Instituto tecnológico Nacional de México

Saltillo

Materia: Graficacion

Proyecto final: Chase a rooster

Miembros del equipo:

Leonardo Contreras Martinez

Ángel Johnattan Gil Herrera

Guillermo Daniel Ortiz Aguilar

Edgar Alonso Carillo Quijano

Javier Alejandro González Gallegos

Contenido

Introduccion

Como proyecto final quedamos en hacer un videojuego en 3D con unity, el cual se llama “Chase A Rooster” (Caza un gallo en español) que como su nombre lo indica se tendrá que casar pollos que estarán esparcidos alrededor del mapa, se aplicaran técnicas como la transformaciones en 3D y animaciones en 3D, también se aplico técnicas de IA integradas en unity y recibiendo inputs del usuario para poder crear interacciones con el mapa y con los pollos, se integra una parte funcional con una linterna para poder hacer mas efectos con luz.

Consiste de 4 escenarios

- Menu principal
- Tutorial
- Primer nivel
- Segundo nivel

Donde el primer escenario consta solamente de una UI simple para iniciar el juego.

El segundo es un tutorial para explicar como funciona el juego.

El tercero es una mapa de noche con una linterna para seguir capturando pollos.

El cuarto consta de una ciudad con 20 pollos esparcidos.

Menu principal

Scripts utilizados en el menu principal

- MainMenu.CS

Assets utilizados en el menú:

- BGVideo.mp4 (Video de fondo que muestra los pollos caminando)
- Canvas para el menú principal
- GameObject de tipo MainMenu para controlar las interacciones

Como funciona

El primer escenario de main menú despliega solamente una UI que contiene el titulo del juego y 3 opciones:

- Comenzar: Este botón ejecuta la función PlayGame() en el script que inicializa el juego
- Ajustes: De momento no hace nada, pero se planeaba que este ajustara la sensibilidad del juego
- Salir: Ejecuta la función de QuitGame() para poder salir del juego

Screenshot de la escena


```
public class MainMenu : MonoBehaviour
{
    public void PlayGame()
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex   1);
    }

    public void QuitGame()
    {
        Application.Quit();
    }
}
```

codesnap.dev

Script que se encarga de cambiar de nivel.

El método PlayGame() lo que hace es llamar a la clase SceneManager de Unity para así poder cambiar de escenas en el juego, se llama al buildIndex que este lo que hace es regresar el índice de la escena actual y se le suma uno para poder acceder a la siguiente escena.

El método QuitGame() simplemente cierra la aplicación con el método Application.Quit().

Tutorial

Scripts utilizados:

- ChickenAi.cs
- InputMovement.cs
- PlayerLook.cs
- ChickenDestroyer.cs
- PlayerMotion.cs
- UIManager.cs
- UIInput.cs

Assets utilizados en el tutorial:

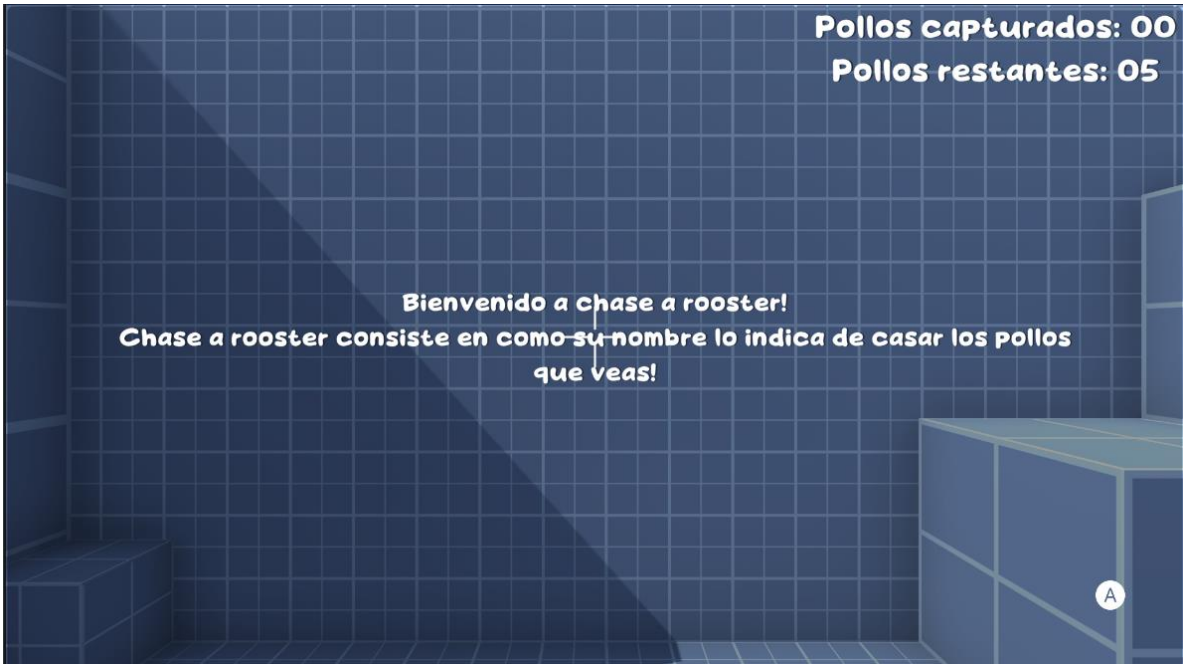
- SpeedTutor – Tutorial Scene (Mapa del tutorial)
- MeshInt free chicken Mega Toon Series (prefab del pollo)
- Sonido de caminata
- Sonido de pollo

