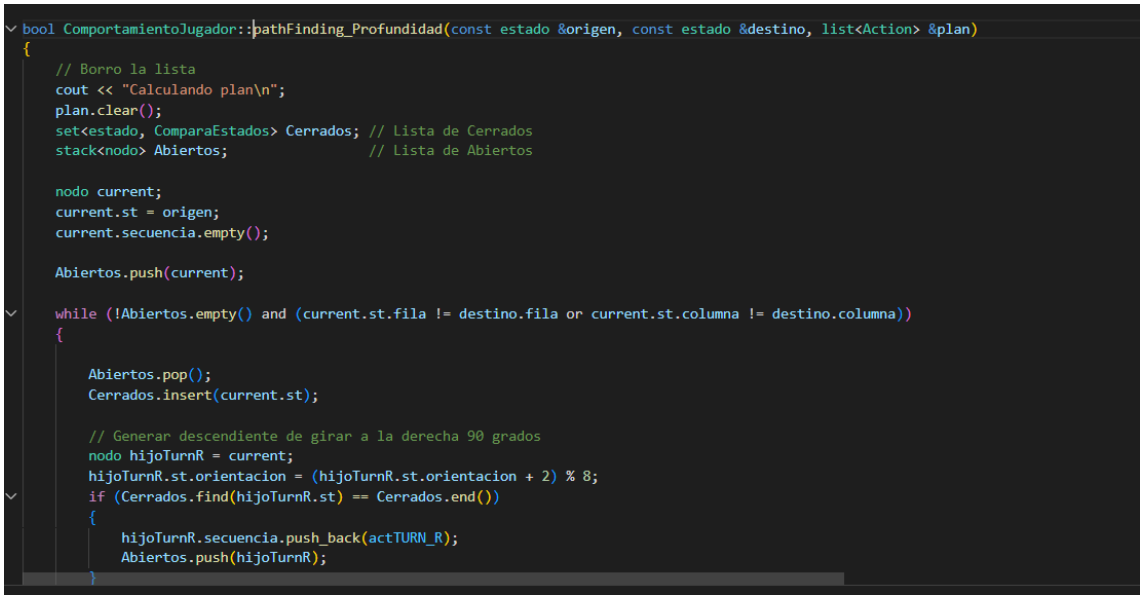
Memoria Practica 2 IA:

David Martínez Díaz GII-ADE

En primer lugar, realizamos en clase el tutorial básico, con esta demostración conseguí adquirir los conocimientos básicos del funcionamiento del path-finding, a través del método de profundidad.

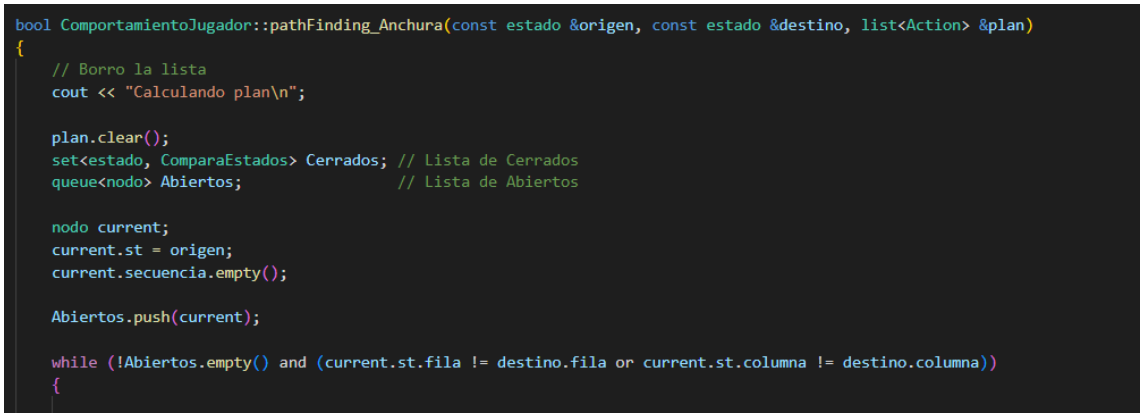
No había que implementar nada en ese nivel:



```
bool ComportamientoJugador::pathFinding_Profundidad(const estado &origen, const estado &destino, list<Action> &plan)
{
    // Borro la lista
    cout << "Calculando plan\n";
    plan.clear();
    set<estado, ComparaEstados> Cerrados; // Lista de Cerrados
    stack<nodo> Abiertos;                 // Lista de Abiertos

    nodo current;
    current.st = origen;
    current.secuencia.empty();

    Abiertos.push(current);

    while (!Abiertos.empty() and (current.st.fila != destino.fila or current.st.columna != destino.columna))
    {

        Abiertos.pop();
        Cerrados.insert(current.st);

        // Generar descendiente de girar a la derecha 90 grados
        nodo hijoTurnR = current;
        hijoTurnR.st.orientacion = (hijoTurnR.st.orientacion + 2) % 8;
        if (Cerrados.find(hijoTurnR.st) == Cerrados.end())
        {
            hijoTurnR.secuencia.push_back(actTURN_R);
            Abiertos.push(hijoTurnR);
        }
```

