Asignatura: INTELIGENCIA ARTIFICIAL
Profesor: D. Sc. Gerardo García Gil
2022-B
López Arellano Ricardo David
Universidad de Guadalajara (CUCEI)

Presentación

La Inteligencia Artificial (IA) es una rama de la informática que estudia procedimientos automatizados para lograr que un autómata logre ser o parecer inteligente. Éstos, tratan de imitar diversas áreas del comportamiento humano, con el fin de acercarse o llegar a superar a una persona común y corriente, por ejemplo, un experto en cierta materia, en la toma de decisiones o diversas tareas que un sistema computacional puede realizar mejor o más rápido. Es en este contexto que la inteligencia artificial se hace presente en áreas como la robótica y videojuegos, en donde se busca lograr un comportamiento inteligente en un robot o un personaje virtual. Un aspecto básico que se tiene que cumplir, en ambas áreas, es ser capaz de moverse de un punto a otro, esquivando, de forma inteligente y natural, obstáculos que se pueden presentar. Para resolver esta problemática existe un área de la inteligencia artificial llamada Pathfinding.

Introducción

Se le llama Pathfinding al área de la inteligencia artificial que se encarga de solucionar este problema. Pathfinding principalmente busca encontrar un camino de un punto a otro lo más rápida y eficientemente posible. Se pueden aplicar distintos enfoques y algoritmos, donde se logran resultados distintos, generalmente para resolver objetivos distintos, dependiendo si se busca un camino óptimo o cercano al óptimo.

En este documento se describirá Pathfinding aplicado a los videojuegos. Área en donde cada vez es más requerido encontrar mejores algoritmos que ayuden a ahorrar recursos y mejorar la experiencia del jugador, con mayor rapidez y naturalidad en el movimiento.

Antecedentes

Para poder utilizar los distintos algoritmos de Pathfinding, primero, es necesario transformar o resumir el problema o el mapa con él que se trabajará a una estructura que sea fácil de procesar en un lenguaje de programación. Para lograr esto, existen los llamados mapas malla, o mallas mapa, que representan generalmente un grafo donde se definen los lugares por donde se puede o no cruzar, y cuan costoso es hacerlo. Para este trabajo se utilizará el mapa cuadriculado de costo uniforme, donde se dividirá el mapa en una cuadrícula, en la que los sub cuadrados, o nodos, tendrán todo un mismo costo para moverse en forma recta de uno a otro y un mismo costo para hacerlo en forma diagonal.

Desarrollo

1) Tener dos listas de celdas; una abierta y una cerrada. En la lista abierta iremos introduciendo celdas que tenemos que evaluar, para ver si son buenas candidatas a formar parte del camino final. En la lista cerrada introduciremos las celdas ya evaluadas.

2) La evaluación de las celdas se hace en base a dos factores: la longitud del camino más corto para llegar desde el punto de origen a la celda actual, y la estimación del camino más corto que podría quedar para llegar al destino. La estimación para el destino se hace usando una función heurística, en concreto se suele usar la Distancia Manhattan, que esencialmente consiste en contar cuantas celdas nos separan del destino vertical y horizontalmente, sin movimientos diagonales; como si estuviéramos recorriendo manzanas en una gran ciudad y no pudiéramos atravesarlas, sólo bordearlas; de ahí viene el nombre.