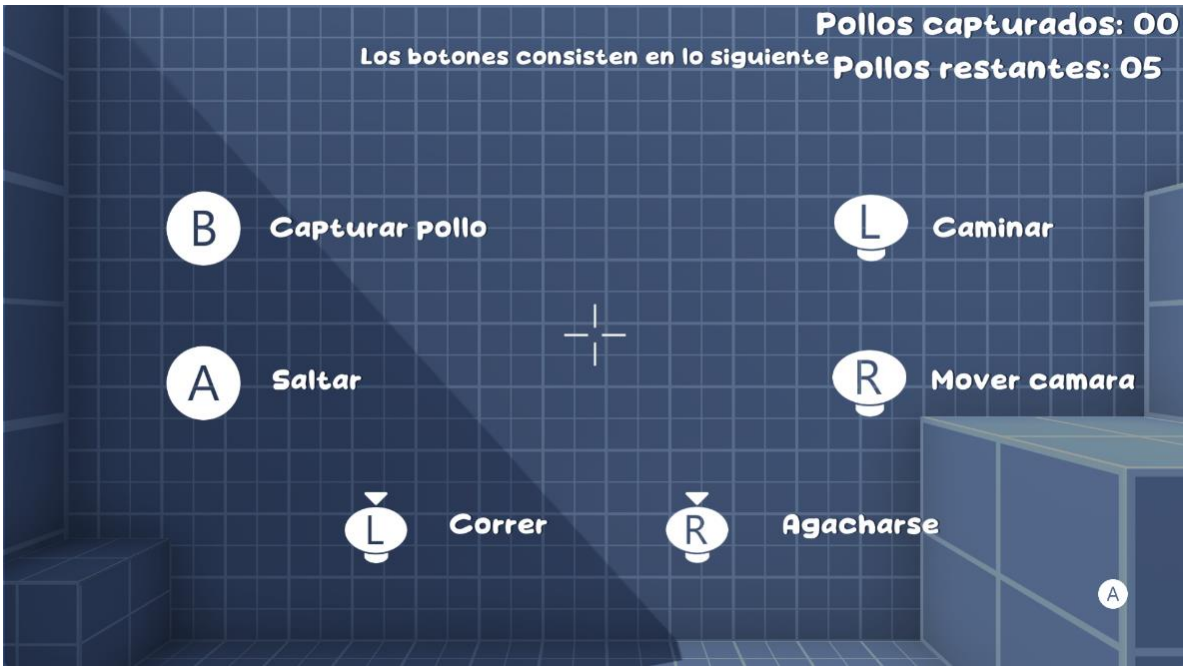
Como funciona

La escena del tutorial es una simple escena con cubos azules que tienen algunos bloques para agregar zonas altas, hay 5 pollos asignados por defecto que el jugador tendrá que atrapar haciendo uso del botón A (en Nintendo switch pro controller) y puede saltar con el boton de B, también el jugador puede correr presionando el joystick derecho y agacharse con el joystick derecho.

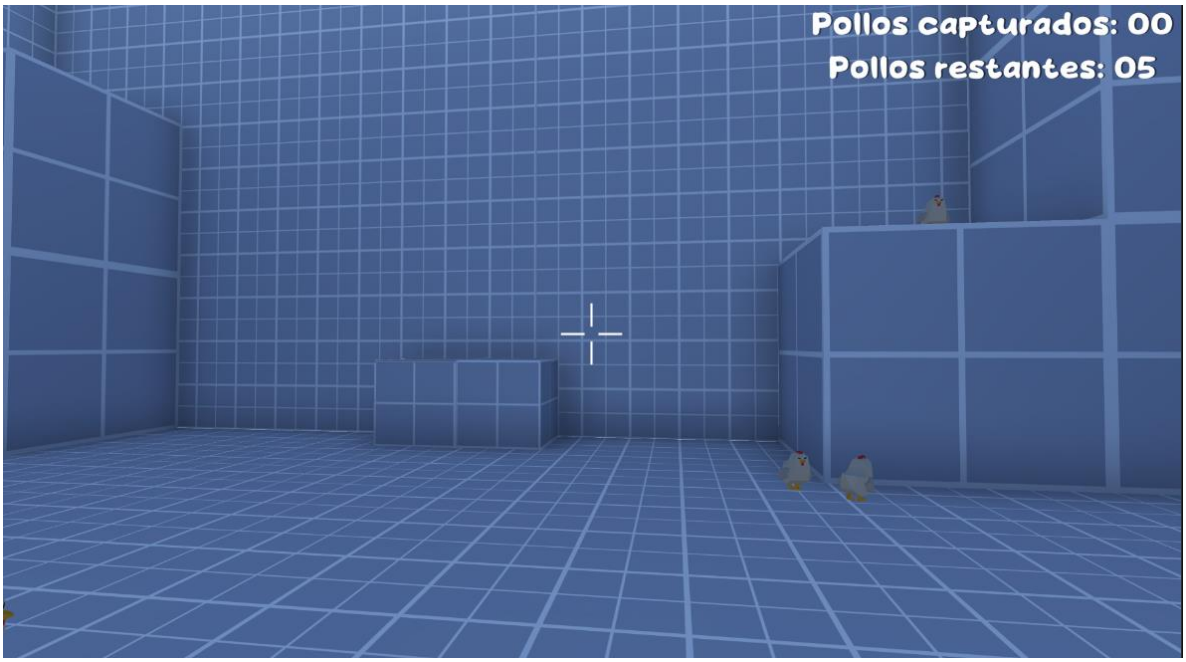
Una vez atrapados los 5 pollos se podrá acceder al siguiente nivel (escena)

Screenshot de la UI del tutorial





Screenshot del nivel



Screenshot de los scripts:

```
public class UIManager : MonoBehaviour
{
    [SerializeField] private TextMeshProUGUI[] text;
    private GameObject panel;

    public bool switchScene;

    public bool tut = true;
    public void changeGui()
    {
        if(!switchScene && tut)
            if (text[0].gameObject.activeSelf)
            {
                text[0].gameObject.SetActive(false);
                text[1].gameObject.SetActive(true);
            }
            else
            {
                panel = GameObject.FindWithTag("UIPanel");
                panel.SetActive(false);
                enableGame();
            }
        else if (switchScene)
        {
            GameManager.Instance.changeScene();
        }
    }

    public void enableGame()
    {
        GameObject player = GameObject.FindWithTag("Player");
        if (player != null)
        {
            InputMovement playerMotion = player.GetComponent<InputMovement>();
            if (playerMotion != null) playerMotion.enabled = true;
        }

        UIInput ui = GetComponent<UIInput>();
        ui.enabled = false;
    }
}
```

codesnap.dev

La clase UIManager es la que se encarga de poder observar las acciones del UI principal en el tutorial, contiene 2 booleanos que son los mas importantes ya que estos deciden si es el tutorial, donde si es el tutorial en el método ChangeGui() que se ejecuta desde la clase de UIInput este hará que los textos se puedan ciclar correctamente en el tutorial, una vez que estos terminan y ya

no hay mas textos para activar se ejecuta las líneas de código para desactivar el panel con panel.setActive(false); para desactivarlo y se ejecuta el juego enableGame() que activa el movimiento del jugador, ya que este se desactiva una vez iniciando el juego, utiliza métodos como GameObject.FindWithTag() para asegurarse que el objeto de tipo jugador exista y pueda ser activado, así como también se busca el script de movimiento para activar este mismo.

