JoeBOT XP - An Overview
-= BETA =-

Johannes Lampel
johannes@lampel.net
http://www.lampel.net

October 17, 2004

1 Abstract

This paper describes some of the ideas I want to realize within my new bot, temporary called ***JoeBOT XP*** *. Showing differences to the old bot project,* ***JoeBOT*** *, I want to show what lead to certain ideas, what is the purpose of different parts of the code, where had I to make compromises, what is not yet implemented.*
The sourcecode of ***JoeBOT*** *, the old bot, is OpenSource, you can download it from*

http://sourceforge.net/projects/joebot

bliblablop, more to come here

Contents

2 Motivation

In October 2000 I started to code ***JoeBOT*** , a bot for Counterstrike. At that point of time I haven't had much experience with projects of that size, i.e. round about a megabyte of sourcecode. After two years of development and having learned a lot of new techniques, it became harder and harder to extend the AI of the bot, because it had no clear structure, there were a lot of different variables and functions interacting with one another in a more or less undefined, nonstandardized way. There were a lot of features which were not carefully coded, sometimes a user suggested something which looked promising and I implemented it right away without thinking much about it or they were just evil hacks to get rid of a bug. Finally in summer 2002, I decided to restart a new bot from scratch. We had organized a bot developer meeting of 7 developers from different countries in Amsterdam and talked about a lot of new ideas, about some more or less philosophic questions like "Are waypoints a method to simulate a human way of navigating ?" and a lot more. I also got insight how nice A* worked in Killaruna's ***parabot*** and that was maybe the main reason to restart. I realized that I would have to rewrite almost the whole bot if I replaced the Floyd Warshall algorithm with A*. Additionally there was the idea of a bot being more engine and/or mod independent, which would have been difficult with the old bot. From this point on old ***JoeBOT*** has only been debugged and tweaked at some points and ***JoeBOT XP*** started to grow up.

3 Goals

Main goals for developing ***JoeBOT XP*** are :

- *A clear structure for **all** elements of the bot*
  Learning from old ***JoeBOT*** , I decided that the best way to restrict access to data and to subdivide data into reasonable pieces, therefore ***JoeBOT XP*** shall be object orientated.
  For example, behaviours shouldn't be just fixing of some basic behaviour, but stated clear and being easy to extend.

- *dynamic pathfinding*
  The Floyd Warshall Tables in ***JoeBOT*** were static by nature. Varying paths was done by searching paths to some abitrary point in between the end and the start waypoint

and then selecting the best way based on some some random influences and the current constitution of the bot, i.e. if it is aggressive, defensive, already weakened, etc.
Using A* as the pathfinding algorithm has several advantages : It is possible to define the exact costs of a path, so that you don't have to imitate dynamic pathfinding by analysing a lot of shortest distance paths, you don't have to have a static waypoint grid on which the pathfinding runs, and finally, A* can be used in some other creative ways, e.g. to estimate a possible position of an enemy. A disadvantage of A* is that is *a lot* slower than a simple table lookup like we had with the Floyd Warshall algorithm. Finding a path in a 700 node mesh might take up to 3ms with a simple cost and heuristic function on a Pentium4 2,66GHz. Looking up a value in a table is 4-6 orders of magnitude faster. Another disadvantage is that it isn't always easy to define appropriate cost and heuristic functions for different situations.

- *dynamic waypoint recording*
  A main problem of all bots is to obtain information about their environment in a suitable way. The bot has to neglect uneccessary data, but it should not get stuck in every wall, though. The first generation of the popular Counterstrike bots did this by requiring handmade waypoint files containing all the information the bot needed for navigation. Those bots weren't able to navigate sufficiently well on unknown maps. Ways to navigate throught the map without waypoints have not been really sucessful. Recently, there are attempts to use the BSP data of a map to obtain those information. This is often a complicated process, and I simply haven't had the time to handle this topic, therefore I decided to create the waypoints on the fly as the players (bots and humans) move around. When there is a little number of waypoints the bots have some basic navigation system which allows them to explore the map, and if there are humans playing, it is even better.

- *mod-independency*
  Wouldn't it be cool if we had a bot, which, once coded, would run on almost all mods for some game, only by defining some basics of the game ? Yeah, I thought so too ...

- *engine-independency*
  The same here ... Who says that I'll still continue to code a bot for Halflife ? There are rants that there will be a sucessor of HL in the far future, so it be great if only the interface of the bot had to be changed to make it compatible. Or even for making it compatible with another first person shooter, if Valve keeps nagging on the 56k users with their f**king Steam system.

4 Implementation

4.1 Useful little things

Bitfields: Storing a lot of bool values in a simple boolean array is pretty simple, but uneffective. For one stored bit, you waste at least seven bits. And if you want to do boolean operations on those arrays, you have to loop throught all of them, 'or'-ing each of those bits one by one. Simple case for creating a new data type, no question - then we can also get rid of those bytewasting arrays. Instead we create an array with an appropriate size of *long* values. On current systems, that's 32 bit. We just have to write some functions to read and set/reset single bits and overload some operators and we are done. 'Or'-ing two bitfields now needs less memory bandwidth and 32 bits can be processed at once.