Implementación

```
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

#include "olcConsoleGameEngine.h"

class OneLoneCoder_PathFinding : public olcConsoleGameEngine
{
public:
	OneLoneCoder_PathFinding()
	{
	m_sAppName = L"Path Finding";
	}

private:

	struct sNode
	{
	bool bObstacle = false;
	bool bVisited = false;
	float fGlobalGoal;
	float fLocalGoal;
	int x;                                          // Nodes position in 2D space
	int y;
	vector<sNode*> vecNeighbours;          // Connections to neighbours
	sNode* parent;                                  // Node connecting to this node that
	offers shortest parent
	};

	sNode *nodes = nullptr;
	int nMapWidth = 16;
```

```
	int nMapHeight = 16;

	sNode *nodeStart = nullptr;
	sNode *nodeEnd = nullptr;


protected:
	virtual bool OnUserCreate()
	{
	// Create a 2D array of nodes - this is for convenience of rendering and construction
	// and is not required for the algorithm to work - the nodes could be placed anywhere
	// in any space, in multiple dimensions...
	nodes = new sNode[nMapWidth * nMapHeight];
	for (int x = 0; x < nMapWidth; x++)
		for (int y = 0; y < nMapHeight; y++)
		{
			nodes[y * nMapWidth + x].x = x; // ...because we give each node its own
	coordinates
			nodes[y * nMapWidth + x].y = y;
			nodes[y * nMapWidth + x].bObstacle = false;
			nodes[y * nMapWidth + x].parent = nullptr;
			nodes[y * nMapWidth + x].bVisited = false;
		}

	// Create connections - in this case nodes are on a regular grid
	for (int x = 0; x < nMapWidth; x++)
		for (int y = 0; y < nMapHeight; y++)
		{
			if(y>0)
				nodes[y*nMapWidth + x].vecNeighbours.push_back(&nodes[(y -
	1) * nMapWidth + (x + 0)]);
			if(y<nMapHeight-1)
				nodes[y*nMapWidth + x].vecNeighbours.push_back(&nodes[(y +
	1) * nMapWidth + (x + 0)]);
			if (x>0)
				nodes[y*nMapWidth + x].vecNeighbours.push_back(&nodes[(y +
	0) * nMapWidth + (x - 1)]);
			if(x<nMapWidth-1)
				nodes[y*nMapWidth + x].vecNeighbours.push_back(&nodes[(y +
	0) * nMapWidth + (x + 1)]);

			// We can also connect diagonally
			/*if (y>0 && x>0)
				nodes[y*nMapWidth + x].vecNeighbours.push_back(&nodes[(y -
	1) * nMapWidth + (x - 1)]);
			if (y<nMapHeight-1 && x>0)
				nodes[y*nMapWidth + x].vecNeighbours.push_back(&nodes[(y +
	1) * nMapWidth + (x - 1)]);
```

```
                if (y>0 && x<nMapWidth-1)
                        nodes[y*nMapWidth + x].vecNeighbours.push_back(&nodes[(y -
1) * nMapWidth + (x + 1)]);
                if (y<nMapHeight - 1 && x<nMapWidth-1)
                        nodes[y*nMapWidth + x].vecNeighbours.push_back(&nodes[(y +
1) * nMapWidth + (x + 1)]);
                */
        }

// Manually positio the start and end markers so they are not nullptr
nodeStart = &nodes[(nMapHeight / 2) * nMapWidth + 1];
nodeEnd = &nodes[(nMapHeight / 2) * nMapWidth + nMapWidth-2];
return true;
}

bool Solve_AStar()
{
// Reset Navigation Graph - default all node states
for (int x = 0; x < nMapWidth; x++)
        for (int y = 0; y < nMapHeight; y++)
        {
                nodes[y*nMapWidth + x].bVisited = false;
                nodes[y*nMapWidth + x].fGlobalGoal = INFINITY;
                nodes[y*nMapWidth + x].fLocalGoal = INFINITY;
                nodes[y*nMapWidth + x].parent = nullptr;        // No parents
        }

auto distance = [](sNode* a, sNode* b) // For convenience
{
        return sqrtf((a->x - b->x)*(a->x - b->x) + (a->y - b->y)*(a->y - b->y));
};

auto heuristic = [distance](sNode* a, sNode* b) // So we can experiment with heuristic
{
        return distance(a, b);
};

// Setup starting conditions
sNode *nodeCurrent = nodeStart;
nodeStart->fLocalGoal = 0.0f;
nodeStart->fGlobalGoal = heuristic(nodeStart, nodeEnd);

// Add start node to not tested list - this will ensure it gets tested.
// As the algorithm progresses, newly discovered nodes get added to this
// list, and will themselves be tested later
list<sNode*> listNotTestedNodes;
listNotTestedNodes.push_back(nodeStart);
```

```
// if the not tested list contains nodes, there may be better paths
// which have not yet been explored. However, we will also stop
// searching when we reach the target - there may well be better
// paths but this one will do - it wont be the longest.
while (!listNotTestedNodes.empty() && nodeCurrent != nodeEnd)// Find absolutely
shortest path // && nodeCurrent != nodeEnd)
{
	// Sort Untested nodes by global goal, so lowest is first
	listNotTestedNodes.sort([](const sNode* lhs, const sNode* rhs){ return lhs-
>fGlobalGoal < rhs->fGlobalGoal; } );

	// Front of listNotTestedNodes is potentially the lowest distance node. Our
	// list may also contain nodes that have been visited, so ditch these...
	while(!listNotTestedNodes.empty() && listNotTestedNodes.front()->bVisited)
		listNotTestedNodes.pop_front();

