8

AI in Games Development

In chapter, the following recipes will be covered:

* Adding artificial intelligence to the game
* Using heuristics in the game
* Using Binary Space Partition Tree
* Creating a decision making AI
* Adding behavioural movements
* Using Neural Networks
* Using Genetic Algorithms
* Using other waypoint systems

# Introduction

Artificial intelligence(AI) can be defined in many ways. Artificial intelligence deals with the study of finding similarities in different situations and difference in similar situations. AI can help in bringing realism to the game. The user playing the game should have a feeling that that entity that he is competing against is another human. Achieving this is extremely difficult and can consume a lot of processing cycles. In fact, there is Turing Test held every year to determine if the AI can fool other humans into believing that it is human. Now if for AI we use a lot of processing cycles, then executing the game at above 40 FPS becomes extremely difficult. Hence we need to write smart algorithms to achieve this.

# Adding artificial intelligence to the game

1. Adding artificial intelligence to a game may be easy or extremely difficult based on what level of realism or complexity we are trying to achieve. In this recipe, we will start with the basics of adding artificial intelligence.

## Getting ready

To step through this recipe, you will need a machine running Windows and a version of Visual Studio. No other prerequisites are required.

## How to do it...

In this recipe we will see how easy it is to add a basic artificial intelligence to the game.

* Add a source file called Source.cpp

// Basic AI : Keyword identification

#include <iostream>

#include <string>

#include <string.h>

std::string arr[] = { "Hello,what is your name ?", "My name is Siri" };

int main()

{

std::string UserResponse;

std::cout << "Enter your question? ";

std::cin >> UserResponse;

if (UserResponse == "Hi")

{

std::cout << arr[0] << std::endl;

std::cout << arr[1];

}

int a;

std::cin >> a;

return 0;

}

## How it works...

In the example above, we are using a string array to store a response. The idea of the software above is to create an intelligent chat bot who can reply to questions asked by users and interact with them as if it was human. Hence the first task was to create an array of responses. Next thing to do, is to ask the user for the question. In this example, we are searching for a basic keyword called “Hi” and based on that we are displaying the appropriate answer. Of course this is a very basic implementation. Ideally we would have a list of keywords and a response when either of the keywords is triggered. We can even personalize this by asking the user for his name and then appending his name with the answer every time.

The user may also ask to search for something. That is actually quite an easy thing to do. If we have detected the word correctly which the user is longing to search, we just need to enter that into the search engine. Whatever result the page displays, we can report it back to the user. We can also use voice commands for entry of the questions and the response. For that case, we would also need to implement some kind of NLP (Natural Language Processing). After the voice command is correctly identified, all the above processes are exactly the same.

# Using heuristic in a game

Adding heuristic in a game is to define rules. We need to define a set of rules for the AI agent so that it can lead to its destination or goal in the best possible way. For example, if we want to write a path finding algorithm, and define only its start and end positions, it may reach there in many different ways. However, if we want the agent to reach the goal in a particular way, we need to establish a heuristic function for it.

## Getting ready

You need a Windows machine and a working copy of Visual Studio. No other pre-requisite is needed.

## How to do it...

In this recipe we will find out how easy it is to add a heuristic function to our game for path finding.

* Add a source file called Source.cpp

int heuristic(GameGrid::Location a, GameGrid::Location b)

{

int x1, y1, x2, y2;

return abs(x1 - x2) + abs(y1 - y2);

}

template<typename Graph>

void a\_star\_search

(const Graph& graph,

typename Graph::Location start,

typename Graph::Location goal,

unordered\_map<typename Graph::Location, typename Graph::Location>& came\_from,

unordered\_map<typename Graph::Location, int>& cost\_so\_far)

{

typedef typename Graph::Location Location;

PriorityQueue<Location> frontier;

frontier.put(start, 0);

came\_from[start] = start;

cost\_so\_far[start] = 0;

while (!frontier.empty()) {

auto current = frontier.get();

if (current == goal) {

break;

}

for (auto next : graph.neighbors(current)) {

int new\_cost = cost\_so\_far[current] + graph.cost(current, next);

if (!cost\_so\_far.count(next) || new\_cost < cost\_so\_far[next]) {

cost\_so\_far[next] = new\_cost;

int priority = new\_cost + heuristic(next, goal);

frontier.put(next, priority);

came\_from[next] = current;

