Path Finding Documentation

Shane Coates *Advanced Diploma of Professional Games Development*
Assessment 3 (Game Artificial Intelligence)

Overview

For this assessment we are required to expand on our previous assignment, using Game Math and OpenGL to demonstrate the implementation of pathfinding and Artificial Intelligence. We were tasked with researching various pathfinding alrgorithms and utilising the most efficient algorithm for certain tasks.

Structure of Graph Data

There are three main methods of data storage that I looked into when preparing for this assessment.

- A 2D array of Nodes, denoting their position on screen

```
#define NODES_X 25
#define NODES_Y 25
#define NODES_W 25
#define NODES_H 25

Node* m_nodes[X_NODES][Y_NODES];

for(x = 0; x < NODES_X; x++)
{
    for(y = 0; y < NODES_Y; y++)
    {
        m_nodes[x][y] = new Node();
        m_nodes[x][y]->SetTranslation(Vec2(x * NODES_W, y * NODES_H));
    }
}
```

- A list of Nodes, which each had a list of it's neighbours:

```
class Node
{
public:
   //public variables and functions:

private:
   //private variables and functions:

   //List of Neighbouring nodes
   std::list<Node*> m_neighbours
};
std::list<Node*> m_nodes;
```

- A list of Nodes, and a list of Edges:

```
struct Node
{

};

struct Edge
{
   Node* node1;
   Node* node2;
};

std::list<Node*> nodes;
std::list<Edge*> edges;
```

Graph Data Algorithms

- **Finding a node in the graph based on a condition.**

Using the list of Nodes, if you want to search for a specific node based on a condition such as it's global transform, you could create a function that returns a Node pointer, and takes in a Vector2 position and a float range. This function would search an area defined in size by the 'range' variable, and at the position of the 'position' variable. This function would search for any node's within that area. This could be useful if you wanted to have an AI seek to the location of the player, or to the location of a mouse click

```
void FindNodeInRange(Vec2 _pos, float range, std::list<Node*> &outNodes)
{
   //pre-define some variables for readability
   float _minX = _pos.x - range;
   float _maxX = _pos.x + range;
   float _minY = _pos.Y - range;
   float _maxY = _pos.Y + range;

   //search through all nodes in the list
   for(auto itr = m_nodes.begin(); itr != m_nodes.end(); itr++)
   {
      //check to see if the node is in range
      if((*itr)->GetTranslation().x > _minX &&
         (*itr)->GetTranslation().x < _maxX &&
         (*itr)->GetTranslation().x > _minY &&
         (*itr)->GetTranslation().x < _maxY)
      outNodes.push_back(itr)
   }
}
```

- **Adding Nodes to the graph**.

When adding nodes to the graph, you need to be careful. You need to make sure that the new node won't interfere with your current graph. You then need to recalculate all of the edges, and your entire graph. To be safe, clearing your edges and recalculating all of them could remove a lot of the risk.

```
void AddNode(Vec2 _position)
{
   m_nodes.push_back(new Node(_position));
   m_edges.clear();
   AddEdges();
}
```

- **Connecting 2 nodes within the graph**

Depending on how your nodes are set up, there is definitely potential for errors to occur when connecting nodes. There is the possibility for errors to occur where:

- Connections are made through smaller object
- There are 2 edges between the same two nodes

In some scenarios, you might want an edge going each way between nodes, and in other scenarios you won't. For example, if you have path's that are only traversable in one direction, such as a cliff you can jump down but not up, you would want to have seperate edges calculated for going in each direction.

When connecting two nodes, or, creating an edge, the simplest way would be to loop through the list of nodes, and for each node, check if there are others in range.

```
void AddEdges()
{
  std::list<Node*> edgeNodes;
  for(auto node = m_nodes.begin(); node != m_nodes.end(); node++)
  {
     FindNodeInRange(node.GetTranslation(), range, edgeNodes);
     for(auto node = edgeNodes.begin(); node != edgeNodes.end(); node++)
       m_edges.push_back(new Edge(node, itr);
     edgeNodes.clear();
  }
}
```