	// ...or abort because there are no valid nodes left to test
	if (listNotTestedNodes.empty())
		break;

	nodeCurrent = listNotTestedNodes.front();
	nodeCurrent->bVisited = true; // We only explore a node once


	// Check each of this node's neighbours...
	for (auto nodeNeighbour : nodeCurrent->vecNeighbours)
	{
		// ... and only if the neighbour is not visited and is
		// not an obstacle, add it to NotTested List
		if (!nodeNeighbour->bVisited && nodeNeighbour->bObstacle == 0)
			listNotTestedNodes.push_back(nodeNeighbour);

		// Calculate the neighbours potential lowest parent distance
		float fPossiblyLowerGoal = nodeCurrent->fLocalGoal +
distance(nodeCurrent, nodeNeighbour);

		// If choosing to path through this node is a lower distance than what
		// the neighbour currently has set, update the neighbour to use this node
		// as the path source, and set its distance scores as necessary
		if (fPossiblyLowerGoal < nodeNeighbour->fLocalGoal)
		{
			nodeNeighbour->parent = nodeCurrent;
			nodeNeighbour->fLocalGoal = fPossiblyLowerGoal;

			// The best path length to the neighbour being tested has
changed, so
			// update the neighbour's score. The heuristic is used to globally
bias
```

```
                        // the path algorithm, so it knows if its getting better or worse. At
some
                        // point the algo will realise this path is worse and abandon it, and
then go
                        // and search along the next best path.
                        nodeNeighbour->fGlobalGoal = nodeNeighbour->fLocalGoal +
heuristic(nodeNeighbour, nodeEnd);
                }
        }
}

return true;
}

virtual bool OnUserUpdate(float fElapsedTime)
{
int nNodeSize = 9;
int nNodeBorder = 2;

// Use integer division to nicely get cursor position in node space
int nSelectedNodeX = m_mousePosX / nNodeSize;
int nSelectedNodeY = m_mousePosY / nNodeSize;

if (m_mouse[0].bReleased) // Use mouse to draw maze, shift and ctrl to place start and
end
{
        if(nSelectedNodeX >=0 && nSelectedNodeX < nMapWidth)
                if (nSelectedNodeY >= 0 && nSelectedNodeY < nMapHeight)
                {
                        if (m_keys[VK_SHIFT].bHeld)
                                nodeStart = &nodes[nSelectedNodeY * nMapWidth +
nSelectedNodeX];
                        else if (m_keys[VK_CONTROL].bHeld)
                                nodeEnd = &nodes[nSelectedNodeY * nMapWidth +
nSelectedNodeX];
                        else
                                nodes[nSelectedNodeY * nMapWidth +
nSelectedNodeX].bObstacle = !nodes[nSelectedNodeY * nMapWidth +
nSelectedNodeX].bObstacle;

                        Solve_AStar(); // Solve in "real-time" gives a nice effect
                }
}

// Draw Connections First - lines from this nodes position to its
// connected neighbour node positions
Fill(0, 0, ScreenWidth(), ScreenHeight(), L' ');
for (int x = 0; x < nMapWidth; x++)
```

```
	for (int y = 0; y < nMapHeight; y++)
	{
		for (auto n : nodes[y*nMapWidth + x].vecNeighbours)
		{
			DrawLine(x*nNodeSize + nNodeSize / 2, y*nNodeSize + nNodeSize
/ 2,
					n->x*nNodeSize + nNodeSize / 2, n->y*nNodeSize +
nNodeSize / 2, PIXEL_SOLID, FG_DARK_BLUE);
		}
	}

// Draw Nodes on top
for (int x = 0; x < nMapWidth; x++)
	for (int y = 0; y < nMapHeight; y++)
	{

		Fill(x*nNodeSize + nNodeBorder, y*nNodeSize + nNodeBorder,
			(x + 1)*nNodeSize - nNodeBorder, (y + 1)*nNodeSize -
nNodeBorder,
			PIXEL_HALF, nodes[y * nMapWidth + x].bObstacle ? FG_WHITE :
FG_BLUE);

		if (nodes[y * nMapWidth + x].bVisited)
			Fill(x*nNodeSize + nNodeBorder, y*nNodeSize + nNodeBorder, (x
+ 1)*nNodeSize - nNodeBorder, (y + 1)*nNodeSize - nNodeBorder, PIXEL_SOLID,
FG_BLUE);

		if(&nodes[y * nMapWidth + x] == nodeStart)
			Fill(x*nNodeSize + nNodeBorder, y*nNodeSize + nNodeBorder, (x
+ 1)*nNodeSize - nNodeBorder, (y + 1)*nNodeSize - nNodeBorder, PIXEL_SOLID,
FG_GREEN);

		if(&nodes[y * nMapWidth + x] == nodeEnd)
			Fill(x*nNodeSize + nNodeBorder, y*nNodeSize + nNodeBorder, (x
+ 1)*nNodeSize - nNodeBorder, (y + 1)*nNodeSize - nNodeBorder, PIXEL_SOLID, FG_RED);


	}


// Draw Path by starting ath the end, and following the parent node trail
// back to the start - the start node will not have a parent path to follow
if (nodeEnd != nullptr)
{
	sNode *p = nodeEnd;
	while (p->parent != nullptr)
	{
		DrawLine(p->x*nNodeSize + nNodeSize / 2, p->y*nNodeSize + nNodeSize /
```

```
		2,
					p->parent->x*nNodeSize + nNodeSize / 2, p->parent-
		>y*nNodeSize + nNodeSize / 2, PIXEL_SOLID, FG_YELLOW);

				// Set next node to this node's parent
				p = p->parent;
			}
		}

		return true;
		}

};

int main()
{
		OneLoneCoder_PathFinding game;
		game.ConstructConsole(160, 160, 6, 6);
		game.Start();
		return 0;
}
```