Lo único importante de este método es la utilización de una pila para los nodos abiertos.

En el nivel 1, había que implementar el path-finding de anchura, donde lo único que había que cambiar era la pila de abiertos por una cola:



```
bool ComportamientoJugador::pathFinding_Anchura(const estado &origen, const estado &destino, list<Action> &plan)
{
    // Borro la lista
    cout << "Calculando plan\n";

    plan.clear();
    set<estado, ComparaEstados> Cerrados; // Lista de Cerrados
    queue<nodo> Abiertos;                 // Lista de Abiertos

    nodo current;
    current.st = origen;
    current.secuencia.empty();

    Abiertos.push(current);

    while (!Abiertos.empty() and (current.st.fila != destino.fila or current.st.columna != destino.columna))
    {
```

En cuanto al nivel 2, la cosa se empieza a complicar ya que en un primer lugar decidir hacer el Coste Uniforme:

```
bool ComportamientoJugador::pathFinding_CosteUniforme(const estado &origen, const estado &destino, list<Action> &plan)
{
        // Borro la lista
        cout << "Calculando plan\n";

        plan.clear();
-       set<estado, ComparaEstados> Cerrados; // Lista de Cerrados
+       set<estado, ComparaEstadosNivel2> Cerrados; // Lista de Cerrados
        //priority_queue<nodo, vector<nodo>, compararCosto> Abiertos;   // Lista de Abiertos
        priority_queue<nodo> Abiertos;  // Lista de Abiertos

        nodo current;
        current.st = origen;
        current.secuencia.empty();
        current.costeUniforme = 0;
        current.st.bikini = false;
        current.st.deportivas = false;

        comprobarVestimenta(current.st);
+       costeCasilla(current, actIDLE);

        Abiertos.push(current);

        while (!Abiertos.empty() and (current.st.fila != destino.fila or current.st.columna != destino.columna))
        {

                Abiertos.pop();
+
+               if(Cerrados.find(current.st) == Cerrados.end()){
+
+
                Cerrados.insert(current.st);
```

Donde vi que era necesario una cola de prioridad para ordenar los nodos abiertos y así conseguir el coste mas eficiente.

Sin embargo, este algoritmo me daba una serie de resultados poco eficientes y lentos por los que decidí pasarme al algoritmo A estrella:

```
struct ComparaEstadosNivel2{

    bool operator()(const estado &a, const estado &n) const{

        if ((a.fila > n.fila) || (a.fila == n.fila and a.columna > n.columna) || (a.fila == n.fila and a.columna == n.columna and a.orientacion > n.orientacion) ||
        (a.fila == n.fila and a.columna == n.columna and a.orientacion == n.orientacion and a.bikini > n.bikini) ||
        (a.fila == n.fila and a.columna == n.columna and a.orientacion == n.orientacion and a.bikini == n.bikini and a.deportivas > n.deportivas ))
            return true;
        else
            return false;
    }

};

void ComportamientoJugador::comprobarVestimenta(estado & estado){

    if(mapaResultado[estado.fila][estado.columna] == 'K' and estado.bikini == false){

        // Como solo se puede coger uno, tendriamos que dejar las zapatillas
        estado.bikini = true;
        estado.deportivas = false;
    }
    else if(mapaResultado[estado.fila][estado.columna] == 'D' and estado.deportivas == false){

        // Como solo se puede coger uno, tendriamos que dejar las zapatillas
        estado.bikini = false;
        estado.deportivas = true;
    }
}
```