}

}

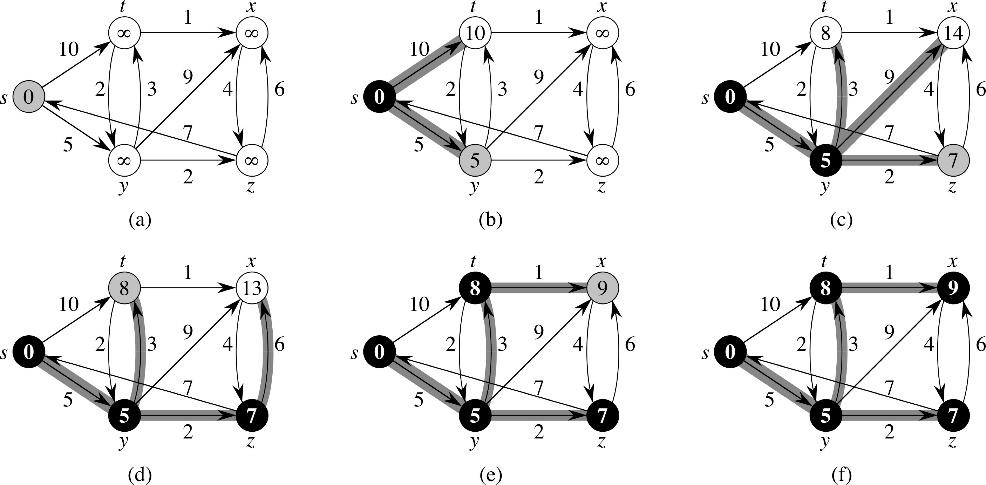
}

}

## How it works...

There are many ways to define what heuristic is. However, the simplest way to think about it is that it is a function which provides hints and directions for the AI to reach the goal. Let us say that our AI needs to go from point A to point D. Now there are also point B and C somewhere on the map. How should the AI decide which path to take? That is what is provided by a heuristic function. In the above example, we have used heuristic in a path finding algorithm called A\*. A special case when the heuristic function is 0, we get an algorithm called Dijkstra’s.

Let us consider Dijkstra’s first. It will be easier to understand A\* later.



Let us consider we need to find the shortest path between s to x traversing all nodes at least once. s,t,y,x,z are the different nodes or the different sub-destinations. The numbers from one node to another node is the cost of going from one node to other. The algorithm states that we start from s with a 0 value and consider all other nodes to be infinity. Next thing to consider is the adjacent nodes to s. The adjacent nodes to s are t and y. The cost of reaching them are 5 and 10 respectively. Hence we note that and then replace the infinity value at those nodes with 5 and 10. Now let us consider the node y. The adjacent nodes are t, x and z. The cost to reach x is 5(its current node values) + 9 (path cost value) = 14. Similarly cost to reach z is 5+2 =7. So we replace the infinity value of x and z with 14 and 7 respectively. Now the cost to reach t is 5 +3 = 8. However, it already has a node value. Its value is 10. Since 8<10, we will replace t with 8. We keep on doing this for all the nodes. A\* is defined by two cost functions. The first cost function determines the cost of all the paths. So this is same as that of Dijkstra’s algorithm. The second cost function is used to determine the path from source node to goal node. Here we need to add some kind of heuristic or guess what the cost of the next path should be. However, we have to remember not to overestimate this cost. This is often referred as admissible heuristic. In A\* algorithm unlike Dijkstra’s algorithm, we do not need to traverse all the nodes. At every node, a decision is to be made whether that node is recommended to be added to the list of nodes that the AI agent should traverse from reaching from source to destination. So writing a good heuristic function is absolutely essential for a good A\* algorithm. There are also other algorithms written to improve on the A\* algorithm, however the one discussed above is the most basic form of A\* algorithm.

In the above example, we have used neighboring nodes and formed a priority list to decide that.

# Using Binary Space Partition Tree

In games sometimes we work with a lot of geometry and huge 3D worlds. If our game camera was to render all of it all the time, then it would be extremely expensive and the game would not be able to run smoothly at higher frame-rates. Hence we need to write intelligent algorithms so that the world if divided into more manageable chunks and which can be traversed easily using a tree structure.

## Getting ready

1. You need to have a working Windows machine and a working copy of Visual Studio.

## How to do it...

* Add a source file called Source.cpp

Code Snippet

//

// struct sTreeAdaptor

// {

// eBspRelation Classify(

// const T\_PlaneDiv& plane,

// const T\_elementType& elem ) const;

//

// void Split(

// const T\_PlaneDiv& plane,

// const T\_elementType& elem,

// T\_elementType\* pFront,

// T\_elementType\* pBack ) const;

//

// void ChooseHyperplane(

// std::vector<T\_elementType>& toProcess,

// T\_PlaneDiv\* pPlane ) const;

// };

// Returned by the Classify method of the tree adaptor.

enum eBspRelation

{

BSP\_FRONT\_SURFACE,

BSP\_BACK\_SURFACE,

BSP\_SPLIT,

BSP\_COPLANAR

};

// This is a non-brep tree. Used just for classification of

// polygons. No leaves.

template< class T\_elementType, class T\_PlaneDiv, class T\_Adaptor >

class cBSPTreePart

{

// Other objects need to see what a node looks like.

public:

struct sNode

{

typedef std::vector< T\_elementType > elemVec;

sNode\* m\_pFrontSurface;

sNode\* m\_pBackSurface;

elemVec m\_values;

T\_PlaneDiv m\_plane;

sNode(elemVec& toProcess, const T\_Adaptor& adap)

: m\_pFrontSurface(NULL)

, m\_pBackSurface(NULL)

{

// Setup

elemVec frontVec, backVec;

frontVec.reserve(toProcess.size());

backVec.reserve(toProcess.size());

// Choose which node we're going to use.

adap.ChooseHyperplane(toProcess, &m\_plane);

// Iterate across the rest of the polygons

elemVec::iterator iter = toProcess.begin();

for (; iter != toProcess.end(); ++iter)

{

T\_elementType front, back;

switch (adap.Classify(m\_plane, \*iter))

{

case BSP\_FRONT\_SURFACE:

frontVec.push\_back(\*iter);

break;

case BSP\_BACK\_SURFACE:

backVec.push\_back(\*iter);

break;

case BSP\_COPLANAR:

m\_values.push\_back(\*iter);

break;

case BSP\_SPLIT:

adap.Split(m\_plane, \*iter, &front, &back);

frontVec.push\_back(front);

backVec.push\_back(back);

break;

default: break;