In-Game Realtime Profiler Often it is very useful where most of the time of your calculation time is spent. There are different ways to do this : You can use third party programs like the AMD CodeAnalyst, or you can use an own, in game, realtime profiler.
Third party profiler like the AMD CodeAnalyst are easy to use, have a lot of functions to analyse the resulting data and multible possibilities to export the data. Those programs often work by sampling where the execution of the program is with a fixed frequency. This produces a lot of data and eventually needs a lot of additional processing time, while they are not always successful in analysing functions which are only executed during a shot timespan, because the sampling rate is relatively low, otherwise we would just get too much data. To get a better resolution, the time of measurement needs to be extended. Unfortunately those average values do not neccessarily show functions using a lot of time only at certain moments.
The profiler of ***JoeBOT XP*** works another way (based on *Steven Rabin / Game Programming Gems 1 / 'Real-Time In-Game Profiling'*) : We have one global instance of the profiler for managing all data, being called by so called *CProfileItem* instances. Those *CProfileItem* instances do a profiler call when being created, i.e. from their destructor, and call a function of the global profiler instance when destructed. This way we can just create an instance of such a *CProfilerItem* in a function to be profiled. Every time this function is called, the variable is instantiated, therewith the time measurement started. When we exit that function, the variable is destructed and the time measurement is done. The task of the main profiler class is to collect this data, to maintain a tree representing the calling hirarchy and to do some simple calculation showing the average time spent, maximum time etc. Finally

we have functions to implement some cursor like system to browse the tree and a function to output this text to a buffer.

```
class CProfileStats{
public:
    int m_iCalls;
    _int64 m_i64TotalTime;
    float m_fPercentage;
};

class CProfileNode{
    friend class CProfiler;
public:
    CProfileNode();
    ~CProfileNode();

    CProfileNode *hasChild(const char *);

    void begin(void);
    void end(void);

    void calcStats(void);
    void resetLogData(void);
    void output(char *,int,bool);

protected:
    unsigned int m_iProfileInstances;       //# of times ProfileBegin called
    int m_iOpenProfiles;                    //# of times ProfileBegin w/o ProfileEnd
    _int64 m_i64StartTime;                  //The current open profile start time
    _int64 m_i64Accumulator;                //All samples this frame added together

    CProfileNode *m_pParent;
    vector<CProfileNode *> m_ppChildren;    // children of this node

    char m_szName[80];                  // name of current profiler node

    CProfileStats m_Stats;
    bool m_bOpened;
};

// just a class for the instances created for profiling
class CProfileItem{
public:
    CProfileItem(char *);
    ~CProfileItem();
protected:
    char m_szName[80];
};

// main profiler class
class CProfiler{
public:
    CProfiler();
    virtual ~CProfiler();
    void startProfile(void);

    void frame(void);   // called each frame to reset data

    bool begin(const char*);            // begin profile item
    bool end(const char *);             // end profile item

    // output information to a text array
    void profileDumpOutputToBuffer( char *szBuffer, CProfileNode *pNode = 0 );
    float createPath(CProfileNode *,char *);    // calculate the percentage
                    //of the calculation time we have here

    void cursor_movein(void);
    void cursor_moveout(void);
    void cursor_movedown(void);
```

```
    void cursor_moveup(void);

    CProfileNode *getCursor(void);

protected:
    _int64 m_i64endProfile;
    _int64 m_i64startProfile;

    vector<CProfileNode *> m_pNodes;
    CProfileNode *m_pCurrNode;
    CProfileNode *m_pRoot;
    CProfileNode *m_pCursor;

    _int64 m_i64LastStat;              // when was the last stat copy ?
};

extern CProfiler g_Profiler;

#define PROFILE(a) CProfileItem _profileiteminstance__profile(a)
```