También al momento de tener activo switchScene una vez que el juego termine con el UIInput se podrá llamar el método de switchScene que se explicara con mas detalle en los siguientes scripts.

```
public class UIInput : MonoBehaviour
{
    private PlayerInput playerInput;

    private PlayerInput.OnGuiActions onGui;

    private UIManager uiManager;

    public bool isTutorial = true;

    // Start is called before the first frame update
    void Awake()
    {
        playerInput = new PlayerInput();
        onGui = playerInput.OnGui;
        uiManager = GetComponent<UIManager>();
        onGui.Continue.performed += ctx => uiManager.changeGui();
        GameObject player = GameObject.FindWithTag("Player");
        if (player != null && isTutorial)
        {
            InputMovement playerMotion = player.GetComponent<InputMovement>();
            if (playerMotion != null) playerMotion.enabled = false;
        }
    }

    private void OnEnable()
    {
        onGui.Enable();
    }

    private void OnDisable()
    {
        onGui.Disable();
    }
}
```

codesnap.dev

La clase UIInput es la que se encarga de manejar que input se manda a la escena de la interfaz de usuario, se utiliza el sistema nuevo de input de unity para poder manejar adecuadamente los controles.

En el método Awake() una vez inicializado en el tutorial se inicia con la escena de UI, por lo que significa que el script se ejecutara de inmediato, lo primero que se hace es conseguir de la clase de PlayerInput los controles correspondientes a la interfaz de usuario, donde se utiliza el método de ctx para poder llamar de forma asíncrona al método ChangeGui que se encuentra en la clase de UIManager anteriormente comentada.

Una vez que se cargan estos se llama a buscar el objeto de tipo player para desactivar el movimiento en el tutorial, de esto luego se encargara la clase de UIManager para reactivarlo.

Los métodos OnEnable() y OnDisable() son métodos genéricos para la funcionalidad de la clase.

```
public class PlayerMotion : MonoBehaviour
{
    private CharacterController controller;
    private Vector3 playerVelocity;
    public float speed = 5;
    private bool isGrounded;
    public float gravity = -9.8f;
    public float vel = 3;
    bool crouching = false;
    float crouchTimer = 1;
    bool lerpCrouch = false;
    bool sprinting = false;
    public AudioSource footSteps;
    private float ogSpeed;

    public float jumpHeight = 3;
    // Start is called before the first frame update
    void Start()
    {
        controller = GetComponent<CharacterController>();
        ogSpeed = speed;
    }

    // Update is called once per frame
    void Update()
    {
        isGrounded = controller.isGrounded;
        if (lerpCrouch)
        {
            crouchTimer += Time.deltaTime;
            float p = crouchTimer / 1;
            p *= p;
            if (crouching)
                controller.height = Mathf.Lerp(controller.height, 1, p);
            else
                controller.height = Mathf.Lerp(controller.height, 2, p);

            if (p > 1)
            {
                lerpCrouch = false;
                crouchTimer = 0;
            }
        }
    }

    //Recive el input de la clase InputManager.cs y los aplica a el controlador del personaje
    public void ProcessMove(Vector2 input)
    {
        Vector3 moveDirection = Vector3.zero;
        moveDirection.x = input.x;
        moveDirection.z = input.y;
        Vector3 moveSpeed = transform.TransformDirection(moveDirection) * speed * Time.deltaTime;
        controller.Move(moveSpeed);
        playerVelocity.y += gravity * Time.deltaTime;
        if (isGrounded && playerVelocity.y < 0)
            playerVelocity.y = -2f;
        controller.Move(playerVelocity * Time.deltaTime);
        if (moveSpeed.magnitude > 0.01 && playerVelocity.y == -2f) footSteps.enabled = true;
        else footSteps.enabled = false;
    }

    public void Jump()
    {
        if (isGrounded)
        {
            playerVelocity.y = Mathf.Sqrt(jumpHeight * -3 * gravity);
        }
    }

    public void Crouch()
    {
        crouching = !crouching;
        crouchTimer = 0;
        lerpCrouch = true;
    }

    public void Sprint()
    {
        sprinting = !sprinting;
        if (sprinting)
            speed = ogSpeed + vel;
        else
            speed = ogSpeed;
    }

    public void Destroy()
    {
        Debug.Log("Hola");
        // Find the ChickenDestroyer script on the player GameObject
        ChickenDestroyer chickenDestroyer = GetComponent<ChickenDestroyer>();
        if (chickenDestroyer != null)
        {
            // Trigger the chicken destruction logic
            chickenDestroyer.DestroyChicken();
        }
    }

    public float getPlayerSpeed()
    {
        return speed;
    }

    public void setPlayerSpeed(float speed)
    {
        this.speed = speed;
    }
}
```

La clase PlayerMotion contiene todos los métodos necesarios para los movimientos del jugador, contiene varios atributos que se encargan principalmente de la velocidad del jugador y algunos para facilitar las conversiones entre vectores y magnitud de los jugadores.

El método Start() simplemente contiene la asignacion del objeto tipo CharacterController, también se asigna la velocidad original para poder ser utilizada nuevamente.

El método Update() se encarga de verificar y hacer que el jugador se pueda agachar, utilizando funciones de Mathf para poder transformar el jugador hacia una posición mas abajo para simular el efecto de agachado.