}

}

// Now recurse if necessary

if (!frontVec.empty())

m\_pFrontSurface = new sNode(frontVec, adap);

if (!backVec.empty())

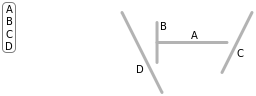
m\_pBackSurface = new sNode(backVec, adap);

}

## How it works...

A BSP (Binary Space Partition) tree as the name implies is a tree structure in which a geometrical space is partitioned. To be more precise, in BSP a plane is portioned into more hyperplanes. A plane is such that it has one dimension less than the ambient space from which it was created. So a 3D plane would have 2D hyperplanes and a 2D planes would have 1D lines. The idea behind this is once we have divided the planes into these hyper-planes in a logical manner we can save the formation into a tree structure. Finally, we can traverse the tree structure in real time to provide better frame rates to the game overall.

Let us consider the following example in which the world looks like the image below. The camera must decide which are the areas that it should render and which it should not. Hence dividing them using a logical algorithm is necessary.



After we apply the algorithm, the tree structure should look like the following.



Finally, we traverse this algorithm like any other tree structure using the concept of parent and child and we get the desired sections that the camera should render.

# Creating a decision making AI

A decision tree is one of the most useful in machine learning in AI. Given a large number of scenarios, based on certain parameters, it needs to take a decision. If we can write a system that can make these decisions well, then we can not only have a well-written algorithm but also have a lot of unpredictability in terms of game-play. This will add a lot of variation in the game and will help in replay ability of the overall game.

## Getting ready

For this recipe, you will need a Windows machine and Visual Studio. No other pre requisite is needed.