4.2 Engine Interface

To make it easier to interface the structs used by the engine, I created two wrapper classes until now. One, the *CEntity* class, handles general entities. Using this class, there is no need anymore to access the *'entvars_t'* member of the Halflife engine interface directly. This may come in use once there are changes in the engine interface or if you want to port the bot for some other engine. Engine functions like *pfnFindEntityInSphere* have now a wrapper function using this class instead of *'edict_t *'*.
There is a global pointer pointing at a class derived from *CGame*. This instance organizes players, checks if a bot should join, handles commands, provides functions to search players, etc. This class contains an array with instances of *CPlayer*, which is derived from *CEntity*. Classes derived from *CGame* can be used to customize these organisatoric functions for different mods. The same is the case for the *CPlayer* class. There may be a difference in detecting the team of a bot. In most mods, you can just check a *'entvars_t'* variable calling the respective *CEntity* function, but in Counterstrike e.g. you have to do it another way. This can be perfectly done using inhertitance, virtual functions and polymorphism.

```
class CEntity  {
public:
    CEntity();
    CEntity(edict_t *p){m_pEntity = p;}
    operator edict_t*(){return m_pEntity;}
    virtual ~CEntity();

    void setEntity(edict_t *pEntity){m_pEntity = pEntity;}
    edict_t *getEntity(void)const{return m_pEntity;}
    edict_t *getContainingEntity(void)const{return m_pEntity->v.pContainingEntity;}

    long isBot(void)const       {return m_pEntity->v.flags&FL_FAKECLIENT;}
    bool isAlive(void)const     {if(m_pEntity){return ((m_pEntity->v.deadflag == DEAD_NO)
```

```
        &&(m_pEntity->v.health > 0) && (m_pEntity->v.movetype != MOVETYPE_NOCLIP));}else return false;}

    int getDefaultFOV(void)const{return 90;}
    Vector& getVelocity(void)   {return ((Vector &)(m_pEntity->v.velocity));}
    Vector& getOrigin(void)     {return ((Vector &)(m_pEntity->v.origin));}
    Vector& getVAngle(void)     {return ((Vector &)(m_pEntity->v.v_angle));}
    Vector& getAngles(void)     {return ((Vector &)(m_pEntity->v.angles));}
    Vector& getVOffset(void)    {return ((Vector &)(m_pEntity->v.view_ofs));}
    Vector& getPunchAngle(void) {return ((Vector &)(m_pEntity->v.punchangle));}
    Vector getGunPosition(void) {return (getOrigin() + getVOffset());}
    long isAttacking(void)const {return m_pEntity->v.button & IN_ATTACK;}
    long isAttacking2(void)const{return m_pEntity->v.button & IN_ATTACK2;}
    long isDucking(void)const   {return m_pEntity->v.button & IN_DUCK;}
    long isJumping(void)const   {return m_pEntity->v.button & IN_JUMP;}

    bool isInFOV(const Vector &);
    bool isInLOS(const Vector &);
    bool isInFOVof(CEntity *);
    bool isInLOSof(CEntity *);
    bool views(const Vector &);

    bool isOnLadder(void)const  {return m_pEntity->v.movetype == MOVETYPE_FLY;}
    int getWaterlevel(void)const{return m_pEntity->v.waterlevel;}
    int getHealth(void)const    {return m_pEntity->v.health;}
    const char *getWeaponModel(void)    {return STRING(m_pEntity->v.weaponmodel);}
    const char *getClassName(void)  {return STRING(m_pEntity->v.classname);}
    const char *getName(void)   {return STRING(m_pEntity->v.netname);}
    const char *getModel(void)  {return STRING(m_pEntity->v.model);}
    virtual int getTeam(void)   {return m_pEntity->v.team;}
    CEntity getOwner(void)      {return CEntity(m_pEntity->v.owner);}

    operator ==(CEntity p){return getEntity() == p.getEntity();}
    const CEntity & operator =(edict_t *p){m_pEntity = p;return ((const CEntity &)(*this));}
private:
    edict_t *m_pEntity;
};

class CAction  {
public:
    CAction();
    virtual ~CAction();

    void reset(void);

    bool jump(void);                // jump, check if it's useful again, and if it is, jump again
    bool reload(void);              // reload
    bool duck(void);                // duck for a .1s
    bool duck(float);               // duck till a certain POT
    bool duckO(float);
    bool pauseAttack(float);        // pause attack till a certain POT
    bool pauseAttackO(float);       // pause for param seconds
    bool slowDown(float fSDown,float fFactor);
    bool slowDownO(float fSDown,float fFactor);
    bool attack(void);              // attack for .1s
    bool attack(float);             // attack till
    bool attackO(float);

    bool attack2(void);             //  like attack, only for attack2
    bool attack2(float);
    bool attack2O(float);

    bool use(void);             // use for .1s
    bool use(float);            // use till
    bool useO(float);

    void pressButton(long lB){m_lPressButton |= lB;}
    void releaseButton(long lB=0xffffffff){m_lPressButton &=~ lB;}// default is releasing all buttons

    void runTo(const Vector &);
    const Vector &getRunDir(void){return m_VRunDir;}    // normalized run direction

    void setMaxSpeed(float fSpeed)      {m_fMaxSpeed = fSpeed;};
```

```
    float getMaxSpeed(void)             {return m_fMaxSpeed;};
    float getSpeed(void)                {return m_fSpeed;};
    void setStrafe(float fStrafe)       {m_fStrafe = fStrafe;}
    float getStrafe(void)               {return m_fStrafe;}

    void setEntity(CEntity pEntity)    {m_pEntity = pEntity;};
    CEntity getEntity(void)            {return m_pEntity;};

    virtual void move(void)=0;
    virtual void look(void)=0;
    virtual void lookTo(const Vector &)=0;

protected:
    float m_fMaxSpeed;          // max speed
    float m_fSpeed;             // current speed
    float m_fStrafe;
    float m_fLastJump;          // time of last jump
    float m_fJumpPause;         // pause between jumps  // more or less hardcoded
    float m_fDuck;              // duck till
    float m_fSlower;
    float m_fSlowDownF;         // slow down factor till time > m_fSlower

    float m_fAttackPause;       // time till atack should be paused
    float m_fAttackTill;
    float m_fAttack2Till;
    float m_fUseTill;

    bool m_bJump;

    Vector m_VRunDir;           // this is normalizes run direction
    Vector m_VRunDirAngles;
    long m_lPressButton;        // press this buttons till it's changed

    float m_fmsecStart;
    float m_fmsecCount;
    float m_fCurrentMSec;

    Vector m_VIdealAngles,
        m_VIdealVAngle;

    float m_fViewSpeedYaw;
    float m_fViewSpeedPitch;
    float m_fAngleSpeedPitch;
    float m_fAngleSpeedYaw;

    CEntity m_pEntity;
};
```

4.3 Waypoint Mesh

This section describes how the waypoint data is represented and which information are stored. The bot's navigation is based on waypoints, therefore the waypoint system is a central part of the bot. This system has to be fast, easy to save and load and extendable, maybe there'll be the need to implement some BSP based approach later. Additionally it should allow fast insertion of new waypoints, which is needed for the waypoint recording. For each waypoint we want to store :

- Basic Information

  - Origin
  - Paths
  - Flags
- Extended static information
  - Number of visible waypoints
  - Average distance to visible waypoints
  - Bitfield for fast visibility lookup
  - some other terrain analysis data
- Statistical information
  - Number of kills
  - Time of kills
  - Damage taken (separated into directions)
  - Damage done (separated into directions)
  - Traffic
  - Connections where a bot got stuck
  - ...

A special feature to mention here is the visibility table. This table is realized by using an array of bitfields. We use such a table, because tracing a line throught the BSP takes a lot of time. Just reading this boolean value from a table is a lot faster and allows you to even use it inside your pathfinding process or doing boolean operations on them. There are only few occassions where you cannot use them, mainly if you don't want to trace a line from the origin of one waypoints to another. This is needed when doing visibility checks for the bot's perception. Another problematic point are moving objects, like doors or breakable entities. The problem was now, that a square visibility table is not easy to extend, because all those bitfields would have to be extended by one bit if a new waypoint is added or we would have to allocate some more memory in advance to avoid this. The solution is simple: We assume if waypoint 1 is visible from waypoint 2, waypoint 2 should be visible from waypoint 1. Based on this assumption we can build a triangular visibility table. For the first waypoint, the bitfield is one bit long, just containing the visibility to itself which is always 1, the second waypoint contains a bit indicating the visibility to the first waypoint and to itself, and so on. If you want to check visibility here between two waypoints, you first have to determine