- **Finding Neighbors of a given node**

In my graph, Neighbours are stored in a list of nodes within each individual Node. To calculate this, I iterated through my list of nodes twice, and calculated the distance between each pair. If the distance was under a certain amount, a neighbour was added. The distance variable I used for this calculation was the hypotenuse of the spacing between nodes. This was intentional, as all of my nodes were uniformly spaced. Because of this calculation, every node's neighbours were any other nodes that were right next to it vertically, horizontally or diagonally.

```
void TestState::CalculateNeighbours()
{
  for(auto node1 = m_nodes.begin(); node1 != m_nodes.end(); node1++)
  {
    (*node1)->ClearNeighbours();
    (*node1)->SetCost(1);
    for(auto node2 = m_nodes.begin(); node2 != m_nodes.end(); node2++)
      {
      float x = abs((*node1)->GetGlobalTransform().GetTranslation().x -
                    (*node2)->GetGlobalTransform().GetTranslation().x);
      float y = abs((*node1)->GetGlobalTransform().GetTranslation().y -
```

```
                    (*node2)->GetGlobalTransform().GetTranslation().y);

      Vec2 dist = Vec2(x, y);
      if(dist.Length() < sqrt((X_SPACING * X_SPACING) +
                              (Y_SPACING * Y_SPACING)))
      {
            (*node1)->AddNeighbours(*node2);
      }
    }
  }
}
```

- **How can this data be saved and loaded from file.**

If I were to save and load this data from file, I would use XML. I didn't in this assessment, because everything was randomly generated. But you could easily store the data of each node into an XML file.

xml:

```
<?xml version="1.0"?>
-<Node>
   <x=”100” y=”100”>
   <x=”200” y=”200”>
<Node/>
```

c++:

```
tinyxml2::XMLDocument doc;
if(doc.LoadFile("nodeData.xml") == tinyxml2::XML_NO_ERROR)
{
  auto element = doc.FirstChildElement("Node")->FirstChildElement();
  while(element != NULL)
    {
    m_nodes.push_back(new Node(Vec2(       atoi(element->Attribute("x"),

atoi(element->Attribute("y”)));
    }
    element = element->NextSiblingElement();
  }
}
```

PathFinding

- **How have you implemented the Dijkstra's Path finding algorithm, how is the the path calculated?**

psuedo-code:

```
Node - currentNode
list of Nodes - openList
list of Nodes - clostList
while(end node not found)
{
  if one of the current node's neighbours is an end node
  {
    make that node the current node
    break the loop
  }
  if not
  add all neighbours to a list
  for each neighbour:
    - set their parent
    - add to open list
    - calculate gScore

  remove the current Node from the open list
  remove any nodes form the open list if they are on the closed list

  sort the open list by gScore
  set the current node to the front of the open list
}

if(end node is found)
{
  push current node onto the path;
  while(current node's parent is NOT the start node)
  {
    push the current node's parent onto the path
    set current node to it's current parent
  }
}
```

c++:

```
//create a list of neighbours
std::list<Node*> _neighbours = _currentNode->GetNeighbours();
//iterate through the neighbours and add them to the open list
for(auto itr = _neighbours.begin(); itr != _neighbours.end(); itr++)
{
  //check for a target
  if((*itr)->GetTarget())
  {
  (*itr)->SetParent(_currentNode);
  _currentNode = *itr;
  break;
  }

  //make sure it hasn't already been searched
  if((*itr)->GetParent() == nullptr)
  {
  //set each neighbour's 'open' variable to true
  (*itr)->SetOpen(true);
  //set each neighbour's parent
  (*itr)->SetParent(_currentNode);
  //set each neighbour's GScore
  (*itr)->m_GScore = (DistanceBetween((*itr), (*itr)->GetParent))
                      + (*itr)->m_GScore);
  //push the neighbour onto the open list
  _openList.push_back(*itr);
  }
}
```

- **How is the path stored, what data structures are you using, and what functions are available for manipulating / getting information from your path?**

(*Provide pseudocode and supporting C++ code Snippets)*

I store my path in a list of Nodes. This list holds all nodes in the order they are meant to be traversed. An AI ship calculates this path, and follows it. To follow it, it moves to the first on the list. Once it reaches that Node, it is removed from the list.