El método ProcessMove() recibe input directamente de la clase Input, aquí se asigna el input proveniente del control y asigna los valores a los ejes en 2D para el movimiento del jugador, se utiliza el método de controller.Move() para recibir un vector y saber hacia donde se moverá el jugador, se utilizan cálculos para poder sacar una velocidad constante con los frames del juego utilizando Time.deltaTime() y también se utilizan métodos como el IsGrounded() para poder saber si el jugador se encuentra en el piso, si es así este podra saltar haciendo una push forcé hacia el eje Y del jugador, de forma contraria no podrá hacerlo.

El método Jump() hace que el jugador pueda saltar.

El método Crouch() hace que el jugador pueda agacharse.

El método Sprint() asigna la velocidad del jugador dependiendo si esta corriendo o no.

El método destroy se comunica directamente con el script de ChickenDestroyer() para poder destruir los pollos una vez se presione el botón correspondiente, se detallar mas en los siguientes scripts.

GetPlayerSpeed() y SetPlayerSpeed() son metodos getter y setters para poder hacer funciones mas fáciles que correspondan con la velocidad del jugador.

```
public class ChickenDestroyer : MonoBehaviour
{
    public float raycastRadius = 30f; // The radius of the raycast from the camera center
    public float detectionDistance = 2f; // The distance at which the chicken can be detected
    public LayerMask chickenLayer; // LayerMask for the chicken GameObjects
    public string chickenLayerName = "ChickenLayer";

    private Camera cam;

    private void Awake()
    {
        cam = GetComponentInChildren<Camera>();
        chickenLayer = LayerMask.GetMask(chickenLayerName);
    }

    public void DestroyChicken()
    {
        // Create a ray from the camera center
        Ray ray = cam.ViewportPointToRay(new Vector3(0.5f, 0.5f, 0f));

        RaycastHit hit;
        // Cast a ray and check if it hits anything on the chicken layer
        if (Physics.Raycast(ray, out hit, detectionDistance, chickenLayer))
        {
            // If the chicken is hit, destroy the chicken GameObject
            Destroy(hit.collider.gameObject);
            updateChicken();
        }
    }

    void updateChicken()
    {
        GameManager.Instance.updateChickens();
    }
}
```

codesnap.dev

Este es un script en C# llamado "ChickenDestroyer" que se encarga de detectar y destruir pollos en el juego en Unity. A continuación, explicaré cada parte del código:

1. Variables públicas:

- raycastRadius: Es el radio del rayo que se emite desde el centro de la cámara. Este rayo se utiliza para detectar pollos.
- detectionDistance: Es la distancia a la que el pollo puede ser detectado por el rayo.
- chickenLayer: Es una máscara de capa (LayerMask) utilizada para identificar los GameObjects de pollos en el juego.
- chickenLayerName: Es el nombre de la capa en la que se encuentran los GameObjects de pollos.

2. Variable privada:

- cam: Es una referencia a la cámara que se encuentra como hijo del GameObject al que está adjunto este script.

3. Método Awake():

- Este método se ejecuta al inicio del juego antes de Start().
- Se obtiene la referencia a la cámara que se encuentra como hijo del GameObject en el que está adjunto el script, utilizando GetComponentInChildren<Camera>().
- Se establece la variable chickenLayer utilizando el nombre de la capa de pollos (chickenLayerName) mediante LayerMask.GetMask(chickenLayerName).

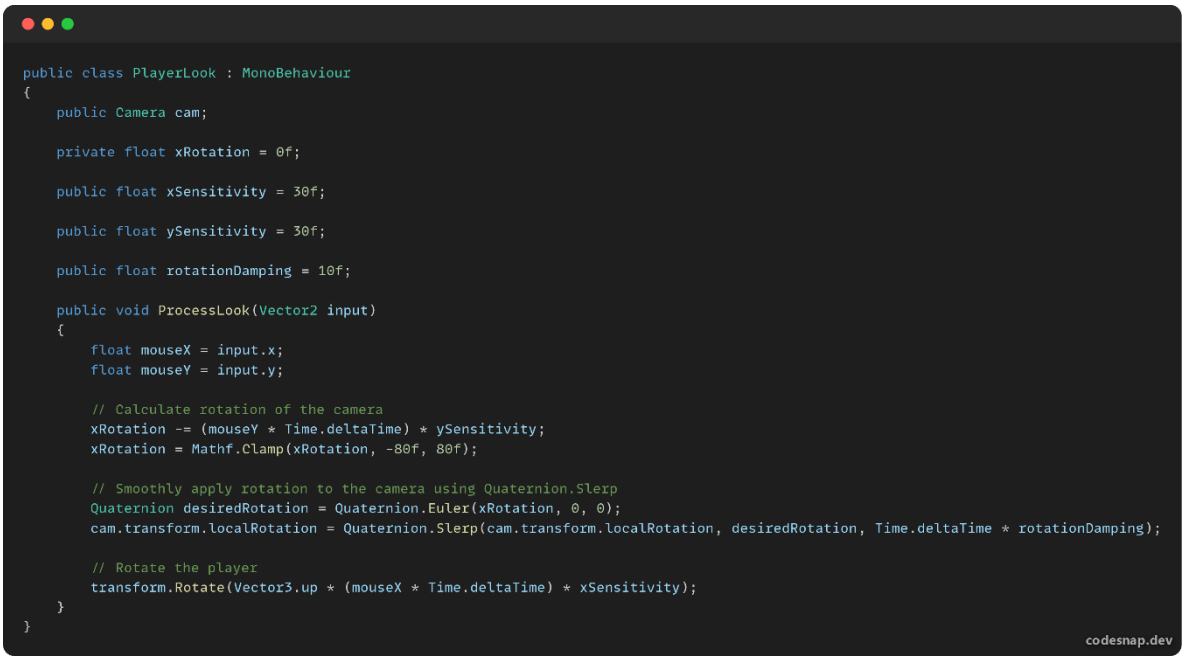
4. Método DestroyChicken():

- Este método se llama para destruir un pollo detectado.
- Se crea un rayo que se emite desde el centro de la cámara (cam) utilizando ViewportPointToRay, que toma una posición de viewport (0.5f, 0.5f) para que el rayo salga desde el centro de la pantalla.
- Se lanza un rayo desde la cámara en la dirección del centro de la vista (ray) y se comprueba si golpea algo en la capa de pollos (chickenLayer).
- Si el rayo golpea un pollo, se destruye el GameObject del pollo mediante Destroy(hit.collider.gameObject).
- Después de destruir el pollo, se llama al método updateChicken().