which index is bigger and then take a look at the corresponding location in your table. This way you save some memory, which might be more cache friendly, but it also complicates some actions : If you want for example to create a bitfield containing the visibility of a given waypoint to another, you can first copy the bitfield of one waypoint, but then you have to go 'down' throught the visibility table, accessing a lot of different bitfields.
To speed up a very frequently called function, a function to get the index of the nearest waypoint to some position, a hashtable was introduced. The 'traditional' way is to loop throught all waypoints, calculate all distances, and return the index of the waypoints with the minimal distance to the position given. Calculating a distance involves 3+2 floating point additions and 3 multiplications, (Calculating the square root is not needed.) and of course loading all that data from RAM to your CPU for each waypoint.
A much faster way is to use a hashtable. The hash function only looks at the x and y component of the waypoint's origin, dividing the whole range from -4096 to 4096 units for each of them into 64 equally sized squares. Of course normal maps do not use this whole area, and one could think about using another hash function, e.g. based on the statistic distribution of all waypoints or determing the exact boundaries of the map to optimize the hash function. I havn't done this yet. To find the nearest waypoint now, you only have to find the bucket which belongs to the given position. Now we only have to check the distances of all waypoints in that very bucket and all neighbouring buckets. If we cannot find any waypoint, we can still call the traditional search function. Often we need to find a waypoints near to some player. In this situation the latter case is highly unprobable, because the waypoint recording part will add a new waypoints as soon as the next waypoint to the player is too far away.
During the game we collect data to know where are dangerous places on the map, where we can expect high traffic, where we can do a lot of damage. Those information are stored with each waypoint. If a bot has low health and doesn't want to be seen, the pathfinding can take this data into account. Or it might be used if a bot searches a spot to snipe : it should be relatively save, overlooking a lot of waypoints at a higher distance and maybe even having already a high damage value which would mean, there were already some players having taken advantage of this position.

(TA - data ?!)

4.4 Waypoint Recording

To create the waypoint mesh, we first add waypoints on position defined by certain entities in the game (see *'Interesting Entities'* in 'Perception'). Doing this, we already can waypoint the start points, some important places like health chargers or ladders. Unfortunately this is not sufficient at all. The other waypoints are created by observing the players inside a game, humans and bots as well. If there are not enough waypoints to navigate, the bot searches a place to run to. This place is found by checking if it is visible to the bot, if it is reachable and if there isn't already a waypoint near to it. By going there, the bot creates waypoints.

How it is done : Ten times per second the so called *CObserver* class analysis the movements of the players and creates waypoints if neccessary. First some basic informations are calculated and stored, namely we get the nearest waypoint to each player, calculate the current distance and the position is stored each time for later processing. Then the main functions are called. First we check the distance to the next waypoint. If this distance is above some threshold, a waypoint has to be added. Also if the line of sight to the last waypoint to which the player was near to is interrupted, we check if we can add a waypoint near to that 'corner'. Each time we want to add a waypoint, we first have to check if the direct connection to the last visited waypoint is valid. For this we have a 'reachable' function, which tests if there are obstacles or gaps in between. If this function tells us that the new origin is 'reachable' from the old one, everything is ok and a new waypoint can be created and connected. If this is not the case, we have to think about other possibilities. Since we know that the player we are observing actually got where he is now, there has to be some connection in between, i.e. it has to be somehow 'reachable'. Here the recorded positions can be used : We take an origin in between the waypoint we want to add and the last visited waypoint from those saved positions and check again if those two connections are being evaluated 'reachable'. If this is the case, we create two waypoints and connect them with paths.
Other player movement recording bots like ***parabot***'s additionally record how a player pressed a button and they check the environment for more possibilities to place waypoint, beside the actual path of the bot. So there are several possibilities to extend this system, but even this simple recording system creates a playable waypoint mesh after about 5 minutes when 8 bots are playing alone. When there are humans playing, it is faster, although ***JoeBOT XP*** still has the problem that in team based mods, the waypoints on the side of the human's team are a lot smoother.

4.5 Pathfinding

In a first person shooter, a bot have two main tasks to fulfill : Shoot appearing enemies and move around. The first thing requires only little intellegence, here the important facts are reaction time, weapon, other teammates and your position on the map. The other task, running around - in combat or just exploring the map - is a more crucial point. If a bot always gets stuck somewhere, the user might say : "oh what a dumb bot" and removes the game from the harddisk. We get the same situation if all bots are always running along the same paths. Therefore pathfinding is very important.
To find a path between two nodes of the waypoint graph, there are several algorithms. Using brute force here is not possible unless you only have less than 200 nodes with only little interconnectivity. ***JoeBOT*** *used the Floyd Warshall algorithm. It calculates a table containing all shortest paths from each waypoint to another before the game starts. When the bot needs then a path, it can be calculated really fast : You only need to look up the next waypoint for your current nearest waypoint and your goal. So if you get off your initially wanted path, continueing to your target does not need additional calculating. If you have only a few targets like in Counterstrike, you need to randomize your paths : You can do this for example by picking up to 50 nodes from the middle of the wanted path, calculating some values for those new paths, like how dangerous a path is, how long the path is at all, if the hostages are able to follow and some random values. The bot then decides which path to take based on its constitution - the bot can be aggressive, normal or defensive. By searching paths this way, they can be randomized almost sufficiently, but defining the exact costs of the paths is problematic. Another problem is that we cannot add waypoints iteratively to the floyd warshall table, so that recording waypoints on the fly would be difficult and we would have at least lags in between two rounds for recalculating.* Therefore I decided to use A*[1] in ***JoeBOT XP*** .

4.5.1 Implementation of A*

A* is an algorithm which allows you to find the 'shortest' path between nodes in a mesh. The cost for getting from one node to another is not fixed, so you can define your own cost functions, considering cover, danger etc. of any spot.

The 'best' node from the open list is the node with the smallest sum of heuristic and cost.

