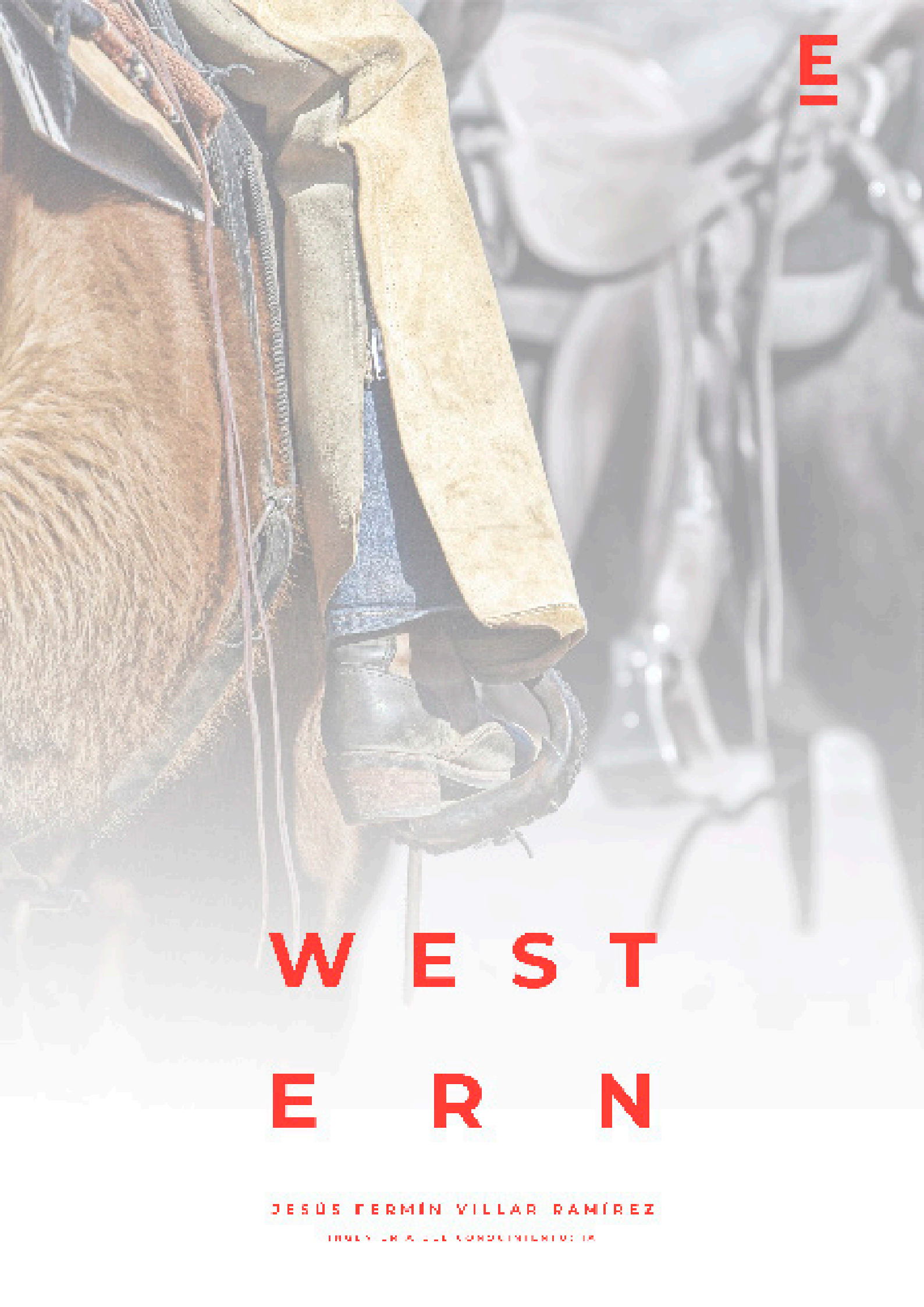

WESTERN

JESÚS FERMÍN VILLAR RAMÍREZ

TABLA CONTENIDOS

TÍTULO: MEMORIA SEGUNDA ENTREGA WESTERN
AUTOR: JESÚS FERMÍN VILLAR RAMÍREZ
ASIGNATURA: INGENIERÍA DEL CONOCIMIENTO: IA
DOCENTE: LUIS PEÑA SÁNCHEZ
GRADO: DISEÑO Y DESARROLLO DE VIDEOJUEGOS, ESNE
FECHA: ABRIL 2019
FOTOGRAFÍA DE PORTADA: Fuente: https://www.pexels.com/photo/person-wearing-cowboy-boots-riding-on-horse-51130/

INTRODUC-CIÓN

Este documento corresponde a la memoria de la segunda entrega de la práctica “Western”, de la asignatura “Ingeniería del Conocimiento: IA” impartida por Luis Peña Sánchez en ESNE.
En dicho documento se define el algoritmo que se ha utilizado así como el modelo de datos actual al haber incorporado dicho algortimo.

En Madrid, abril de 2019
Jesús Fermín Villar Ramírez
Diseño y Desarrollo de Videojuegos

ALGORITMO UTILIZADO

A la hora de implementar un algoritmo se ha usado la metodología del Hill Climbing o Ascenso de Colinas. De esta forma, cada nodo está constituido por cada cobertura.
La expansión de los nodos se realiza gracias a la lista de los nodos conocidos por el enemigo, es decir, todos aquellos nodos que ha visto el enemigo desde que ha empezado el juego, evaluando en cada uno de ellos su valor heurístico.

Esta heurística depende de los siguientes parámetros:

(alpha * distancia al nodo) + (beta * índice de ocupación del nodo) + (gamma * índice de visibilidad del nodo por parte del player)

Este último parámetro solo se aplica a la heurística si el enemigo ve al jugador.