## How to do it...

1. In this recipe we will find out how easy it is to create a decision making AI
2. /\* Decision Making AI\*/
3. #include <iostream>
4. #include <ctime>
5. using namespace std;
6. class TreeNodes
7. {
8. public:
9. //tree node functions
10. TreeNodes(int nodeID/\*, string QA\*/);
11. TreeNodes();
12. virtual ~TreeNodes();
13. int m\_NodeID;
14. TreeNodes\* PrimaryBranch;
15. TreeNodes\* SecondaryBranch;
16. };
17. //contrctor
18. TreeNodes::TreeNodes()
19. {
20. PrimaryBranch = NULL;
21. SecondaryBranch = NULL;
22. m\_NodeID = 0;
23. }
24. //deconstructor
25. TreeNodes::~TreeNodes()
26. { }
27. //Step 3! Also step 7 hah!
28. TreeNodes::TreeNodes(int nodeID/\*, string NQA\*/)
29. {
30. //create tree node with a specific node ID
31. m\_NodeID = nodeID;
32. //reset nodes/make sure! that they are null. I wont have any funny business #s -\_-
33. PrimaryBranch = NULL;
34. SecondaryBranch = NULL;
35. }
36. //the decision tree class
37. class DecisionTree
38. {
39. public:
40. //functions
41. void RemoveNode(TreeNodes\* node);
42. void DisplayTree(TreeNodes\* CurrentNode);
43. void Output();
44. void Query();
45. void QueryTree(TreeNodes\* rootNode);
46. void PrimaryNode(int ExistingNodeID, int NewNodeID);
47. void SecondaryNode(int ExistingNodeID, int NewNodeID);
48. void CreateRootNode(int NodeID);
49. void MakeDecision(TreeNodes\* node);
50. bool SearchPrimaryNode(TreeNodes\* CurrentNode, int ExistingNodeID, int NewNodeID);
51. bool SearchSecondaryNode(TreeNodes\* CurrentNode, int ExistingNodeID, int NewNodeID);
52. TreeNodes\* m\_RootNode;
53. DecisionTree();
54. virtual ~DecisionTree();
55. };
56. int random(int upperLimit);
57. //for random variables that will effect decisions/node values/weights
58. int random(int upperLimit)
59. {
60. int randNum = rand() % upperLimit;
61. return randNum;
62. }
63. //constructor
64. //Step 1!
65. DecisionTree::DecisionTree()
66. {
67. //set root node to null on tree creation
68. //beginning of tree creation
69. m\_RootNode = NULL;
70. }
71. //destructor
72. //Final Step in a sense
73. DecisionTree::~DecisionTree()
74. {
75. RemoveNode(m\_RootNode);
76. }
77. //Step 2!
78. void DecisionTree::CreateRootNode(int NodeID)
79. {
80. //create root node with specific ID
81. // In MO, you may want to use thestatic creation of IDs like with entities. depends on how many nodes you plan to have
82. //or have instantaneously created nodes/changing nodes
83. m\_RootNode = new TreeNodes(NodeID);
84. }
85. //Step 5.1!~
86. void DecisionTree::PrimaryNode(int ExistingNodeID, int NewNodeID)
87. {
88. //check to make sure you have a root node. can't add another node without a root node
89. if (m\_RootNode == NULL)
90. {
91. cout << "ERROR - No Root Node";
92. return;
93. }
94. if (SearchPrimaryNode(m\_RootNode, ExistingNodeID, NewNodeID))
95. {
96. cout << "Added Node Type1 With ID " << NewNodeID << " onto Branch Level " << ExistingNodeID << endl;
97. }
98. else
99. {
100. //check
101. cout << "Node: " << ExistingNodeID << " Not Found.";
102. }
103. }
104. //Step 6.1!~ search and add new node to current node
105. bool DecisionTree::SearchPrimaryNode(TreeNodes \*CurrentNode, int ExistingNodeID, int NewNodeID)
106. {
107. //if there is a node
108. if (CurrentNode->m\_NodeID == ExistingNodeID)
109. {
110. //create the node
111. if (CurrentNode->PrimaryBranch == NULL)
112. {
113. CurrentNode->PrimaryBranch = new TreeNodes(NewNodeID);
114. }
115. else
116. {
117. CurrentNode->PrimaryBranch = new TreeNodes(NewNodeID);
118. }
119. return true;
120. }
121. else
122. {
123. //try branch if it exists
124. //for a third, add another one of these too!
125. if (CurrentNode->PrimaryBranch != NULL)
126. {
127. if (SearchPrimaryNode(CurrentNode->PrimaryBranch, ExistingNodeID, NewNodeID))
128. {
129. return true;
130. }
131. else
132. {
133. //try second branch if it exists
134. if (CurrentNode->SecondaryBranch != NULL)
135. {
136. return(SearchSecondaryNode(CurrentNode->SecondaryBranch, ExistingNodeID, NewNodeID));
137. }
138. else
139. {
140. return false;
141. }
142. }
143. }
144. return false;
145. }
146. }
147. //Step 5.2!~ does same thing as node 1. if you wanted to have more decisions,
148. //create a node 3 which would be the same as this maybe with small differences
149. void DecisionTree::SecondaryNode(int ExistingNodeID, int NewNodeID)
150. {
151. if (m\_RootNode == NULL)
152. {
153. cout << "ERROR - No Root Node";
154. }
155. if (SearchSecondaryNode(m\_RootNode, ExistingNodeID, NewNodeID))
156. {
157. cout << "Added Node Type2 With ID " << NewNodeID << " onto Branch Level " << ExistingNodeID << endl;
158. }
159. else
160. {
161. cout << "Node: " << ExistingNodeID << " Not Found.";
162. }
163. }
164. //Step 6.2!~ search and add new node to current node
165. //as stated earlier, make one for 3rd node if there was meant to be one
166. bool DecisionTree::SearchSecondaryNode(TreeNodes \*CurrentNode, int ExistingNodeID, int NewNodeID)
167. {
168. if (CurrentNode->m\_NodeID == ExistingNodeID)
169. {
170. //create the node
171. if (CurrentNode->SecondaryBranch == NULL)
172. {
173. CurrentNode->SecondaryBranch = new TreeNodes(NewNodeID);
174. }
175. else
176. {
177. CurrentNode->SecondaryBranch = new TreeNodes(NewNodeID);
178. }
179. return true;
180. }
181. else
182. {
183. //try branch if it exists
184. if (CurrentNode->PrimaryBranch != NULL)
185. {
186. if (SearchSecondaryNode(CurrentNode->PrimaryBranch, ExistingNodeID, NewNodeID))
187. {
188. return true;
189. }
190. else
191. {
192. //try second branch if it exists
193. if (CurrentNode->SecondaryBranch != NULL)
194. {
195. return(SearchSecondaryNode(CurrentNode->SecondaryBranch, ExistingNodeID, NewNodeID));
196. }
197. else
198. {
199. return false;
200. }
201. }
202. }
203. return false;
204. }
205. }
206. //Step 11
207. void DecisionTree::QueryTree(TreeNodes\* CurrentNode)
208. {
209. if (CurrentNode->PrimaryBranch == NULL)
210. {
211. //if both branches are null, tree is at a decision outcome state
212. if (CurrentNode->SecondaryBranch == NULL)
213. {
214. //output decision 'question'
215. ///////////////////////////////////////////////////////////////////////////////////////
216. }
217. else
218. {
219. cout << "Missing Branch 1";
220. }
221. return;
222. }
223. if (CurrentNode->SecondaryBranch == NULL)
224. {
225. cout << "Missing Branch 2";
226. return;
227. }
228. //otherwise test decisions at current node
229. MakeDecision(CurrentNode);
230. }
231. //Step 10
232. void DecisionTree::Query()
233. {
234. QueryTree(m\_RootNode);
235. }
236. ////////////////////////////////////////////////////////////
237. //debate decisions create new function for decision logic
238. // cout << node->stringforquestion;
239. //Step 12
240. void DecisionTree::MakeDecision(TreeNodes \*node)
241. {
242. //should I declare variables here or inside of decisions.h
243. int PHealth;
244. int MHealth;
245. int PStrength;
246. int MStrength;
247. int DistanceFBase;
248. int DistanceFMonster;
249. ////sets random!
250. srand(time(NULL));
251. //randomly create the numbers for health, strength and distance for each variable
252. PHealth = random(60);
253. MHealth = random(60);
254. PStrength = random(50);
255. MStrength = random(50);
256. DistanceFBase = random(75);
257. DistanceFMonster = random(75);
258. //the decision to be made string example: Player health: Monster Health: player health is lower/higher
259. cout << "Player Health: " << PHealth << endl;
260. cout << "Monster Health: " << MHealth << endl;
261. cout << "Player Strength: " << PStrength << endl;
262. cout << "Monster Strength: " << MStrength << endl;
263. cout << "Distance Player is From Base: " << DistanceFBase << endl;
264. cout << "Disntace Player is From Monster: " << DistanceFMonster << endl;
266. if (PHealth > MHealth)
267. {
268. }
269. else
270. {
271. }
272. if (PStrength > MStrength)
273. {
274. }
275. else
276. {
277. }
278. //recursive question for next branch. Player distance from base/monster.
279. if (DistanceFBase > DistanceFMonster)
280. {
281. }
282. else
283. {
284. }
285. }
286. void DecisionTree::Output()
287. {
288. //take repsective node
289. DisplayTree(m\_RootNode);
290. }
291. //Step 9
292. void DecisionTree::DisplayTree(TreeNodes\* CurrentNode)
293. {
294. //if it doesn't exist, don't display of course
295. if (CurrentNode == NULL)
296. {
297. return;
298. }
299. //////////////////////////////////////////////////////////////////////////////////////////////////
300. //need to make a string to display for each branch
301. cout << "Node ID " << CurrentNode->m\_NodeID << "Decision Display: " << endl;
302. //go down branch 1
303. DisplayTree(CurrentNode->PrimaryBranch);
304. //go down branch 2
305. DisplayTree(CurrentNode->SecondaryBranch);
306. }
307. void DecisionTree::RemoveNode(TreeNodes \*node)
308. {
310. if (node != NULL)
311. {
312. if (node->PrimaryBranch != NULL)
313. {
314. RemoveNode(node->PrimaryBranch);
315. }
316. if (node->SecondaryBranch != NULL)
317. {
318. RemoveNode(node->SecondaryBranch);
319. }
320. cout << "Deleting Node" << node->m\_NodeID << endl;
321. //delete node from memory
322. delete node;
323. //reset node
324. node = NULL;
325. }
326. }
327. int main()
328. {
329. //create the new decision tree object
330. DecisionTree\* NewTree = new DecisionTree();
331. //add root node the very first 'Question' or decision to be made
332. //is monster health greater than player health?
333. NewTree->CreateRootNode(1);
334. //add nodes depending on decisions
335. //2nd decision to be made
336. //is monster strength greater than player strength?
337. NewTree->PrimaryNode(1, 2);
338. //3rd decision
339. //is the monster closer than home base?
340. NewTree->SecondaryNode(1, 3);
341. //depending on the weights of all three decisions, will return certain node result
342. //results!
343. //Run, Attack,
344. NewTree->PrimaryNode(2, 4);
345. NewTree->SecondaryNode(2, 5);
346. NewTree->PrimaryNode(3, 6);
347. NewTree->SecondaryNode(3, 7);
349. NewTree->Output();
350. //ask/answer question decision making process
351. NewTree->Query();
352. cout << "Decision Made. Press Any Key To Quit." << endl;
353. int a;
354. cin >> a;
355. //release memory!
356. delete NewTree;
357. //return random value
358. //return 1;
359. }