[1] pronounced A - Star

Pseudo code of A*:

- Put your start node to the open list
- Loop while the open list is not empty
  - Get the best node from the open list
  - If this node is our goal, then we have our path
  - Loop through all connected nodes
    - If already in the open list
      - Add the node on the open list to the children of the best node and update heuristic and cost if necessary
      - continue looping throught the other connected nodes
    - If already in the closed list
      - Update heuristic and cost if necessary, move that entry from the closed to the open list
      - continue looping throught the other connected nodes
    - If none of this is the case, just add the node to the open list with accordingly set cost and heuristic
  - Add this best node to the closed list

Cost - Function [g(x)]: The cost function determines the 'cost' of getting from one node to another. The cost can be for example the euclidian distance being proportional to the travel time between the nodes. If you want to get a path which is less visible than others, you may multiply the distance with an apprioate factor if a connection is visible by a lot of other nodes.

Heuristic [h(x)]: Heuristic means 'educated guess'. This function is used to guess the minimum distance to the goal. A good guess in a first person shooter for the remaining cost is often the so called Manhattan distance $||.||_1$.
By using an appropriate heuristic function, the number of examined nodes while searching a path can be minimized. If the heuristic is always $h(x) = 0$, A* behaves like Dijkstra's Algorithm, i.e. all neighboring nodes around the current node are 'equal', no matter if they are near or far to the goal. Dijkstra's Algorithm is guaranteed to find the shortest path, but may not be optimal regarding runtime.
A* is still guaranteed to find the shorted path if h(x) is always smaller than the actual cost

of the path to find. If the heuristic exactly matches the cost of the path, A* will find it and be very fast. Unfortunately it is often impossible to know the exact cost.
If the heuristic is bigger than the actual cost of the path, the shortest path cannot always be found. In one extreme, if the heuristic is significantly bigger than the cost, A* performs like the Best-First-Search algorithm. This algorithm only regards the distance for example to the goal node, therefore it is very fast on maps without obstacles. On maps with obstacles, the actual cost of the path to the node we are just examining can play in important role as well. A* merges the Dijkstra's and the BFS approach to profit from both's advantages.

The A* part of ***JoeBOT XP*** , called *AStarMachine*, is a template class. The template parameter is another class providing the cost and heuristic function, thus enabling the user to use whatever function wanted, also being able to code more than just a cost and heuristic function there, but also functions being called from the constructor or before starting the pathfinding process to adjust some variables.

4.5.2 Speeding up A*

When implementing A* , we have to decide what types of data structures to use for the different lists. A common operation on both list is to check if a node is already an element of it, therefore it is useful to use some sort of hashtable in this case, to get $O(1)$ operation time here. Another possibility is to create an array and set flags according to the location the node is in.
The other common operation on the open list is getting the 'best' node. Searching for the best node each loop takes $O(N)$ when the code just loops throught all entries. To avoid this, one can use priority queues, allowing to get the best node in $O(lgN)$, or priority queues on top of an unsorted rest, so called 'hot queues', which can be fast as well, because some nodes will be only checked once, and so they stay out of the sorted part of the hot queue which has to be searched.
When using A*-nodes which are added to the open and closed list, you can speed your implementation up by using a memory pool. With a memory pool, you allocate quite a number of wanted instances ahead, and then it is faster to allocate or free a node using the memory pool instead of using *new* and *delete*.

JoeBOT XP 's A* machine looks like this :

```
template <class tAStarGoal>
class AStarMachine : public AStarBase
{
public:
    AStarMachine(){
    }
    virtual ~AStarMachine(){
    }

    virtual void setStart(int iWP){
        m_Goal.setStart(iWP);
    }
    virtual void setDestination(int iWP){
        m_Goal.setDestination(iWP);
    }
    virtual int getStart(void){
        return m_Goal.getStart();
    }
    virtual int getDestination(void){
        return m_Goal.getDestination();
    }

    virtual void linkChild(AStarNode *pNode,CPath_iterator &path_iter){
        // we have a new node: So we gotta check if it's on the open or on the closed list
        // If none of both, we gotta put it on the open list.
        // in all cases we have to set the pointers correctly, so we'll get a nice tree here :)

        int iChildWP = path_iter.getTo();

        int g = pNode->m_g +
            m_Goal.getCost(pNode->m_iWaypoint,path_iter);        // cost for this child

        int iCheck;        // if it is already in some list, the index is stored here

        assert(iChildWP > -1 && iChildWP < _MAX_WAYPOINTS);

        if(iCheck = m_PQOpen.getIndex(iChildWP)){
            assert(iCheck <= m_PQOpen.m_ln && iCheck >= 0);
            // it's already on the open list ...
            AStarNode *pCheck = m_PQOpen.getItem(iCheck)->m_p;

            pNode->m_Children.addChild(pCheck);
            if(g < pCheck->m_g){
                pCheck->m_pParent = pNode;
                pCheck->m_g = g;
                pCheck->m_f = g + pCheck->m_h;

                m_PQOpen.updated(iCheck);
            }
            return;
        }
        else if(iCheck = m_PQClosed.getIndex(iChildWP)){
            assert(iCheck <= m_PQClosed.m_ln && iCheck >= 0);
            // it's already on the closed list ...
            AStarNode *pCheck = m_PQClosed.getItem(iCheck)->m_p;

            //pNode->m_Children.addChild(pCheck);
            if(g < pCheck->m_g){
                pCheck->m_pParent = pNode;
                pCheck->m_g = g;
                pCheck->m_f = g + pCheck->m_h;

                //updateParents(pCheck);
                m_PQClosed.removeUnsorted(iCheck);
                //return;
            }
            else
                return;
        }

        // it's neither on the open nor on the closed list ...
        AStarNode *pNewChild;
```

```
    pNewChild = newNode(iChildWP);

    pNewChild->m_pParent = pNode;
    pNewChild->m_g = g;
    pNewChild->m_h = m_Goal.getHeuristic(pNewChild->m_iWaypoint);
    pNewChild->m_f = pNewChild->m_h + g;

    AStarNodeContainer PnC;
    PnC.m_p = pNewChild;

    pNode->m_Children.addChild(pNewChild);
    m_PQOpen.insert(PnC);
}

virtual bool runAStar(void){
    ///PROFILE("runAStar();");
    // main function of all this stuff :)
    // - main loop
    // - handle slices
    // - getting best nodes
    // - linking nodes
    // - getting new open nodes

    int iConnectedWP,
        iCurrentWP;

    AStarNode *P,*B;
    AStarNodeContainer PC;
    CPath_iterator path_iter;

    if(!m_Goal.isValid())
        return false;

    if(m_bFinished){            // if the last job was finished, this is a new job
        resetNodes();

        m_iCSlice = 0;

        m_iBreakReason = BR_NONE;            // no break reason by default

        m_lRevolution = 0;
        m_lCSlice2 = m_lSliceSize;

        P = newNode(m_Goal.getStart());

        P->m_g = /*m_Goal.getCost(m_Goal.getStart(),m_Goal.getStart())*/0;
        P->m_h = m_Goal.getHeuristic(m_Goal.getStart());
        P->m_f = P->m_g + P->m_h;

        PC.m_p = P;            // set container pointer

        m_PQOpen.insert(PC);
    }
    else{            // job isnt finished yet, so just prepare a new slice and go on
        m_iCSlice ++;
        m_lCSlice2 += m_lSliceSize;
    }

    while(true){
        //PROFILE("runAStar-loop");
        // check if this is enough
        if(m_lRevolution > m_lMaxRevolutions){    // is the maximum revolution count reached ?
            m_iBreakReason = BR_MAXREV;
            m_bFinished = true;
            logResult();
            return false;
        }

        if(m_lRevolution > m_lCSlice2){            // a slice is full
            m_iBreakReason = BR_MAXSLICE;

            m_bFinished = false;
```

```
            return true;
        }

        if(m_PQOpen.empty()){        // no more open nodes - no path has been found
            m_bFinished = true;
            logResult();
            return false;
        }
        PC = m_PQOpen.getTop();
        B = PC.m_p;
        if(m_Goal.isDestination(B)){        // a path has been found
            m_pDestinationNode = B;
            m_bFinished = true;
            m_iBreakReason = BR_NONE;        // it's not a break :D
            logResult();
            return true;
        }
        iCurrentWP = B->m_iWaypoint;

        path_iter = g_Map.m_Waypoints[iCurrentWP];
        while(1){
            iConnectedWP = path_iter.getTo();

            if(iConnectedWP != -1){
                if(m_Goal.isOpen(iCurrentWP,path_iter)){
                    linkChild(B,path_iter);
                }
                ++path_iter;
            }
            else
                break;
        }

        m_PQClosed.insertUnsorted(PC);

        m_lRevolution ++;
    }
  }
  // data

  tAStarGoal m_Goal;
};
```

