Práctica 4. Exploración de grafos

Alejandro Segovia Gallardo
alejandro.segoviagallardo@alum.uca.es
Teléfono: 608842858
NIF: 32083695Y

29 de enero de 2018

1. Comente el funcionamiento del algoritmo y describa las estructuras necesarias para llevar a cabo su implementación.

   Mi algoritmo está basado en el algoritmo visto en clase de A*, para ello, me he basado en las estructuras definidas en Asedio.h, creando asi las siguientes estructuras para la implementación del algoritmo:

   AStarNode* abiertos: Un vector del tipo AStarNode para llevar el control de los nodos abriertos, el cual ordenaremos en cada iteración del algoritmo para mejora el camino.

   AStarNode* cerrados: Otro vector del mismo tipo que el anterior, el cual usaremos para insertar los nodos ya visitados, evitando asi la reevaluación de los nodos.

   AStarNode* current: Estructura del tipo AStarNode donde almacenaremos el nodo que estamos evaluando en todo momento.

   El funcionamiento del algoritmo de A* es el siguiente: Partiendo del nodo de origen, iremos expandiendo todos sus nodos hijos (se encontraran en el vector de abiertos), y siempre eligiendo la mejor opcion tras un análisis, y cambiando siempre el nodo current por el que estamos evaluando actualmente tras el análisis, los nodos ya visitados se irán guardando en el vector de nodos cerrados y cuando finaliza el analisis de todos los nodos hasta llegar al nodo destino, se produce un bucle para la recuperación del camino.

2. Incluya a continuación el código fuente relevante del algoritmo.

```
void DEF_LIB_EXPORTED calculateAdditionalCost(float** additionalCost,
                                              int cellsWidth, int cellsHeight,
                                              float mapWidth, float mapHeight,
                                              List<Object*> obstacles,
                                              List<Defense*> defenses)
{
    Vector3 dst;
    for(std::list<Defense*>::const_iterator def = defenses.begin();
      def != defenses.end();def++)
        for(int i = 0 ; i < cellsHeight ; ++i) {
            for(int j = 0 ; j < cellsWidth ; ++j) {
                dst.x = (*def)->position.x - (i*cellsWidth + cellsWidth);
                dst.y = (*def)->position.y - (j*cellsHeight + cellsHeight);

                additionalCost[i][j]=dst.length();
            }
        }
}


void DEF_LIB_EXPORTED calculatePath(AStarNode* originNode, AStarNode* targetNode
                    , int cellsWidth, int cellsHeight, float mapWidth, float mapHeight
                    , float** additionalCost, std::list<Vector3> &path) {

    bool nodoTerminal = false;
    float cellWidth = mapWidth/cellsWidth;
    float cellHeight = mapHeight/cellsHeight;

    AStarNode* current = originNode;
```

```
    std::vector<AStarNode*> abiertos;
    abiertos.push_back(current);
    std::vector<AStarNode*> cerrados;

    while(abiertos.size() != 0 and nodoTerminal == false){

        current = abiertos.front();
        abiertos.erase(abiertos.begin());
        cerrados.push_back(current);

        if(current == targetNode){
            nodoTerminal = true;
            path.push_front(current->position);
            current = targetNode;
        }

        else{
            int i,j;
            float distancia;
            for(List<AStarNode*>::iterator nodo = current->adjacents.begin();
                nodo != current->adjacents.end(); ++nodo)
                if(cerrados.end() == std::find(cerrados.begin(), cerrados.end(), (*nodo)))
                    if(abiertos.end() == std::find(abiertos.begin(), abiertos.end(), (*nodo))
                        ) {
                        i = (*nodo)->position.x / cellWidth;
                        j = (*nodo)->position.y / cellHeight;
                        (*nodo)->G = current->G +
                            _distance(current->position, (*nodo)->position)
                            + additionalCost[i][j];
                        (*nodo)->H =_sdistance((*nodo)->position, targetNode->position);
                        (*nodo)->F = (*nodo)->G + (*nodo)->H;
                        (*nodo)->parent = current;

                        abiertos.push_back(*nodo);
                    }

                    else {
                        distancia = _distance(current->position,(*nodo)->position);

                        if((*nodo)->G > current->G + distancia) {
                            (*nodo)->G = current->G + distancia;
                            (*nodo)->F = (*nodo)->G + (*nodo)->H;
                            (*nodo)->parent = current;
                        }
                    }
                    std::sort(abiertos.begin(), abiertos.end());
        }
    }

    while(current->parent != originNode) {
        current = current->parent;
        path.push_front(current->position);
    }
}
```

Todo el material incluido en esta memoria y en los ficheros asociados es de mi autoría o ha sido facilitado por los profesores de la asignatura. Haciendo entrega de esta práctica confirmo que he leído la normativa de la asignatura, incluido el punto que respecta al uso de material no original.