## How it works...

As the name suggests, a decision tree is a subset of the tree data structure. Hence there is root node and two child nodes. The root node will denote a condition and the child nodes will have the probable solutions. In the next depth, those solution nodes will become part of the condition which will lead to two more solution nodes. Hence as the example above shows, the entire structure is modelled on the basis of a tree structure. We have a root node and then primary and secondary nodes. We need to traverse the tree to continuously find the answers to a situation based on the root nodes and the child nodes.

We have also written a query function which will query in the tree structure, what the most probable scenario might be on that situation. That in turn will take the help of a decision function which will add its own level of heuristic combined with the result of the query and generate the output for the solution.

Decision tree are extremely fast because at every scenario we are checking only half the tree. So in effect we have reduced the search space to half. The tree structure also makes it robust so that we can add and remove nodes on the fly as well. This gives us a lot of flexibility and the overall architecture of the game is improved in this case.

# Adding behavioural movements

When we talk of AI in games, after path finding the next most important thing to consider is movements. When does an AI decide that it has to walk, run, jump and slide. The ability to make these decisions fast and correctly will make the AI really competitive in games and extremely difficult to beat. We can do all these with the help of behavioral movements.

## Getting ready

1. For this recipe, you will need a Windows machine and Visual Studio. No other requirement is there.

## How to do it...

In this example you will find out how easy it is to create behavioral movements.

1. Add a source file called Source.cpp
2. /\* Adding Behavorial Movements\*/
3. #include <iostream>
4. using namespace std;
5. class Machine
6. {
7. class State \*current;
8. public:
9. Machine();
10. void setCurrent(State \*s)
11. {
12. current = s;
13. }
14. void Run();
15. void Walk();
16. };
17. class State
18. {
19. public:
20. virtual void Run(Machine \*m)
21. {
22. cout << " Already Running\n";
23. }
24. virtual void Walk(Machine \*m)
25. {
26. cout << " Already Walking\n";
27. }
28. };
29. void Machine::Run()
30. {
31. current->Run(this);
32. }
33. void Machine::Walk()
34. {
35. current->Walk(this);
36. }
37. class RUN : public State
38. {
39. public:
40. RUN()
41. {
42. cout << " RUN-ctor ";
43. };
44. ~RUN()
45. {
46. cout << " dtor-RUN\n";
47. };
48. void Walk(Machine \*m);
49. };
50. class WALK : public State
51. {
52. public:
53. WALK()
54. {
55. cout << " WALK-ctor ";
56. };
57. ~WALK()
58. {
59. cout << " dtor-WALK\n";
60. };
61. void Run(Machine \*m)
62. {
63. cout << " Changing behaviour from WALK to RUN";
64. m->setCurrent(new RUN());
65. delete this;
66. }
67. };
68. void RUN::Walk(Machine \*m)
69. {
70. cout << " Changing behaviour RUN to WALK";
71. m->setCurrent(new WALK());
72. delete this;
73. }
74. Machine::Machine()
75. {
76. current = new WALK();
77. cout << '\n';
78. }
79. int main()
80. {
81. Machine m;
82. m.Run();
83. m.Walk();
84. m.Walk();
85. int a;
86. cin >> a;
87. return 0;
88. }