Searching itself can be optimized, too . A* will spend a lot of time searching for a path, if such a path does not exist. If the mesh you are navigating on is static, you can create a table which stores if 2 waypoints are connected to each other. Another possibility is to think about certain break conditions for pathfinding, although this might sometimes prevent the pathfinder from finding the right path, so one has to be very careful here.
In some situations, mainly when pathfinding in a rectangular grid, you can observe that there are nodes in your open list with the same sum of heuristic and cost. In this case it is adviseable to take other properties into account, for example the smoothness of a path using the dot product, to make A* search a smaller space.
If you do pathfinding on a 'well structured' map, i.e. if you have some specific points on your map which are visited often, like weapon spawns in HL-Deathmatch, you can use different levels of pathfinding : To test if there may be a path, a low detail grid, and for the actual navigation a grid with a higher density. This is often called *hierarchial A**.

If searching a path takes a lot of time, but you want the unit to move quickly, it is often useful to go directly into the direction of the goal. When the path is calculated, the unit will move that right path then. This creates the illusion for the user, that the unit is responding immediately without appearing stupid.

4.5.3 *JoeBOT XP* 's A* Machine

The A* machine of ***JoeBOT XP*** should fulfill the following points

- Find paths
- Be Fast
- Flexible handling of cost and heuristic
- Sliceable pathfinding process

"Finding paths", that's obviously the main reason for all this. It has to be fast, because the game the bot is acting in is realtime, lags or long periods to 'think' about anything are simply not wanted.
Like already said above, the next point, the flexible handling of cost and heuristic, is implemented with a template class in ***JoeBOT XP*** . Those 'plugins' to the A* machine can now make the A* machine just find the shortest path, or it can be used to estimate a possible position of an enemy.
Making the pathfinding process 'sliceable' has something to do with being fast. Since our environment is realtime and we only have very little time each frame and then we have to hand over control back to the engine, we cannot wait until a long calculation is done. If there is e.g. no possible path, A* might take really long to find this out. Another problematic situation is when multible bots are requesting paths at the same time. We can avoid lags in those situations by slicing up the pathfinding to several frames, only calculating for example 50 iterations per frame. To do this, ***JoeBOT XP*** has an *AStarTaskManager*, which gets pointer to A* machines which have to be run, and runs them subsequently each frame. If wanted, a callback function can be also provided, so that the *AStarTaskManager* can notify the 'owner' of that A* machine that the pathfinding process is finished.

```
class AStarTask{
    friend class AStarTaskManager;
```

```
public:
    AStarTask();
    AStarTask(AStarBase *p,int iPriority,CBaseBot *pBot,CCallback_onPathCreation *pPC=0,long lTime=0)
    :m_pAStar(p),m_iPriority(iPriority),m_pBot(pBot),m_pCB_onPathCreation(pPC),m_lExecution(lTime),
    m_lExecuted(0){}
    virtual ~AStarTask();
protected:
    AStarBase *m_pAStar;
    CCallback_onPathCreation *m_pCB_onPathCreation;
    CBaseBot *m_pBot;
    int m_iPriority;
    long m_lExecution;            // last execution cycle

    long m_lExecuted;            // how many slices are already executed ?
};

class AStarTaskManager  {
public:
    AStarTaskManager();
    virtual ~AStarTaskManager();
    void addTask(AStarBase *,int,CBaseBot *,CCallback_onPathCreation *pPC=0);
    void run(void);
    int deleteTasksFrom(CBaseBot *);

protected:
    int m_iSlicespF;                // Slices per frame
    long m_lCounter;
    long m_lCTasksTotal;
    int m_iSlicesNow;
    list <AStarTask> m_LTasks;
};
```