psuedo-code:

```
calculate the path of nodes.
MoveTowards(path.front())
if(touching path.front())
  remove path.front()
```

c++:

```
//create a list of neighbours
std::list<Node*> _neighbours = _currentNode->GetNeighbours();
//iterate through the neighbours and add them to the open list
for(auto itr = _neighbours.begin(); itr != _neighbours.end(); itr++)
{
  //check for a target
  if((*itr)->GetTarget())
  {
  (*itr)->SetParent(_currentNode);
  _currentNode = *itr;
  break;
  }

  //make sure it hasn't already been searched
  if((*itr)->GetParent() == nullptr)
  {
  //set each neighbour's 'open' variable to true
  (*itr)->SetOpen(true);
  //set each neighbour's parent
  (*itr)->SetParent(_currentNode);
  //set each neighbour's GScore
  (*itr)->m_GScore = (DistanceBetween((*itr), (*itr)->GetParent))
                      + (*itr)->m_GScore);
  //push the neighbour onto the open list
  _openList.push_back(*itr);
  }
}
```

- **Path Smoothing, what are some methods you could use?**

My path's are smoothed just in the way that they are used. My AI ships head to the next node on their path. They rotate so that they are facing the node, at a speed relative to the angle between the way it is currently facing and the way it wants to be facing.
If drawing a line along the path, it would not be all too complicated to calculate a bezier curve between each Edge or set of neighbours.

- **How have you modified the Dijkstra's pathfinding Algorithm to perform the AStar pathfinding algorithm?** (*show C++ code Snippets)*

The only differences between my two algorithms is that they both have different conditions they are looking for, and AStar calculates a heuristic value based on the distance from a given node to the end node. This heuristic value is added to the gScore of the node and used to sort the open list, rather than just the gScore. By doing this, AStart prioritises nodes that are physically closer to the final end node when searching.

```
//calculating the hScore of a given node
float _dx1;
float _dy1;
float _dx2 = _currentNode->m_xPos - _endNode->m_xPos;
float _dy2 = _currentNode->m_yPos - _endNode->m_yPos;
float _cross;
for(auto itr = _nodes.begin(); itr != _nodes.end(); itr++)
{
      _dx1 = (*itr)->m_xPos - _endNode->m_xPos;
      _dy1 = (*itr)->m_yPos - _endNode->m_yPos;
      _cross = abs(_dx1*_dy2 - _dx2*_dy1);
      (*itr)->m_HScore = _cross * 0.001;
}
```

- **What are some scenarios you would use the AStar Pathfinding algorithm?**
    - To return “home”
    - To chase a player, or object
    - To move towards a specific object on the screen

- **What are some scenarios you would use the Dijkstra's Pathfinding Algorithm.**
    - Finding the nearest health pack when low on health
    - Finding the nearest enemy
    - Finding the nearest resource or object

- **What are the Pro’s and Cons for both Dijkstra's and AStar pathfinding?**

	Dijkstra	**AStar**
Pro	• Can find something without already knowing where it is • If there are multiple targets, finds the closest one	• Usually fast • Prioritises the end node
Con	• Can be slow • Doesn’t prioritise the search towards the end node	• Can take unnecessarily long if the correct path involves heading away from the end node at any point

Implementation

Show a small code snippet for how you will find a path using your Pathfinding Algorithm. Here is an example implementation: This implementation will allow for multiple paths to be calculated simultaneously on the same graph.

```
// Member variables
std::list<Node*> m_nodes;
std::list<Node*> m_path;

//called when you want to start finding a path
//-----------------------------------------------------------
m_path.clear();
GetDijkstrasPath(m_nodes, m_path);
//-----------------------------------------------------------

//during your update loop
if(m_path.size() > 1)
{
  m_AI->TurnTowards(m_path.front()->GetGlobalTransform().GetTranslation(),
10.0f);

  Vec2 shipTrans = m_AI->GetGlobalTransform().GetTranslation();
  Vec2 nodeTrans = m_path.front()->GetGlobalTransform().GetTranslation();

  if( shipTrans.x - x < nodeTrans.x &&
      shipTrans.x + x > nodeTrans.x &&
      shipTrans.y - y < nodeTrans.y &&
      shipTrans.y + y > nodeTrans.y)
  m_path.pop_front();
}
```