DM810 - Project 1 report

NavBot - Bot pathfinding in Counter-Strike

Christian Funder Sommerlund
(chsom09 | 260189)

December 2, 2012

1 Introduction

Back in 1998, the first-person shooter video game Half-Life [1] was released. Its game engine[2] was a heavily modified version of the Quake[3] game engine. The game became hugely popular, in part because it was very mod[4]-friendly (especially considering the industry at the time) because Valve (the developer) shipped a pretty extensive SDK[5] alongside the game (in a quality that would be considered questionable today, but an impressive feat at the time being).

The SDK was used by both Valve itself and the community to build mods for the game. One of the free mods built by players, Counter-Strike[6], became one of the most popular first-person shooters ever. Valve aquired the rights to the game in 2000, and it has since sold an additional 27 million copies in various versions and editions. It is now available through their hugely popular digital delivery platform, Steam[7].

Multiplayer is handled by Half-Life and its mods by having players connect to community-driven servers capable of hosting games for up to 32 people simultaneously. As mods need to be able to control and override the default behaviour of the Half-Life gameplay to provide its own game, there is extensive communication between the Half-Life engine and the mod code on the server. In broad terms the engine handles the technical and practical stuff and the mod handles the game logic.

An interesting side effect of mods being able to hook into the Half-Life engine with help from the SDK is that people figured out how to build server-side plugins. These sit as man-in-the-middle between the engine and the mod code and modify the game in various ways. Many plugins were released, so many that in order to handle running several at the same time, a plugin manager named Metamod[8] was released as a "boss-in-the-middle" to take care of coordinating all the function calls going back and forth between the plugins, the mod and the engine.

One of the things that people made plugins for was bots. A guy who called himself botman[9] pioneered these, and released a bot capable of playing not only the original deathmatch multiplayer mode of Half-Life, but also several of the popular official and community mods, including Counter-strike.

Thankfully, he made the source code available which then served as the de-facto framework for most bots for Half-Life and its mods.

Botmans bots, and in turn most (if not all) other bots for the games, find their way around the maps by using waypoint navigation. These are placed in the game world by a player (typically through a user interface provided by the bot code) prior to playing. In the simple case, the player runs around the map spreading waypoints in all areas where the bots should be able to travel[10]. These are then saved on the server for the bots to use during games.

When bots are added to a game on the server, they will use the previously placed waypoints to calculate paths towards their goals. Assuming the bots spawn[11] locations as well as its possible

[1] http://en.wikipedia.org/wiki/Half-Life_(video_game)
[2] http://en.wikipedia.org/wiki/GoldSrc
[3] http://en.wikipedia.org/wiki/Quake_(video_game)
[4] http://en.wikipedia.org/wiki/Mod_(video_gaming)
[5] http://en.wikipedia.org/wiki/Software_development_kit
[6] http://en.wikipedia.org/wiki/Counter-Strike
[7] http://en.wikipedia.org/wiki/Steam_(software)
[8] http://metamod.org/
[9] http://hpb-bot.bots-united.com/
[10] Example: http://www.youtube.com/watch?v=NDRAXv2CuFM
[11] http://en.wikipedia.org/wiki/Spawning_(video_gaming)

goals all lie close to waypoints in the game world, it is trivial to adapt the pathfinding problem to take advantage of popular pathfinding algorithms.

2 Project description

For this project I have taken Botman's HPB bot framework, patched it to support gcc and C++ (the C code unfortunately contained many parts that did not compile without errors and warnings as C++), disabled the pathfinding code (an implementation of the Floyd–Warshall algorithm[12]) and implemented the A* algorithm[13] instead.

When a bot joins the game, it is given a random waypoint as its goal. Once it reaches its goal waypoint, it is given a new random waypoint as its goal. This is repeated forever for the purpose of testing the algorithm and not actually playing the game in a sensible way.

Different bots are given different waypoints, and thus filling the server with 32 bots gives you an army of bots running randomly around (until they meet bots from the opposing team and starts killing each other, that is. Apparently Botman made them quite good at headshotting too).