4.5.4 Using A* in the Bot

A behaviour using A* is *CBV_HLDM_HideReload*, which is active when a bots needs to reload. (See *'Behaviours'* for further information of how they work.)

```
// Hide and reload then

void CBV_HLDM_HideReload::evaluate(list<CGoal> *pGoals,CPerceipt* p){
    if(p->m_lType != CPerceipt::PT_ACTION)
        return;

    if(!(p->m_lTypeSpec & CPerceipt::PTX_ACTION_ENEMY))
        return;

    if(g_pGame->getTime() - m_fLastExecution < .5f) // dont check so often
        return;

    if(g_pGame->getTime() - p->m_fLUpdate > .05f) // it is a current perception
        return;

    int iPriority;

    if(float(m_pBot->m_pWeapons->m_iCurrentWClip) /
        float(m_pBot->m_pWeapons->getCWeapon()->m_iClipSize) < .15f
        || m_pBot->m_pWeapons->m_iCurrentWClip <= 2){
        iPriority = 50;

        pGoals->push_front(CGoal(iPriority
            ,CGoal::GT_MOVE,this,p));
    }
```

```
    return;
}

void CBV_HLDM_HideReload::initialize(CGoal *pGoal){
    m_fLastExecution = g_pGame->getTime();

    int iENow = pGoal->m_pPerceipt->m_pPlayer->m_iNearestWP;

    AStarMachine<AStarGoalHide> *pAStar = new AStarMachine<AStarGoalHide>;

    pAStar->setStart(m_pBot->m_pPlayer->m_iNearestWP);
    pAStar->m_Goal.hideFrom(iENow,pGoal->m_pPerceipt->m_VOrigin,m_pBot->m_pPlayer->getOrigin());
    pAStar->m_Goal.setMinimumCost(300); // travel at least 300 units

    m_pBot->m_pMovePath->m_fLastVisitedWP = g_pGame->getTime();

    g_AStarTasks.addTask(pAStar,5,m_pBot,this);        // add task, calling back on task creation

    m_iExecuteFSMS ++;        // increment state
}

void CBV_HLDM_HideReload::onPathCreation(AStarBase *p){
    m_iExecuteFSMS ++;

    m_pBot->m_pMovePath->getResult(p,this); // we want to be called back on reached destination

    delete p;
}

void CBV_HLDM_HideReload::onReachedDestination(CPlayer *pPlayer){
    if(float(m_pBot->m_pWeapons->m_iCurrentWClip) /
        float(m_pBot->m_pWeapons->getCWeapon()->m_iClipSize) < .15f        // still right ?!
        || m_pBot->m_pWeapons->m_iCurrentWClip <= 2){
        m_pBot->m_pAction->reload();
        if(!m_pBot->m_pEMEnemy->m_iVisCount) // no enemies in sight
            m_pBot->m_pAction->slowDown0(2.f,0.f);
    }
}
```

The *AStarGoalHide* A* machine plugin is defined as follows :

```
void AStarGoalHide::hideFrom(int iWP,const Vector&VHideFrom,const Vector&VMe){
    m_iHidefromWP = iWP;
    m_VHideAim = VMe + (VMe - VHideFrom).normalize() * 5000.f;
}

int AStarGoalHide::getHeuristic(int i1){
    float fHeuristic,fDistance;

    fDistance = (g_Map.m_Waypoints[i1].m_VOrigin-m_VHideAim).CBLength();

    fHeuristic = fDistance;

    return (int)fHeuristic;
}

void AStarGoalHide::setMinimumCost(int iMinCost){
    m_iMinCost = iMinCost;
}

bool AStarGoalHide::isDestination(const AStarNode *pASN){
    // return true for a waypoint which isnt visible from m_iHidefromWP and which is far enough
    if(pASN->m_g >= m_iMinCost)
        return (!g_Map.m_Waypoints.getVisible(m_iHidefromWP,pASN->m_iWaypoint));
    return false;
```

```
}

bool AStarGoalHide::isValid(void){        // returns true is this instance is ready to be used
    if(getStart() == -1)
        return false;
    if(m_iHidefromWP == -1)
        return false;
    return true;
}
```

4.6 Perception

Another important part of the bot is the perception. Somehow the bot has to get the information where it can see an enemy, where a weapon is spawned, if it has been hit, if it heared footsteps etc.

JoeBOT *had several functions to fulfil these tasks. One function e.g. looped throught all player, determined which where visible and then decided on which enemy to shoot. This information was then just stored in a single variable and used, if necessary, later by another function. It was the same for sounds, entities, etc..* But all those perceipts had some properties in common, therefore I decided to unify all this perception parts into one bigger system. Doing this, it is simpler to extend this system, to modify it, because everything is in one place, you don't need to change several functions and variables all over your code to add a long term memory e.g.. Memorizing events has become a part of the perception of ***JoeBOT XP*** . All perceipts remain in the so called short term memory, which represents all present perceipts, for a fixed amount of time, which is about 3 seconds. Some of them are deleted after this time, others are added to the long term memory.

When a bot is executing the perception code, it checks if there are entities to percept, if there are visible players - teammates or enemies - or if something can be heard. If the bot has a perception, it is put into a list of perceipts. (Perceptions caused by network messages are added directly when the bot gets the message.) This list is then compared with the current list of perceipts: New perceipts are added, old, but remaining ones are updated so that the bot can see that they are still percepted. Now the list of new perceipts is empty. We only need to remove too old perceipts from our current list and eventually add some entries to the long term memory.