5. Método updateChicken():

- Este método llama al método updateChickens() de la instancia del GameManager mediante GameManager.Instance.updateChickens().
- Esto permite actualizar la cantidad de pollos capturados y restantes en el juego, ya que el GameManager controla el recuento de pollos.

En resumen, este script se encarga de detectar y destruir pollos en el juego cuando el jugador interactúa con ellos. Una vez que un pollo es destruido, se actualiza la cantidad de pollos capturados y restantes utilizando el GameManager para reflejar el cambio en el juego.



```
public class PlayerLook : MonoBehaviour
{
    public Camera cam;

    private float xRotation = 0f;

    public float xSensitivity = 30f;

    public float ySensitivity = 30f;

    public float rotationDamping = 10f;

    public void ProcessLook(Vector2 input)
    {
        float mouseX = input.x;
        float mouseY = input.y;

        // Calculate rotation of the camera
        xRotation -= (mouseY * Time.deltaTime) * ySensitivity;
        xRotation = Mathf.Clamp(xRotation, -80f, 80f);

        // Smoothly apply rotation to the camera using Quaternion.Slerp
        Quaternion desiredRotation = Quaternion.Euler(xRotation, 0, 0);
        cam.transform.localRotation = Quaternion.Slerp(cam.transform.localRotation, desiredRotation, Time.deltaTime * rotationDamping);

        // Rotate the player
        transform.Rotate(Vector3.up * (mouseX * Time.deltaTime) * xSensitivity);
    }
}
```


Este es un script en C# llamado "PlayerLook" que controla la rotación de la cámara y el jugador en primera persona en Unity. A continuación, se explicara el codigo:

1. Variables públicas:

- cam: Es una referencia a la cámara en el juego que se utiliza para obtener la rotación actual de la cámara y aplicar la rotación deseada.
- xSensitivity: Es la sensibilidad de rotación horizontal (eje X) de la cámara.
- ySensitivity: Es la sensibilidad de rotación vertical (eje Y) de la cámara.
- rotationDamping: Es la cantidad de suavizado que se aplicará a la rotación de la cámara al seguir el movimiento del ratón.

2. Variable privada:

- xRotation: Es una variable para almacenar el ángulo de rotación vertical de la cámara en grados.

3. Método ProcessLook(Vector2 input):

- Este método se encarga de procesar el movimiento del ratón para rotar la cámara y el jugador.
- input: Es un Vector2 que contiene la información de entrada del ratón (movimiento horizontal y vertical).

4. Procesamiento de la rotación vertical de la cámara:

- La rotación vertical (xRotation) se calcula restando el movimiento vertical del ratón (mouseY) multiplicado por la sensibilidad vertical (ySensitivity) y multiplicado por

Time.deltaTime para hacerlo independiente del framerate. El resultado se limita entre -80 y 80 grados para evitar que la cámara gire completamente hacia arriba o abajo.

- Luego, se utiliza Quaternion.Euler para crear una rotación deseada en el eje X de la cámara basado en xRotation. Este valor se almacena en desiredRotation.

5. Aplicación suave de la rotación de la cámara:

- Se utiliza Quaternion.Slerp para aplicar una rotación suave a la cámara desde su rotación actual (cam.transform.localRotation) hacia la rotación deseada (desiredRotation). El parámetro rotationDamping controla la cantidad de suavizado aplicado. Esto se hace para que la rotación de la cámara se sienta más natural y no sea brusca.

6. Rotación del jugador:

- Se rota el jugador en el eje Y (horizontal) mediante transform.Rotate, utilizando el movimiento horizontal del ratón (mouseX) multiplicado por la sensibilidad horizontal (xSensitivity) y multiplicado por Time.deltaTime para que la rotación sea independiente del framerate.