Me hice un par de métodos de comprobación para poder realizar la búsqueda de nodos de la manera correcta y obtener un coste coherente:

```
int ComportamientoJugador::costeCasilla(nodo & nodo1, const Action & accion){

    int coste = 0;

    if(accion == actIDLE){

        return 0;
    }
    else{

        switch (mapaResultado[nodo1.st.fila][nodo1.st.columna])
        {

        case 'A':

            if(nodo1.st.bikini == true){

                if(accion == actTURN_L || accion == actSEMITURN_L || accion == actSEMITURN_R || accion == actTURN_R){
                    coste = 5;
                }
                else{
                    coste = 10;
                }

            }
            else {

                if(accion == actTURN_L || accion == actSEMITURN_L || accion == actSEMITURN_R || accion == actTURN_R){

                    coste = 500;
                }
                else{
```

Utilice la heurística de chebychev para ordenar los nodos, ya que la distancia de Manhattan no me daría el camino mas eficiente y por tanto no sería el correcto:

```
int ComportamientoJugador::Chebychev(nodo & nodo1, const estado & objetivo){

    int diferencia_filas = abs(objetivo.fila - nodo1.st.fila);
    int diferencia_columnas = abs(objetivo.columna - nodo1.st.columna);

    int distancia_chebychev = max(diferencia_columnas, diferencia_filas);

    return distancia_chebychev;
}

struct compararValorF{

    bool operator()(const nodo & nodo1, const nodo & nodo2) const {
        return nodo1.st.valor_F > nodo2.st.valor_F;
    }
};

bool ComportamientoJugador::pathFinding_A_Estrella(const estado &origen, const estado &destino, list<Action> &plan)
{
    // Borro la lista
    cout << "Calculando plan\n";
    cout << "Destino f: " << destino.fila << " c: " << destino.columna;
    cout << "Origen f: " << origen.fila << " c: " << origen.columna;
    plan.clear();
    set<estado, ComparaEstadosNivel2> Cerrados; // Lista de Cerrados
    priority_queue<nodo, vector<nodo>, compararValorF> Abiertos;    // Lista de Abiertos

    nodo current;
    current.st = origen;
```

Para el nivel 3, donde se trata otra vez el tema de exploración, simplemente diseñe un código que buscase objetivos que no hayan sido descubiertos todavía, y que cuando este terminase realizase una búsqueda uniforme recorriendo el mapa de arriba hacia abajo.

También tuve que tener en cuenta que si cogía los elementos del borde provocaría fallos por lo que limite el rango del mapa para la búsqueda:

```
while(hay_plan == false){

    bool encontrado = false;
    objetivos.clear();
    plan.clear();

    estado aux;

    for(; fila_busqueda<mapaResultado.size()-3 and !encontrado; fila_busqueda++){

        if(mapaResultado.size() - 3 <= columna_busqueda){
            columna_busqueda = 3;
        }

        for(; columna_busqueda<mapaResultado.size()-3 and !encontrado; columna_busqueda++){

            if(mapaResultado[fila_busqueda][columna_busqueda] == '?'){

                aux.fila = fila_busqueda;
                aux.columna = columna_busqueda;
                encontrado = true;
            }
        }

    }

    if(mapaResultado.size() - 3 <= fila_busqueda){
            fila_busqueda = 3;
    }

    if(!encontrado){

        fila_aux += 2;
        if(mapaResultado.size() - 3 <= fila_aux or fila_aux < 3){
            fila_aux = 3;
            columna_aux += 5;
        }
        if(mapaResultado.size() - 3 <= columna_aux or columna_aux < 3){
            columna_aux = 3;
        }

        aux.fila = fila_aux;
        aux.columna = columna_aux;
    }
```

Por otro lado, añadí un algoritmo de búsqueda para la casilla de recarga por lo que cuando tuviese poca batería iría a recargar directamente si conoce su ubicación exacta:

```
    if(casillaEncontrada == false){

        casillaEncontrada = descubiertoCasillaEspecial();

        if(casillaEncontrada == true){

            casillaEspecial.fila = fila_especial;
            casillaEspecial.columna = columna_especial;

            objetivo_casilla_especial.clear();
            objetivo_casilla_especial.push_back(casillaEspecial);

            cout << "Casilla especial en la posicion [" << casillaEspecial.fila << "][" << casillaEspecial.columna << "]." << endl;

        }
    }

    if(casillaEncontrada == true and fin_recarga == false ){

        cout << "Buscando casilla recarga" << endl;

        if(sensores.bateria >= 3000 or sensores.vida < 200){
            fin_recarga = true;
            hay_plan = false;
            plan.clear();
        }

        if(sensores.terreno[0] == 'X' and fin_recarga == false ){
            accion = actIDLE;
        }
        else{

            if(hay_plan == false){

                plan.clear();
                hay_plan = pathFinding_A_Estrella(actual, casillaEspecial, plan);
            }

            if (hay_plan == true and plan.size() > 0){
                accion = plan.front();
                plan.pop_front();
            }
        }


        return accion;
    }
```

