#pragma once

#include "../common.h"

#include <iostream>

#include <vector>

#include <list>

#include "LNode.h"

#include "LEdge.h"

using namespace std;

namespace DS

{

template<class N, class E>

class LGraph

{

public :

typedef LGraph<N,E> self;

typedef LNode<self> Node;

typedef LEdge<self> Edge;

typedef N NodeData;

typedef E EdgeData;

vector<Node\*> nodes;

Node\* insertNode( N node\_data, double x, double y, int id );

vector<Edge\*> insertEdge( Node\* from, Node\* to, E edge\_data );

void removeEdge( Edge\* edge );

void removeNode( Node\* node );

int nodes\_glIndx;

int edges\_glIndx;

~LGraph()

{

for ( int q = 0; q < nodes.size(); q++ )

{

for ( int e = 0; e < nodes[q]->edges.size(); e++ )

{

if ( nodes[q]->edges[e] != NULL )

{

delete nodes[q]->edges[e];

nodes[q]->edges[e] = NULL;

}

}

nodes[q]->edges.clear();

}

for ( int q = 0; q < nodes.size(); q++ )

{

if ( nodes[q] != NULL )

{

delete nodes[q];

nodes[q] = NULL;

}

}

nodes.clear();

}

void print();

};

template<class N, class E>

LNode< LGraph<N,E> >\* LGraph<N,E>::insertNode( N node\_data, double x, double y, int id )

{

LNode< LGraph<N,E> >\* \_pNode = new LNode<LGraph<N,E> >( node\_data, x, y, id );

nodes.push\_back( \_pNode );

return \_pNode;

}

template<class N, class E>

vector<LEdge< LGraph<N,E> >\*> LGraph<N,E>::insertEdge( LNode<LGraph<N,E> >\* from,

LNode<LGraph<N,E> >\* to,

E edge\_data )

{

vector<LEdge< LGraph<N,E> >\*> \_res;

// Check if already has this edge

for ( int q = 0; q < from->edges.size(); q++ )

{

LEdge<LGraph<N,E> >\* \_edge = from->edges[q];

if ( \_edge->nodes[1]->id == to->id )

{

// Already there

return \_res;

}

}

for ( int q = 0; q < to->edges.size(); q++ )

{

LEdge<LGraph<N,E> >\* \_edge = to->edges[q];

if ( \_edge->nodes[1]->id == from->id )

{

// Already there

return \_res;

}

}

LEdge<LGraph<N,E> >\* \_edge\_from = new LEdge<LGraph<N,E> >( edge\_data );

\_edge\_from->nodes[0] = from;

\_edge\_from->nodes[1] = to;

from->edges.push\_back( \_edge\_from );

\_edge\_from->drawable = true;

LEdge<LGraph<N,E> >\* \_edge\_to = new LEdge<LGraph<N,E> >( edge\_data );

\_edge\_to->nodes[0] = to;

\_edge\_to->nodes[1] = from;

to->edges.push\_back( \_edge\_to );

\_edge\_to->drawable = false;

\_res.push\_back( \_edge\_from );

\_res.push\_back( \_edge\_to );

return \_res;

}

template<class N, class E>

void LGraph<N,E>::removeEdge( LEdge<LGraph<N,E> >\* edge )

{

// Remove from the "from" node's edges

LNode<LGraph<N,E> >\* \_node\_from = edge->nodes[0];

LNode<LGraph<N,E> >\* \_node\_to = edge->nodes[1];

int \_indx = -1;

for ( int q = 0; q < \_node\_from->edges.size(); q++ )

{

if ( \_node\_from->edges[q]->nodes[1] == \_node\_to )

{

\_indx = q;

break;

}

}

if ( \_indx != -1 )

{

\_node\_from->edges.erase( \_node\_from->edges.begin() + \_indx );

}

// Remove from the "to" node's edges

\_indx = -1;

for ( int q = 0; q < \_node\_to->edges.size(); q++ )

{

if ( \_node\_to->edges[q]->nodes[1] == \_node\_from )

{

\_indx = q;

break;

}

}

if ( \_indx != -1 )

{

\_node\_to->edges.erase( \_node\_to->edges.begin() + \_indx );

}

delete edge;

}

template<class N, class E>

void LGraph<N,E>::removeNode( LNode<LGraph<N,E> >\* node )

{

// Remove edges from the "from" node and, if bidir, from the "to" this node

vector<Edge\*> \_edges = node->edges;

// TODO: Check issue here

for ( int q = 0; q < \_edges.size(); q++ )

{

removeEdge( \_edges[q] );

}

// Remove the node

int \_indx = -1;

for ( int q = 0; q < nodes.size(); q++ )

{

if ( nodes[q] == node )

{

\_indx = q;

break;

}

}

if ( \_indx != -1 )

{

nodes.erase( nodes.begin() + \_indx );

}

}

template<class N, class E>

void LGraph<N,E>::print()

{

cout << "graph: " << endl;

for ( int q = 0; q < nodes.size(); q++ )

{

cout << " node ( " << nodes[q]->data << " ): ";

vector<LEdge<LGraph<N,E> >\* > \_node\_edges = nodes[q]->edges;

for ( int p = 0; p < \_node\_edges.size(); p++ )

{

cout << ( \_node\_edges[p]->nodes[1] )->data << "(" << \_node\_edges[p]->data << ") ";

}

cout << endl;

}

cout << "\*\*\*\*\*\*\*\*" << endl;

}

}

#pragma once

#include "LGraph.h"

#include <vector>

#include <utility>

using namespace std;

namespace DS

{

template<class G>

class LNode

{

public :

typedef typename G::NodeData N;

typedef typename G::Edge Edge;

N data;

vector<Edge\*> edges;

double x;// X coordinate in the scene

double y;// Y coordinate in the scene

int id;// Unique identifier, the same as index in graph's vector of nodes

int glIndx;

double f,g,h; // For A\* search

float ff[MAX\_PARALLEL\_REQUESTS];

float gg[MAX\_PARALLEL\_REQUESTS];

float hh[MAX\_PARALLEL\_REQUESTS];

LNode\* parent;// For A\* search path reconstruction

float d;// For Dijkstra precalculation

pair<LNode\*,Edge\*> parentInfo[MAX\_PARALLEL\_REQUESTS];

bool inOpen;

LNode( N node\_data, double x, double y, int id )

{

data = node\_data;

inOpen = false;

this->glIndx = -1;

this->x = x;

this->y = y;

this->id = id;

this->f = 0;

this->g = 0;

this->h = 0;

this->d = INF;

this->parent = NULL;

for ( int q = 0; q < MAX\_PARALLEL\_REQUESTS; q++ )

{

this->parentInfo[q].first = NULL;

this->parentInfo[q].second = NULL;

this->ff[q] = 0.0f;

this->gg[q] = 0.0f;

this->hh[q] = 0.0f;

}

}

};

}