Este script se utiliza para controlar la rotación de la cámara y el jugador en una perspectiva de primera persona en Unity, y es comúnmente usado en conjunto con un script de movimiento del jugador para lograr una experiencia de movimiento más inmersiva y realista.

```
public class RandomMovement : MonoBehaviour
{
    public NavMeshAgent agent;
    public float range; //radius of sphere
    public float stoppingDistance = 0.5f; // Adjust this value as needed
    private Animator animator;
    public AudioSource chickenSound;
    public AudioSource chickenWalk;

    public Transform centrePoint; //centre of the area the agent wants to move around in

    // New variables for player detection and running away
    public LayerMask playerLayer; // LayerMask to define the player's layer
    public float detectionDistance = 10.0f; // Distance within which the chicken detects the player
    public float runAwayDistance = 5.0f; // Distance from the player at which the chicken starts running
    public float runSpeed = 5.0f; // Speed at which the chicken runs when escaping
    private float startTime;
    private float timePassed;
    private bool isPlayingSound = false;
    private bool isGrounded; // Variable to check if the chicken is grounded
    private float gravity = -9.81f; // Gravity value

    void Start()
    {
        agent = GetComponent<NavMeshAgent>();
        agent.stoppingDistance = stoppingDistance;
        animator = GetComponent<Animator>();
        startTime = Time.time;
        chickenSound.enabled = false;
    }

    void Update()
    {
        float timePassed = Time.time - startTime;

        // Check if 5 seconds have passed and the sound is not already playing
        if (timePassed >= 4f && !isPlayingSound)
        {
            // Play the sound
            chickenSound.enabled = true;
            isPlayingSound = true;
        }

        // Check if 1 second has passed and the sound is playing
        if (timePassed >= 5f && isPlayingSound)
        {
            // Stop the sound
            chickenSound.enabled = false;
            isPlayingSound = false;

            // Reset the timer for the next sound
            startTime = Time.time;
        }

        // Check for player detection
        Collider[] detectedPlayers = Physics.OverlapSphere(transform.position, detectionDistance, playerLayer);

        if (detectedPlayers.Length > 0)
        {
            // Player detected, run away from the player
            Vector3 playerDirection = transform.position - detectedPlayers[0].transform.position;
            Vector3 targetPoint = transform.position + playerDirection.normalized * runAwayDistance;
            agent.SetDestination(targetPoint);
            agent.speed = runSpeed;
        }
        else
        {
            if (agent.remainingDistance <= agent.stoppingDistance) //done with path
            {
                Vector3 point;
                if (RandomPoint(centrePoint.position, range, out point)) //pass in our centre point and radius of area
                {
                    Debug.DrawRay(point, Vector3.up, Color.blue, 1.0f); //so you can see with gizmos
                    agent.SetDestination(point);
                }
            }
        }
        float speed = agent.velocity.magnitude;
        if (speed > 0.01)
        {
            animator.SetBool("Walk", true);
            chickenWalk.enabled = true;
        }
        else if (Mathf.Approximately(speed, runSpeed))
        {
            chickenWalk.enabled = true;
            animator.SetBool("Run", true);
        }
        else
        {
            animator.SetBool("Walk", false);
            animator.SetBool("Run", false);
            chickenWalk.enabled = false;
        }
    }

    void FixedUpdate()
    {
        // Check if the chicken is grounded using a raycast
        CheckGrounded();

        // Apply gravity to the chicken
        ApplyGravity();
    }

    void CheckGrounded()
    {
        // Cast a ray from slightly above the chicken's position to check if it's grounded
        isGrounded = Physics.Raycast(transform.position + Vector3.up * 0.1f, Vector3.down, 0.2f);
    }

    void ApplyGravity()
    {
        if (!isGrounded)
        {
            // Calculate the new position with gravity
            Vector3 newPosition = transform.position + Vector3.up * gravity * Time.fixedDeltaTime;

            // Check if the new position is above the ground using a raycast
            RaycastHit hit;
            if (Physics.Raycast(newPosition, Vector3.down, out hit, 0.2f))
            {
                // If the new position is above the ground, set the position to the hit point
                newPosition.y = hit.point.y;
            }

            // Set the chicken's position to the new position
            transform.position = newPosition;
        }
    }

    bool RandomPoint(Vector3 center, float range, out Vector3 result)
    {
        Vector3 randomPoint = center + Random.insideUnitSphere * range; //random point in a sphere
        NavMeshHit hit;
        if (NavMesh.SamplePosition(randomPoint, out hit, 1.0f, NavMesh.AllAreas))
        {
            //the 1.0f is the max distance from the random point to a point on the navmesh, might want to increase if range is big
            //or add a for loop like in the documentation
            result = hit.position;
            return true;
        }

        result = Vector3.zero;
        return false;
    }
}
```

ste es un script en C# llamado "RandomMovement" que controla el movimiento de un objeto (en este caso, un pollo) en un entorno de juego utilizando el componente "NavMeshAgent" de Unity. A continuación, te explicaré qué hace cada parte del código:

1. Variables públicas:

- agent: Es una referencia al componente NavMeshAgent que se utiliza para controlar el movimiento del objeto.
- range: Es el radio de la esfera dentro de la cual el objeto puede moverse alrededor de un punto central (centrePoint).
- stoppingDistance: Es la distancia a la que el objeto se detendrá cuando alcance su destino.
- animator: Es una referencia al componente Animator, que se utiliza para controlar las animaciones del objeto (caminar y correr).
- chickenSound: Es una referencia al componente AudioSource que reproduce un sonido del pollo.
- chickenWalk: Es una referencia al componente AudioSource que reproduce el sonido de caminar del pollo.
- centrePoint: Es un Transform que representa el centro del área en la que el objeto se moverá al azar.

2. Variables para la detección del jugador y huir:

- playerLayer: Es una LayerMask que define la capa del jugador en el juego.
- detectionDistance: Es la distancia dentro de la cual el pollo detectará al jugador.
- runAwayDistance: Es la distancia a la que el pollo comenzará a huir del jugador.
- runSpeed: Es la velocidad a la que el pollo corre cuando está huyendo.

3. Variables privadas:

- startTime: Es el tiempo en segundos desde el inicio del juego.
- timePassed: Es el tiempo en segundos que ha pasado desde que comenzó el sonido del pollo.
- isPlayingSound: Es una bandera para verificar si el sonido del pollo está reproduciéndose actualmente.
- isGrounded: Es una bandera que indica si el pollo está en el suelo.
- gravity: Es el valor de la gravedad aplicado al pollo cuando no está en el suelo.

4. Método Start():

- Se obtienen las referencias a los componentes necesarios (NavMeshAgent y Animator).

- Se configura el tiempo de inicio (startTime) y se desactiva el componente chickenSound para que no se reproduzca al principio.

5. Método Update():

- Se comprueba si ha pasado un cierto tiempo (timePassed) para reproducir y detener el sonido del pollo en intervalos regulares.
- Se detecta la presencia del jugador mediante el uso de Physics.OverlapSphere, y si se encuentra un jugador, el pollo huirá de él estableciendo un nuevo destino usando SetDestination.
- Si no se detecta un jugador, el pollo se moverá al azar dentro del área definida por el centro (centrePoint) utilizando el método RandomPoint.
- Se actualiza la animación del pollo según su velocidad de movimiento (speed) y se activa/desactiva el sonido de caminar (chickenWalk) según si el pollo se está moviendo.

6. Método FixedUpdate():

- CheckGrounded(): Utiliza un raycast para verificar si el pollo está en el suelo y actualiza la variable isGrounded.
- ApplyGravity(): Aplica la gravedad al pollo si no está en el suelo, ajustando su posición en función del tiempo de actualización (Time.fixedDeltaTime) y un raycast para mantenerlo en el suelo.

7. Método RandomPoint(Vector3 center, float range, out Vector3 result):

- Genera un punto aleatorio dentro de una esfera alrededor del centro (centrePoint).
- Utiliza NavMesh.SamplePosition para encontrar la posición más cercana en la superficie del NavMesh a partir del punto aleatorio generado. Esto asegura que el punto esté dentro del área caminable del NavMesh.