Para el nivel 4, me he creado una serie de métodos que tuvieran en cuenta múltiples objetivos y la existencia tanto de lobos como de aldeanos:

```
void ComportamientoJugador::esObjetivo(estado & estado_nodo){

    bool esObjetivo = false;
    int i = 0;

    estado objetivo;

    for(auto it=objetivos_vector.begin(); it != objetivos_vector.end() && esObjetivo == false; ++it){

        objetivo = *it;
        if(estado_nodo.fila == objetivo.fila and estado_nodo.columna == objetivo.columna){
            esObjetivo = true;
            estado_nodo.objetivos[i] = true;
        }
        ++i;
    }
}

int ComportamientoJugador::ChebychevMultipleObjetivos(nodo & nodo1, const vector<estado> &objetivos, const vector<bool> & objetivos_visitados){

    int aux = 1000000;

    auto it = objetivos.begin();

    for(int i=0; i<3; i++){

        if(objetivos_visitados.at(i) && aux > Chebychev(nodo1, *it)){
            aux = Chebychev(nodo1, *it);
        }
        it++;
    }

    return aux;
}

bool EsObstaculo4(unsigned char casilla, Sensores sensores)
{
    if (casilla == 'P' or casilla == 'M' or sensores.superficie[2] != '_')
        return true;
    else
        return false;
}
```

También me he vuelto a crear un A estrella, pero esta vez orientado a múltiples objetivos, creándome un vector de objetivos dentro del nodo para poder ir sabiendo cuando he pasado por uno o no.

En el método think, voy añadiendo los objetivos de 3 en 3, y los añado a un vector para luego pasárselo al path-finding.

```
if(sensores.nivel == 4){

    if(nuevos_objetivos == true){

        objetivos_vector.clear();

        for(int i=0; i<sensores.num_destinos; i++){

            estado aux;

            aux.fila = sensores.destino[2 * i];
            aux.columna = sensores.destino[2 * i + 1];

            objetivos_vector.push_back(aux);
            objetivos.push_back(aux);

            cout << "Tamaño del vector: " << objetivos_vector.size() << endl;
            cout << "Añadimos el destino " << i << " f: " << objetivos_vector.at(i).fila << " c: " << objetivos_vector.at(i).columna << endl;
        }

        nuevos_objetivos = false;
    }
}
```

Y poco a poco, cuando termino un recorrido compruebo que los objetivos han sido alcanzados y si ese es el caso añado unos nuevos automáticamente:

```
if(plan.size() == 0){
    nuevos_objetivos = true;
    cout << "Creando nuevos objetivos" << endl;
}

if(plan.front() == actFORWARD and (sensores.superficie[2] != '_')){
    hay_plan = false;
    accion = actIDLE;
    ultimaAccion = accion;
    return accion;
}

if(ultimaAccion == actWHEREIS){

    actual.fila = sensores.posF;
    actual.columna = sensores.posC;
    actual.orientacion = sensores.sentido;

    nivel_4_columna = sensores.posC;
    nivel_4_fila = sensores.posF;
    nivel_4_orientacion = sensores.sentido;

    while(hay_plan == false){

        plan.clear();
        tres_objetivos.clear();

        cout << endl << "----------------------------" << endl;

        for(int i=0; i<3; i++){

            cout << "Añadimos el destino " << i << " f: " << objetivos_vector.at(i).fila << " c: " << objetivos_vector.at(i).columna << endl;
            tres_objetivos.push_front(objetivos_vector.at(i));
            i++;
        }

        cout << endl << "----------------------------" << endl;

        if(casillaEncontrada == true and fin_recarga == false){
            plan.clear();
            hay_plan = pathFinding_A_Estrella(actual, casillaEspecial, plan);
        }
        else {
            hay_plan = pathFinding_A_Estrella_4(actual, objetivos_vector, plan, sensores);
        }

    }
```

Por último, comentar que también he tenido en cuenta la casilla de recarga, por lo que cuando tenga poca batería ira a por ella, aunque no he implementado la logística cuando te empuja un lobo.