## How it works...

In this example we have implemented a simple state machine. The state machine is created with the State-Machine design pattern in mind. So the states in this case are walk and run. The objective is that if the AI is walking and then it needs to switch to running, it can do so at runtime. Similarly, if it is running, it can switch to walking at runtime. However, if it is already walking, and a request comes to walk, it should notify its own self that there is no need to change the state.

All these change of states are handled by a class called machine. Hence the name state-machine pattern. The reason why this structure is preferred by many over the traditional state machine design is that all the states need not be defined in one class and then a switch case statement to change states. Although the above method is correct, every additional steps that maybe added to the game would require changing and adding to the same class structure. This is a recipe for bugs and possible disasters in the future. Instead we are going for a more object oriented approach in which every state is a class in itself.

The machine class holds a pointer to the state class and then pushes the request to the appropriate child class of the state. If we need to add the jump state. We do not need to change much in the code. We need to write a new jump class, and add the corresponding functionalities. Because the machine has a pointer to the base class (state), it will correspondingly push the request of jump to the correct derived class.

# Using Neural Network

Artificial Neural networks(ANN) are an advanced form of AI that is used in some games. They may not be directly used in-game; however, they might be used during the production phase to train the AI agents. The neural nets are used mostly as a predicting algorithm. Based on certain parameters, and historical data, what is the most likely decision or attribute that the AI agent will distribute. The ANN is not restricted to games itself, they are used across multiple diverse domains to predict possible outcomes.

## Getting ready

To step through this recipe, you will need a machine running Windows and Visual Studio.

## How to do it...

In this example, we will see how easy it is to write a neural network

1. class neuralNetworkTrainer
2. {
3. //class members
4. //--------------------------------------------------------------------------------------------
5. private:
6. //network to be trained
7. neuralNetwork\* NN;
8. //learning parameters
9. double learningRate; // adjusts the step size of the weight update
10. double momentum; // improves performance of stochastic learning (don't use for batch)
11. //epoch counter
12. long epoch;
13. long maxEpochs;
15. //accuracy/MSE required
16. double desiredAccuracy;
18. //change to weights
19. double\*\* deltaInputHidden;
20. double\*\* deltaHiddenOutput;
21. //error gradients
22. double\* hiddenErrorGradients;
23. double\* outputErrorGradients;
24. //accuracy stats per epoch
25. double trainingSetAccuracy;
26. double validationSetAccuracy;
27. double generalizationSetAccuracy;
28. double trainingSetMSE;
29. double validationSetMSE;
30. double generalizationSetMSE;
31. //batch learning flag
32. bool useBatch;
33. //log file handle
34. bool loggingEnabled;
35. std::fstream logFile;
36. int logResolution;
37. int lastEpochLogged;
38. //public methods
39. //--------------------------------------------------------------------------------------------
40. public:
42. neuralNetworkTrainer( neuralNetwork\* untrainedNetwork );
43. void setTrainingParameters( double lR, double m, bool batch );
44. void setStoppingConditions( int mEpochs, double dAccuracy);
45. void useBatchLearning( bool flag ){ useBatch = flag; }
46. void enableLogging( const char\* filename, int resolution );
47. void trainNetwork( trainingDataSet\* tSet );
48. //private methods
49. //--------------------------------------------------------------------------------------------
50. private:
51. inline double getOutputErrorGradient( double desiredValue, double outputValue );
52. double getHiddenErrorGradient( int j );
53. void runTrainingEpoch( std::vector<dataEntry\*> trainingSet );
54. void backpropagate(double\* desiredOutputs);
55. void updateWeights();
56. };

class neuralNetwork

{

//class members

//--------------------------------------------------------------------------------------------

private:

//number of neurons

int nInput, nHidden, nOutput;

//neurons

double\* inputNeurons;

double\* hiddenNeurons;

double\* outputNeurons;

//weights

double\*\* wInputHidden;

double\*\* wHiddenOutput;

//Friends

//--------------------------------------------------------------------------------------------

friend neuralNetworkTrainer;

//public methods

//--------------------------------------------------------------------------------------------

public:

//constructor & destructor

neuralNetwork(int numInput, int numHidden, int numOutput);

~neuralNetwork();

//weight operations

bool loadWeights(char\* inputFilename);

bool saveWeights(char\* outputFilename);

int\* feedForwardPattern( double\* pattern );

double getSetAccuracy( std::vector<dataEntry\*>& set );

double getSetMSE( std::vector<dataEntry\*>& set );

//private methods

//--------------------------------------------------------------------------------------------

private:

void initializeWeights();

inline double activationFunction( double x );

inline int clampOutput( double x );

void feedForward( double\* pattern );

};

## How it works...

In the above example snippet, we have created the backbone to write a neural network which can predict an alphabet which is drawn on the screen. Many devices and touch screen tablets have this capability to detect an alphabet that you draw on screen. Let us take this and think in terms of game design. If we want to create a game in which we draw shapes, and the corresponding weapon will be given to us which we can use in battle, we can use the above as template to train the agents to identify a shape before the game is released in the market. Generally, games like these only detect the basic shapes. Those can be easily detected and do not require neural nets to train the agents.

