform.forward, JugadorDesdeIA);

            if (angulo <= rangoFov)
            {
                visto = true;
```

```
                puntoUltimaPosicionJugador = jugador.position; // Actualizamos
la última posición conocida
            }
        }
        return visto;
    }
}
```

```
//Autor : Emanuel Rendon Veloz
//Fecha : 12/03/2024
enum estados03 { Patrullar, Perseguir, Huir, Vigilar, Esperar }

public class MEF03 : MonoBehaviour
{
    private int estado = (int)estados03.Patrullar;

    public float velocidad = 5;

    private int cantidadPasos;
    private List<Vector3> ruta = new List<Vector3>();
    private int indice = 0;
    private Vector3 objetivo;

    public Transform jugador;
    public float rangoVision = 25;
    public float rangoFov = 30;

    private Vector3 JugadorDesdeIA;
    float distanciaAJugador = 0;
    float angulo = 0;

    // Start is called before the first frame update
    void Start()
    {
        // Creamos la ruta que usaremos en el patrullaje
        ruta.Add(new Vector3(13, 0.69f, 22));
        ruta.Add(new Vector3(-26, 0.69f, 11));
        ruta.Add(new Vector3(-2, 0.69f, 24));
        ruta.Add(new Vector3(-11, 0.69f, 3));
        ruta.Add(new Vector3(-5, 0.69f, 17));

        indice = 0;
        objetivo = ruta[indice];
        cantidadPasos = ruta.Count;
    }

    // Update is called once per frame
```

```
    void Update()
    {
        if (estado == (int)estados03.Patrullar)
        {
            velocidad = 5;

            // Verificamos si hemos llegado a un punto de la ruta
            if ((transform.position - ruta[indice]).magnitude < 1)
            {
                indice++;
                if (indice >= cantidadPasos)
                    indice = 0;

                objetivo = ruta[indice];
            }
            transform.LookAt(objetivo);
            transform.Translate(Vector3.forward * velocidad * Time.deltaTime);

            // Verificamos si es necesaria la transición
            if (VeJugador())
                estado = (int)estados03.Perseguir;
        }
        else if (estado == (int)estados03.Perseguir)
        {
            velocidad = 10;
            transform.LookAt(jugador.position);

            // Nos movemos hacia el objetivo
            transform.Translate(Vector3.forward * velocidad * Time.deltaTime);

            // Verificamos si hay transición
            if ((transform.position - jugador.position).magnitude < 1)
                estado = (int)estados03.Huir;
        }
        else if (estado == (int)estados03.Huir)
        {
            velocidad = 15;
            // Cuida el valor de Y cuando no estemos en un plano
            Vector3 punto = jugador.position - transform.position;
            Vector3 vision = transform.position - punto;
            vision.y = jugador.position.y;

            transform.LookAt(vision);

            // Nos movemos hacia la visión
            transform.Translate(Vector3.forward * velocidad * Time.deltaTime +
transform.position);

            // Verificamos si hay transición
            if ((transform.position - jugador.position).magnitude > 50)
                estado = (int)estados03.Patrullar;
```

```
            if (Random.Range(1, 100) == 5 && (transform.position -
jugador.position).magnitude > 10)
                estado = (int)estados03.Perseguir;
        }
        else if (estado == (int)estados03.Vigilar)
        {
            // Lógica para vigilar (puede ser solo rotar y mirar al jugador)
            velocidad = 2;  // Menor velocidad para vigilar
            transform.LookAt(jugador.position);

            // Puedes agregar más lógica aquí, como detectar cambios en la
posición del jugador o puntos de vigilancia
            if (Random.Range(1, 100) < 10) // Simulando un cambio de vigilancia
                estado = (int)estados03.Patrullar;
        }
        else if (estado == (int)estados03.Esperar)
        {
            // Lógica para esperar (por ejemplo, quedarse en el lugar por un
tiempo)
            velocidad = 0;  // No se mueve mientras espera

            // Lógica para determinar cuándo dejar de esperar
            if (Random.Range(1, 100) < 5) // Después de un tiempo aleatorio, se
mueve
                estado = (int)estados03.Patrullar;
        }
    }

    public bool VeJugador()
    {
        bool visto = false;

        // Calculamos la distancia cuadrada
        distanciaAJugador = Vector3.SqrMagnitude(transform.position -
jugador.position);

        // Verificamos si está en el rango de visión
        if (distanciaAJugador <= (rangoVision * rangoVision))
        {
            // Vector de la IA al personaje
            JugadorDesdeIA = jugador.position - transform.position;

            // Calculamos el ángulo
            angulo = Vector3.Angle(transform.forward, JugadorDesdeIA);

            // Verificamos si está en el ángulo de visión
            if (angulo <= rangoFov)
            {
                visto = true;
            }
```

```
        }
        return visto;
    }
}
```

Diagrama