Este método de calcular la heurística garantiza que se tenga en cuenta, a la hora de elegir a qué nodo moverse, la distancia a la que está dicho nodo, si ese nodo está ocupado ya por otro enemigo (este hecho es para evitar que se aglutinen varios enemigos en el mismo nodo) y si ese nodo es visto por el jugador.

Ya que tiene diferentes niveles o capas este sistema, se ha incorporado un controlador que sirve de árbol de comportamiento. Este árbol de comportamiento evalúa varios estados: la visibilidad hacia el jugador, el estado de cobertura del enemigo y el estado de movimiento.
Además se tiene por otro lado el sistema de percepción planteado en la entrega anterior. Este sistema de percepción actualiza valores que son consultados en la máquina de estados a la hora de elaborar una decisión de actuación.

MODELO DE DATOS

```
public class EnemyController : MonoBehaviour
{
    public float seconds_between_cast = 1f;
    public bool visible_for_player { get; set; }
    public Node current_node;

    List<Node>          known_nodes          = new List<Node>();
    MovementController Locomotion;
    EnemyView           this_enemy_view;
    ShotController      this_enemy_shot_controller;

    enum PerceptionSystem { seing_player, searching_player        }
    enum CoverState       { half_covered, full_covered, uncovered }
    enum MovementStates   { moving, at_destination, stoped        }

    PerceptionSystem state;
    CoverState       this_enemy_cover_state;
    MovementStates   this_enemy_movement_state;

    void Start()...
    void Update()...

    /// <summary> Adds the given node to the list of known nodes.
    public void AddNode(Node node)...
    /// <summary> Method called when the enemy is at the destination
    public void AtDestination() ...
    /// <summary> Method that updates the flag of seeing the player
    public void SeeingPlayer()...
    /// <summary> Method that updates the flag of seeing the player.
    public void NotSeeingPlayer()...
    /// <summary> Updates the current node ocupation.
    public void UpdatesNodeOcupation(int amount)...
    /// <summary> Method that makes the decision of the enemy behaviour
    public void MakeADecision()...

    /// <summary> Courutine that calls the method "CastTheRay" of the view system ea ...
    IEnumerator CastingCoroutine(float seconds)...
```

```
/// <summary> Script that manages the movement of the enemies This script should ...
public class MovementController : MonoBehaviour {


    /// <summary> Movement speed
    public float mov_speed;
    /// <summary> The angle to rotate on each rotation
    public float angles_to_rotate;

    /// <summary> Final position X coordinate
    float x_position_to_check;
    /// <summary> Final position Z coordinate
    float z_position_to_check;
    /// <summary> The direction of the movement
    Vector3 direction;
    /// <summary> Reference to the controller of this enemy
    EnemyController this_enemy_controller;
    /// <summary> If this enemy is able to move.
    bool able_to_move;
    Node from_node = null;


    /// <summary> Rotates the enemy
    public void Rotate()...
    /// <summary> Stop the movement.
    public void Stop()...
    /// <summary> Resumes the movement.
    public void ResumeMovement()...
    /// <summary> Gets the new destination.
    public void GetNewDestination(List<Node> nodes)...

    /// <summary> At the first frame
    private void Start()...
    /// <summary> On each frame
    private void Update()...
    /// <summary> Method that set the destination position
    private void MoveToCover(Vector3 target_position)...
```

```
/// <summary> Script that manages the enemy vision This script should be attache ...
public class EnemyView : MonoBehaviour {

    [Space]
    [Header("Angulaciones del rayo")]
    /// <summary> The negative angle of view (left to the view axis)
    public float min_angle_ray;
    /// <summary> The positive angle of view (right to the view axis)
    public float max_angle_ray;
    /// <summary> The angle between each casted ray
    public float angle_between_rays;

    [Space]
    [Header("Orígenes del rayo")]
    /// <summary> The origin of the eye rays
    public Transform eye_ray_origin;
    /// <summary> The origin of the foot rays
    public Transform foot_ray_origin;

    /// <summary> If this enemy is being seen by the player
    private bool im_visible;
    /// <summary> The controller of this enemy.
    EnemyController controller_of_this_enemy;



    /// <summary> Method that casts the rays
    public void CastTheRay()...
    /// <summary> At the first frame
    void Start()    ...
```

```
public class Node : MonoBehaviour
{
    public bool    GetVisibleByPlayer ...
    public int     GetOcupation       ...
    public Vector3 GetPosition        ...
    public Node    GetNode            ...

    private float _vision_mulitplier      = 1f;
    private float _disspersion_multiplier = 1f;
    private float _distance_multiplier    = 1f;

    Cover this_cover;

    private void Start()...

    /// <summary> Method to know the Fstar value of a node
    public float GetFStar(bool enemy_see_player, Vector3 enemy_position)...

    private float CalculateG(Vector3 position) ...
    private float CalculateVisionIndex() ...
    private float CalculateDisspersionIndex()...
```

RESULTADOS OBTENIDOS

Con la implementación de este algoritmo, el funcionamiento del proyecto ha mejorado considerablemente.
Se observa, por ejemplo, que la elección de la cobertura tiene más sentido que anteriormente.

Lo cierto es que al no evitar ciclos complejos -ya que no tiene sentido en este caso- en ocasiones los enemigos se quedan dando vueltas entre los mismos nodos. Esto se podría solucionar, como he dicho anteriormente, evitando ciclos complejos; quizá evitando ciclos complejos siempre que no se vea al jugador y cuando se vea al jugador no evitándolos, aunque resultaría más evidente este comportamiento erróneo en esos casos en los que el enemigo ve al jugador pero se queda dando vueltas.

En cualquier caso, los resultados son bastante positivos con respecto al resultado del método anterior.