In games ANN will mostly be used to create good AI behaviours. However, it is not wise to use ANN while the game is being played as it is expensive and takes a long time to train the agents. Let us look at the example below

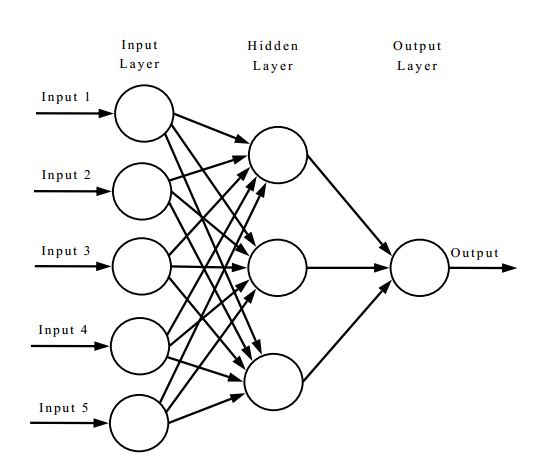
|  |  |  |
| --- | --- | --- |
| Melee | Speed ( 4) | 25 (HP) |
| Archer | Speed (7) | 22 (HP) |
| Magic | Speed (6.4) | 20 (HP) |
| ? | Speed (6.6) | 21 (HP) |

Given the data, what is the most likely class of the unknown.? Now remember the data in this case is just 3, in reality it will be over 100. The number of parameters (Class type, Speed and HP) is just 3, in reality it will over 10. It will be difficult to predict the class by just looking at those number. That’s where ANN comes in. It can predict any of the missing column data based on the other column data and previous historical data. This becomes a very handy tool for the designer to balance the game.

Few concepts of the ANN which we have implemented is necessary to understand.

A typical ANN is composed of 3 primary stages,

* The inter connection between the different neurons that are present in the layers
* The learning process that is required to train the neurons so that the network provides the correct output
* The activation function that is used to trigger or fire the different neurons.



Input Layer: This is the layer where we supply all the column data that is known, historical and new. The process involves first supplying data whose output we already know. This phase is known as the learning phase. There are two types of learning algorithms. Supervised and non-supervised. The explanation for those out of the scope of this book. After that there is some kind of training algorithm that is applied to minimise the errors in the desired output. Back-propagation is one such techniques in which the weights calculating the neural network function is adjusted till we get closed to the desired result. After the network is set and is giving correct results on already known outputs, we can then supply new data and find out the result of the unknown column data.

# Using Genetic algorithms

Genetic algorithms are a method of evolutionary algorithms (EA). They are particularly useful when we want to write prediction algorithms in which only the strongest selection is selected and the rest is rejected. Hence its gets its name. So at every iteration it mutates, does a cross-over and only the best is selected for the next iteration of population. The idea behind genetic algorithms is that after multiple iterations only the best possible candidates are left.

## Getting ready

To step through this recipe, you will need a machine running Windows with an installed version of Visual Studio.

## How to do it...

1. In this recipe we will find out how easy it is to write a genetic algorithm.
2. void crossover(int &seed);
3. void elitist();
4. void evaluate();
5. int i4\_uniform\_ab(int a, int b, int &seed);
6. void initialize(string filename, int &seed);
7. void keep\_the\_best();
8. void mutate(int &seed);
9. double r8\_uniform\_ab(double a, double b, int &seed);
10. void report(int generation);
11. void selector(int &seed);
12. void timestamp();
13. void Xover(int one, int two, int &seed);

## How it works...

Genetic algorithms(GA) may seem extremely difficult to understand or make sense at first. However, GA is extremely simple. Let us think of a situation where we have a land which is filled with dragons with some varied attributes. The objective or goal of the dragon is to defeat a human(player) who has some attributes.

Dragon(AI)

|  |  |  |  |
| --- | --- | --- | --- |
| Dragon 1 | Run | ~~Defend~~ | ~~Attack~~ |
| Dragon 2 | ~~Run~~ | ~~Defend~~ | Attack |
| Dragon 3 | ~~Run~~ | Defend | ~~Attack~~ |

Human(Player)

|  |  |  |  |
| --- | --- | --- | --- |
| Player | Run | Defend | Attack |

So for the Dragon to be competitive against the Human, it must learn how to run, defend and attack. Let us see how GA helps us to do it.

Step 1 (Initial Population)

Dragon(AI)

|  |  |  |  |
| --- | --- | --- | --- |
| Dragon 1 | Run | ~~Defend~~ | ~~Attack~~ |
| Dragon 2 | ~~Run~~ | ~~Defend~~ | Attack |
| Dragon 3 | ~~Run~~ | Defend | ~~Attack~~ |

This is our initial population. Each has its own set of properties. We are just considering 3 dragons. In practise there will be more than that.