Experimentos

Hice varias pruebas hasta que por fin me dio el resultado correcto para el algoritmo A*.

Resultados

JPS presenta una mejora bastante significativa en el rendimiento. JPS, en todos los casos probados raramente supera el segundo de ejecución, y la cantidad de nodos expandidos es considerablemente menor a la de A*, con todas las combinaciones de las heurísticas. En los casos más extremos, con los caminos más largos, la diferencia entre los dos algoritmos en el tiempo de ejecución llega a ser de aproximadamente 11.000 veces menor para JPS. Esto tipo de ventaja se produjo en todos los tipos de mapa.

Conclusión

Pathfinding es el área de la inteligencia artificial que busca encontrar el mejor camino de un punto a otro en mapas representados digitalmente. Se han desarrollado distintos algoritmos con el objetivo de aminorar el tiempo y los recursos utilizados en esta tarea. En este proyecto se busca probar el rendimiento de uno de estos algoritmos, JPS (Jump Point Search) desarrollado por Daniel Harabor y Alban Grastien. El algoritmo se compara con A*, utilizando tres distintas heurísticas.

Se explicaron los conceptos básicos de Pathfinding, los enfoques que se pueden tomar, los algoritmos más importantes entre otros aspectos de la teoría. Se incluyó una explicación del principal algoritmo a utilizar para comparado, JPS,

creado por David Harabor y Alban Grastien, que básicamente, se utiliza en conjunto con A* para poder omitir la cantidad de nodos a expandir eliminando simetrías alrededor del camino que se está recorriendo.

Referencias

1. Harabor D. 2012, Fast Pathfinding via Symmetry Breaking. AIgamedev, Disponible vía web en http://aigamedev.com/open/tutorial/symmetry-in-pathfinding/. (Acceso en 30 /8/ 2013).
2. Harabor, D. and Grastien, A.. Online Graph Pruning for Pathfinding On Grid Maps. 2011 . Proceeding of the national conference on artificial inteligence". 1114-1119
3. Jeremy Christman, “General Pathfinding: Tables and Navigation”, disponible vía web http://www.cse.lehigh.edu/~munoz/CSE348/classes/ChristmanPathfinding.pptx, (Acceso en 30/8/2013).

Otro

```
# coding: utf-8
# Your code here!

import tensorflow as tf
import numpy as np

main (){
 celsius = np.array([-40,-10,0,8,15,22,38], dtype = float)
 fahrenheit = np.array([-40,14,32,46,59,72,100], dtype = float)

 oculta1 = tk.keras.layers.Dense(units = 1, input_shape = [1])
 oculta2 = tk.keras.layers.Dense(units = 3)
 salida = tk.keras.layers.Dense(units = 1)
 modewlo = tk.keras.layers.Dense([oculta1,oculta2,salida])

 modelo.compile{
    optimizer = tk.keras.optimizer.Adam(0.1),
    loss = 'mean_squared_error'
 }

 print("Comenzando entrenamiento...")
 historia = modelo.fit(celsius,fahrenheit,epochs=1000,verbose=False)
 print("Modelo entrenado!")

 import matplotlib.pyplot as plt
 plt.xlabel("Epoca: ")
 plt.ylabel("Magintud de pérdida: ")
 plt.plot(historia.history["loss"])
```

```
 print("Hagamos una prediccion!")
 resultado = modelo.predict([100.0])
 print("El resultado es " + str(resultado) + "fahrenheit!")

 print("Variable internas del modelo")
 #print(capa.get_weights())
 print(oculta1.get_weights())
 print(oculta2.get_weights())
 print(salida.get_weights())

}
```