3 Implementation notes and snippets

3.1 Initialization

All that is done at initialization (map load) is resetting data structures and loading waypoints into a graph. These structures looks as follows:

src/navbot.h

```
17  typedef boost::shared_ptr<Edge> EdgePtr;
18
19  struct Edge
20  {
21    Node * origin;
22    Node * dest;
23    int length;
24
25    Edge(Node * origin, Node * dest, int length) : origin(origin), dest(dest), length(↩
        length) {}
26  };
27
28  struct Node
29  {
30    bool active;
31    int wid;
32    vector<EdgePtr> edges;
33    Vector position;
34
35    Node() : active(false), wid(-1), edges(), position() {}
36  };
```

The *active* boolean of the Node struct simply indicates whether this particular struct is used at the moment or not (a static number of Nodes are intialized and re-used to keep things simple). The *wid* is the waypoint ID number.

[12] http://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm
[13] http://en.wikipedia.org/wiki/A*_search_algorithm

Note the use of Boost shared_ptr[14] for memory management of the edges.

The length of edges are calculated during initialization using standard vector math:

src/navbot.cpp

```
70    EdgePtr edge(new Edge(nodeFrom, nodeTo, pathLength));
```

3.2 Heuristic

The heuristic function for the A* algorithm is the commonly used euclidean distance from the origin to the destination:

src/navbot.cpp

```
109 long navbot_route_heuristic(Node * node1, Node * node2)
110 {
111   return (long)(node1->position - node2->position).Length();
112 }
```

3.3 Route reusing

To save time, routes are reused if the bot's position is anywhere in the last calculated route for the bot, and the goal is still the same:

src/navbot.cpp

```
125   if (botRoutes[botIndex].back() == widTo)
126   {
127     // Goal is still the same. Look for current position in old route
128     list<int>::iterator iter = std::find(botRoutes[botIndex].begin(), botRoutes[↩
        botIndex].end(), widFrom);
129
130     if (iter != botRoutes[botIndex].end())
131     {
132       // Found. Previously calculated route is still usable
133       botRoutes[botIndex].erase(botRoutes[botIndex].begin(), ++iter);
134       return botRoutes[botIndex].front();
135     }
136   }
```

3.4 Private DNode layer

A* pathfinding depends on being able to associate execution specific attributes to nodes during its search, such as current cost and previous node (for backtracking). To logically separate the execution specific data from the global node graph, I have implemented these in their own struct:

src/navbot.h

```
38 struct DNode
39 {
40   DNode * prev;
```

[14] http://www.boost.org/doc/libs/1_52_0/libs/smart_ptr/shared_ptr.htm

```
41    Node * node;
42    long distance;
43    long heuristic;
44    bool visited;
45    bool enqueued;
46
47    bool operator()(DNode * const &lhs, DNode * const &rhs) const
48    {
49      return -(lhs->heuristic) < -(rhs->heuristic);
50    }
51
52    DNode() : prev(NULL), node(NULL), distance(-1), heuristic(-1), visited(false),
        enqueued(false) {}
53  };
```

Instances of the DNode struct are created during pathfinding and then thrown away afterwards, leaving the original graph intact at all times.

Notice the comparison operator which is used by the Boost fibonacci_heap[15] used to queue the nodes during pathfinding. The heuristic values are inversed because the heap is a max-heap and not min-heap as expected by the pathfinding algorithm. As all real (and heuristic) distances are positive, inversion solves the mismatch.

4 Results

To test and get an idea of the performance of my implementation, I waypointed the standard map de_dust with 336 waypoints.

As the routes are reused as long as the bot does not deviate to far from it (because of combat or getting stuck in other bots or the like), the A* algorithm only needs to run about once every 15 seconds per bot.

A single A* execution takes **10-500 *micro*seconds** (including timing and console print overhead) on my Intel Core i5-2520M laptop using a single core, averaging ~ 100 **microseconds** or **10.000 pathfindings per second**.

[15] http://www.boost.org/doc/libs/1_52_0/doc/html/heap.html