Step 2 (Fitness function)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 60% | Dragon 1 | Run | ~~Defend~~ | ~~Attack~~ |
| 75% | Dragon 2 | ~~Run~~ | ~~Defend~~ | Attack |
| 20% | Dragon 3 | ~~Run~~ | Defend | ~~Attack~~ |

The fitness function (%) determines how fit a particular dragon is from the population. 100% is perfect fitness.

Step 3 Cross-over

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 60% | Dragon 1 | Run | ~~Defend~~ | ~~Attack~~ |
| 75% | Dragon 2 | ~~Run~~ | ~~Defend~~ | Attack |
| 20% | Dragon 3 | ~~Run~~ | Defend | ~~Attack~~ |

Based on the fitness function and the attributes that are missing, there will be a cross-over or reproduction phase to create a new dragon with both properties. The dragon with the least fitness function will be removed from the population. (Survival of the fittest).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 60% | Dragon 1 | Run | ~~Defend~~ | ~~Attack~~ |
| 75% | Dragon 2 | ~~Run~~ | ~~Defend~~ | Attack |
| ~~20%~~ | ~~Dragon 3~~ | ~~Run~~ | ~~Defend~~ | ~~Attack~~ |

Step 4 Mutate

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 67% | Dragon 4 | Run | ~~Defend~~ | Attack |

So we have now got a new dragon who can run as well as attack and has a fitness function of 67%. We must now repeat the process (new generation) with other dragons in the population will we are satisfied with the goal. The ideal population will be when all dragons will have the following capability.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 100% | Dragon 4 | Run | Defend | Attack |

However, this may not be always possible. Hence we need to be satisfied it is closer to the goal. All the stages described here are implemented as functions which could be expanded upon based on the requirement of the AI agent.

Now one could ask the question that why don’t we create dragons with all properties in the first place. That’s where adaptive AI comes into place. If we define all properties in the dragons before the user plays the game, it may be very easy to defeat the dragons as the game progress. However, if the AI dragons can adapt based on how the player is defeating the units, it may get progressively difficult to beat the AI. As the player defeats the AI, we need to record the parameters and add that parameter as goal attribute for the dragon which it can achieve after few cross-overs and mutation.

# Using other waypoint systems

Waypoints are a way of writing path finding algorithms. They are extremely easy to write. However, if not thought out properly they can be extremely buggy and the AI can look extremely stupid. Many older games often had this bug which resulted in a revolution to change the implementation of waypoint systems.

## Getting ready

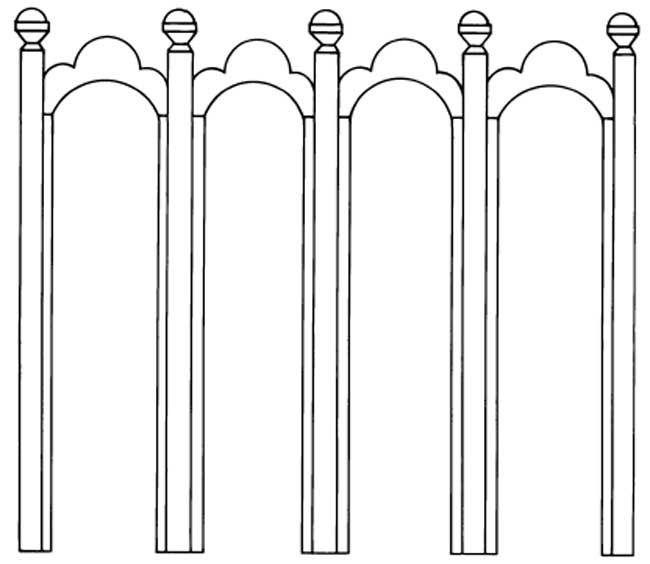
To step through this recipe, you will need a machine running Windows with an installed version of Visual Studio. No other pre-requisites needed.

## How to do it...

1. In this recipe we will find out how easy it is to create waypoint systems.
2. #include <iostream>
3. using namespace std;
4. int main()
5. {
6. float positionA = 4.0f; float positionB = 2.0f; float positionC = -1.0f; float positionD = 10.0f; float positionE = 0.0f;
7. //Sort the points according to Djisktra's
8. //A\* can be used on top of this to minimise the points for traversal
9. //Transform the objects over these new points.
10. return 0;
11. }

## How it works...

In this example we will just discuss a basic implementation of the waypoint system. As the name suggests, waypoints are just 2D/3D points in world space which we want the AI agent to follow. All the agent has to do is move from point A to point B. However, this has complications. For example, let us consider the following diagram below.



To get from A to B is easy. Now to get from B to C, it has to follow a path finding algorithm like A\* or Djikstra’s algorithm. In that case, it will avoid the obstacle in the centre and move towards C. Now let’s say it has suddenly seen the user at point A. How should it react? If we just provide waypoints, it will look at its dictionary of the points that it is allowed to move and which is closest to it. The answer will be A. However, if it starts going towards A, it will be blocked by the all and it may get stuck in a loop of hitting the wall continuously. You may have seen this behaviour a lot in older games. In this case, the AI must make a decision to go back to C and then A. So we can’t use waypoint algorithm on its own. For better performance and efficiency, we need to write a decision making algorithm, a path finding algorithm along with it to give the best results. This is what is used in most modern games along with techniques such as NavMesh and so on.