Este script permite que el pollo se mueva al azar dentro de un área específica y huya del jugador si está dentro de una cierta distancia. Además, tiene una animación de caminar y correr, así como la capacidad de reproducir sonidos para dar más vida al personaje en el juego.

```
public class GameManager : MonoBehaviour
{
    public static GameManager Instance { get; private set; }
    public GameObject winPanel;
    [SerializeField] private TextMeshProUGUI chickenUpdater;
    [SerializeField] private float restingChicken;
    public string sceneName;
    public bool ENDGAME = false;

    private float capturedChickens;
    private void Awake()
    {
        if (Instance == null)
            Instance = this;
        else Debug.Log("ola");
    }


    public void updateChickens()
    {
        chickenUpdater.text = $"Pollos capturados: {capturedChickens+= 1} \n" +
                              $"Pollos restantes: {restingChicken -= 1}";
        if (restingChicken == 0)
            endGame();
    }

    void endGame()
    {
        winPanel.gameObject.SetActive(true);
        GameObject player = GameObject.FindWithTag("Player");
        if (player != null && !ENDGAME)
        {
            InputMovement playerMotion = player.GetComponent<InputMovement>();
            if (playerMotion != null) playerMotion.enabled = false;
        }
    }

    public void changeScene()
    {
        if(!ENDGAME)
            SceneManager.LoadScene(sceneName);
        else if (ENDGAME && restingChicken == 0)
            Application.Quit();
    }
}
```

codesnap.dev

Este script en C# llamado "GameManager" es el administrador de juego en Unity. A continuación, explicaré cada parte del código:

1. Variables y propiedades públicas:

- Instance: Es una propiedad estática que representa la única instancia del GameManager en el juego, implementada como un patrón singleton para asegurar que solo haya una instancia del GameManager en todo momento.
- winPanel: Es una referencia a un GameObject que representa el panel que se muestra cuando se gana el juego.
- chickenUpdater: Es una referencia a un componente TextMeshProUGUI utilizado para mostrar información sobre el número de pollos capturados y restantes.
- restingChicken: Es una variable que representa la cantidad de pollos restantes en el juego.
- sceneName: Es una cadena que almacena el nombre de la escena a cargar cuando se reinicia el juego.
- ENDGAME: Es una bandera booleana que indica si el juego ha terminado.

2. Variables privadas:

- capturedChickens: Es una variable que representa la cantidad de pollos capturados hasta el momento.

3. Método Awake():

- Es un método de inicialización que se ejecuta antes de Start().
- Aquí se implementa el patrón singleton para asegurarse de que solo exista una instancia del GameManager en el juego. Si no hay una instancia existente, se establece Instance como esta instancia. Si ya existe una instancia, se muestra un mensaje de advertencia.

4. Método updateChickens():

- Es llamado cuando se captura un pollo en el juego.
- Incrementa capturedChickens en 1 y decrementa restingChicken en 1.
- Actualiza el componente chickenUpdater.text para reflejar la cantidad actualizada de pollos capturados y restantes.
- Si restingChicken llega a 0, llama al método endGame().

5. Método endGame():

- Activa el panel winPanel para mostrar al jugador que ha ganado el juego.
- Busca el GameObject que tiene la etiqueta "Player" (jugador) y deshabilita el componente InputMovement asociado (un script que maneja el movimiento del jugador) para evitar que el jugador se mueva después de ganar el juego.

6. Método changeScene():

- Es llamado cuando se necesita cambiar de escena (ya sea reiniciando el juego o saliendo del mismo).
- Si el juego no ha terminado (ENDGAME es falso), carga la escena cuyo nombre está almacenado en sceneName utilizando SceneManager.LoadScene(sceneName).
- Si el juego ha terminado (ENDGAME es verdadero) y no quedan pollos restantes (restingChicken es 0), sale de la aplicación utilizando Application.Quit().

Este script es parte esencial del funcionamiento del juego y se encarga de controlar aspectos importantes, como la cantidad de pollos capturados y restantes, el cambio de escenas, el manejo del panel de victoria y la lógica para evitar el movimiento del jugador después de ganar el juego.

Primer nivel

Scripts utilizados:

- Los mismos del tutorial
- ChickenSpawner.cs

Assets utilizados:

- Mismos que el tutorial (menos el mapa)
- Mapa low poly de un bosque
- Slenderman (prefab)

Como funciona

El primer nivel es en un mapa de un bosque en low poly style, este nivel contiene 10 pollos que el jugador tendrá que capturar los 10 pollos para poder pasar hacia el siguiente nivel, de momento el mapa parece funcionar normalmente pero en caso de que un pollo este buggeado el jugador podrá utilizar el botón de Start para saltar de nivel.

También se simula una linterna utilizando el objeto de spotlight para simular una linterna utilizando también una cookie para aumentar mas la inmersión de la linterna.

Screenshot del mapa:

```
public class ChickenSpawner : MonoBehaviour
{
    public GameObject chickenPrefab;
    public int numberOfChickens = 10;
    public float spawnRadius = 20f;
    public float obstacleCheckRadius = 5f;

    void Start()
    {
        SpawnChickens();
    }

    public void SpawnChickens()
    {
        for (int i = 0; i < numberOfChickens; i++)
        {
            Vector3 randomSpawnPoint = GetRandomPointOnNavMesh(transform.position, spawnRadius);
            GameObject newChicken = Instantiate(chickenPrefab, randomSpawnPoint, Quaternion.identity);
            newChicken.GetComponent<NavMeshAgent>().enabled = true;
        }
    }

    private Vector3 GetRandomPointOnNavMesh(Vector3 center, float radius)
    {
        int maxAttempts = 10;
        int attempts = 0;

        while (attempts < maxAttempts)
        {
            Vector3 randomPoint = center + Random.insideUnitSphere * radius;
            NavMeshHit hit;
            if (NavMesh.SamplePosition(randomPoint, out hit, radius, NavMesh.AllAreas))
            {
                // Check for obstacles around the random point
                Collider[] colliders = Physics.OverlapSphere(randomPoint, obstacleCheckRadius);
                bool foundObstacle = false;
                foreach (Collider collider in colliders)
                {
                    if (collider.gameObject.CompareTag("Obstacle"))
                    {
                        foundObstacle = true;
                        break;
                    }
                }

                if (!foundObstacle)
                {
                    return hit.position;
                }
            }

            attempts++;
        }

        // If no valid point is found after maxAttempts, return center position
        return center;
    }
}
```

codesnap.dev

Este es un script en C# llamado "ChickenSpawner" que se utiliza para generar pollos en posiciones aleatorias dentro del NavMesh en Unity. A continuación se explica cada parte del código:

1. Variables públicas:

- chickenPrefab: Es el prefab (modelo) del pollo que se generará en el juego.
- numberOfChickens: Es la cantidad de pollos que se generarán.
- spawnRadius: Es el radio dentro del cual se generará cada pollo.
- obstacleCheckRadius: Es el radio utilizado para verificar si hay obstáculos alrededor del punto de generación del pollo.

2. Método Start():

- Este método se ejecuta al inicio del juego y llama al método SpawnChickens() para generar los pollos.

3. Método SpawnChickens():

- Este método se encarga de generar los pollos en posiciones aleatorias dentro del NavMesh.
- Utiliza un bucle for para generar el número de pollos especificado (numberOfChickens).
- Para cada pollo, obtiene una posición de generación aleatoria en el NavMesh utilizando el método GetRandomPointOnNavMesh.
- Luego, instancia el pollo (chickenPrefab) en la posición aleatoria generada y activa su componente NavMeshAgent para que pueda moverse usando el NavMesh.

4. Método GetRandomPointOnNavMesh(Vector3 center, float radius):

- Este método devuelve una posición aleatoria dentro del NavMesh alrededor del centro especificado.
- Utiliza un bucle while con un número máximo de intentos (maxAttempts) para obtener un punto aleatorio que esté en el NavMesh y no tenga obstáculos alrededor.
- En cada intento, genera un punto aleatorio dentro de una esfera alrededor del centro utilizando Random.insideUnitSphere y lo evalúa con NavMesh.SamplePosition para verificar si está en el NavMesh y obtener el punto más cercano en el NavMesh (representado por NavMeshHit hit).
- Después, verifica si hay obstáculos alrededor del punto aleatorio utilizando Physics.OverlapSphere, buscando colisiones con objetos etiquetados como "Obstacle". Si encuentra algún obstáculo, descarta ese punto y continúa buscando otro punto.
- Si se encuentra un punto válido sin obstáculos después de los intentos permitidos (maxAttempts), se devuelve ese punto. Si no se encuentra un punto válido, se devuelve el centro especificado.

En resumen, este script se encarga de generar una cantidad determinada de pollos en posiciones aleatorias dentro del NavMesh en el juego. Antes de generar un pollo, se verifica que la posición

aleatoria esté en el NavMesh y no tenga obstáculos cercanos para asegurar que los pollos se generen en áreas accesibles y seguras para el juego.

Segundo (ultimo) nivel

Scripts utilizados

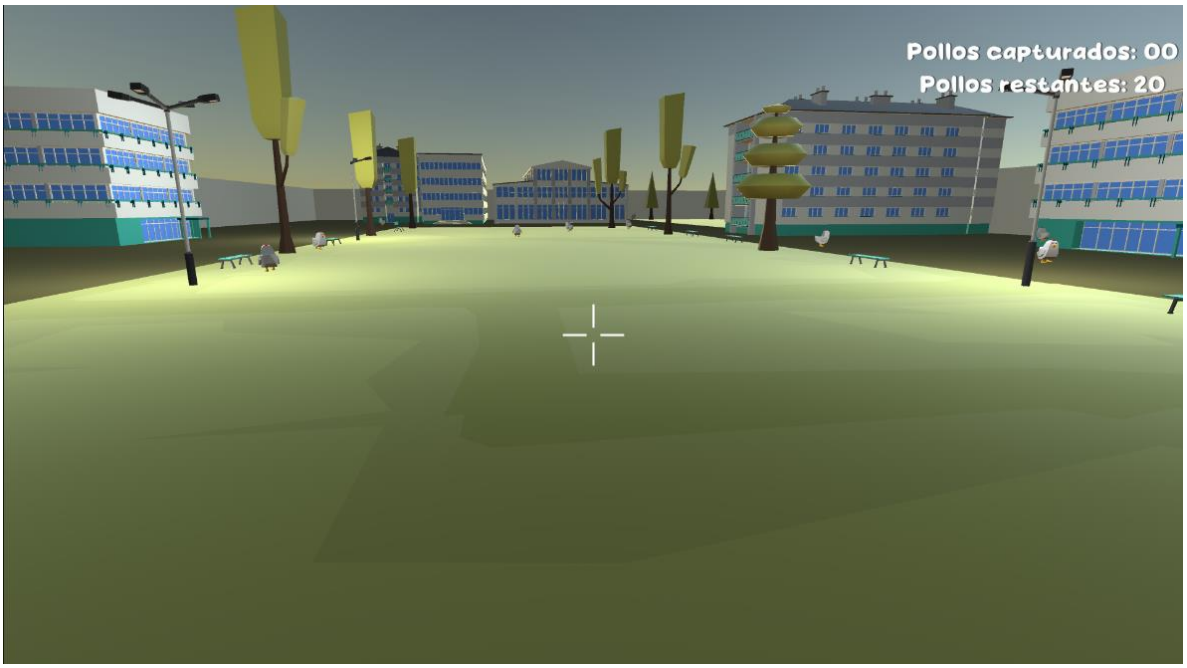
- Los mismos que el nivel pasado

Assets utilizados

- Mismos que el nivel pasado (Excepto el mapa)
- Mapa de ciudad Low Poly

Como funciona

El ultimo nivel contiene 20 pollos que el jugador cazara, con la diferencia que una vez que se capturen los 20 pollos se ejecutara el script de UIManager en vez de cambiar de escena se activara con el booleano de ENDGAME que se acabo el juego dándole al usuario la opción de finalizar el juego.



Mapa del nivel

Captura de fin del juego