4.6.1 'Interesting Entities'

One of the goals of ***JoeBOT XP*** was being engine/game/mod independent as far as possible. Therefore we need to have some generic structure storing information about a game or a mod, so that we don't have to care about differences between mods in our perception process. So we need to define the important entities in a mod once and then the bot is able to use them, to 'know' what it can do with it. Additonally this data is used in ***JoeBOT XP*** to create the first waypoints when a new map is loaded which hasn't any waypoints yet.

```
class CIEntity  {
public:
    CIEntity();
    CIEntity(const char *,bool);
    virtual ~CIEntity();

    bool compare(CEntity&);
    void assign(CEntity&,CPerceipt *);

    virtual bool isValid(CEntity&); // looks if this entity is valid - for weapons e.g.
        \\look if there's an owner ... if there is one, someone carries it,
        \\it's uninteresting

    virtual Vector getWPOrigin(CEntity&); // get the location ... may not be pev->v.origin for some entities

    virtual long addWP(CEntity&); // add a waypoint at this entity

    char m_szClassname[30];   // classname
    int m_iType;              // type of entity
    int m_iWeaponId;          // weapon ID ( for weapon entities )
    int m_iAmmoId;            // ammo id ( for ammo entities )
    int m_iContaining;        // health, armor, whatever
    bool m_bStatic;           // if true, you are allowed to place a waypoint there
    long m_lWPFlag;           // waypoint flag to set -> for ZONE Type IEntities
    int m_iId;                // all interesting entities are numbered

    // define types of interesting entities ...
    enum types{
        NONE = 0,
        WEAPON,
        WEAPONWBX,
        AMMO,
        CHARGER,
        ITEM,
        LADDER,
        ZONE
    };
    enum containing{ // does an entity contain something of the following ?
        HEALTH = 1,
        ARMOR
    };
};
```

The game class contains a list with pointers to interesting entities. We cannot just add only instances of *CIEntity* there, because there might be differences in certain functions. A zone in a game like Counterstrike for example is not drawn, therefore the EF_NODRAW flag is set. On the other side, we want to ignore weapons which are invisible. Another example is the *addWP* function, which has to be obviously different for ladders and weapons. To do

this, we add pointer to instances of class which are derived from CIEntity. For Counterstrike it looks like this :

```
m_LIEntities.push_back(pIEntity = new CIEntity("info_player_start",true));        // ct spawn
m_LIEntities.push_back(pIEntity = new CIEntity("info_player_deathmatch",true)); // te spawn

m_LIEntities.push_back(pIEntity = new CIEntity("hostage_entity",true));  // hostage
m_iIEntityIdHostage = pIEntity->m_iId;

m_LIEntities.push_back(pIEntity = new CIEntity("grenade",false));      // bomb, grenade
m_iIEntityIdGrenade = pIEntity->m_iId;

m_LIEntities.push_back(pIEntity = new CInterestingLadder("func_ladder"));

m_LIEntities.push_back(pIEntity = new CInterestingWeaponboxWeapon("weapon_scout",CS_WEAPON_SCOUT,false));
m_LIEntities.push_back(pIEntity = new CInterestingWeaponboxWeapon("weapon_xm1014",CS_WEAPON_XM1014,false));
m_LIEntities.push_back(pIEntity = new CInterestingWeaponboxWeapon("weapon_c4",CS_WEAPON_C4,false));
m_LIEntities.push_back(pIEntity = new CInterestingWeaponboxWeapon("weapon_mac10",CS_WEAPON_MAC10,false));
m_LIEntities.push_back(pIEntity = new CInterestingWeaponboxWeapon("weapon_aug",CS_WEAPON_AUG,false));
m_LIEntities.push_back(pIEntity = new CInterestingWeaponboxWeapon("weapon_ump45",CS_WEAPON_UMP45,false));
m_LIEntities.push_back(pIEntity = new CInterestingWeaponboxWeapon("weapon_sg550",CS_WEAPON_SG550,false));
m_LIEntities.push_back(pIEntity = new CInterestingWeaponboxWeapon("weapon_awp",CS_WEAPON_AWP,false));
m_LIEntities.push_back(pIEntity = new CInterestingWeaponboxWeapon("weapon_mp5navy",CS_WEAPON_MP5NAVY,false));
m_LIEntities.push_back(pIEntity = new CInterestingWeaponboxWeapon("weapon_m249",CS_WEAPON_M249,false));
m_LIEntities.push_back(pIEntity = new CInterestingWeaponboxWeapon("weapon_m3",CS_WEAPON_M3,false));
m_LIEntities.push_back(pIEntity = new CInterestingWeaponboxWeapon("weapon_m4a1",CS_WEAPON_M4A1,false));
m_LIEntities.push_back(pIEntity = new CInterestingWeaponboxWeapon("weapon_tmp",CS_WEAPON_TMP,false));
m_LIEntities.push_back(pIEntity = new CInterestingWeaponboxWeapon("weapon_g3sg1",CS_WEAPON_G3SG1,false));
m_LIEntities.push_back(pIEntity = new CInterestingWeaponboxWeapon("weapon_sg552",CS_WEAPON_SG552,false));
m_LIEntities.push_back(pIEntity = new CInterestingWeaponboxWeapon("weapon_ak47",CS_WEAPON_AK47,false));
m_LIEntities.push_back(pIEntity = new CInterestingWeaponboxWeapon("weapon_p90",CS_WEAPON_P90,false));

m_LIEntities.push_back(pIEntity = new CInterestingZone("func_hostage_rescue",CWaypoint::FLAG_GOAL));
m_LIEntities.push_back(pIEntity = new CInterestingZone("func_bomb_target",CWaypoint::FLAG));
```

We defined the spawnpoints only for initial waypoint creation, then we defined hostages and bombs and stored their respective *CIEntity ID* to be able to check very fast if it's a hostage or a bomb without having to compare strings. We create a ladder *CIEntity* so that ladders are automatically waypointed if a map is started the first time. Adding the entries for the weapons, we additionally tell which id each of those entities would correspond with. The 'false' parameter is being stored in the m_bStatic member variable of *CIEntity*, to show, if something is static or not. If it is, we can add a waypoint there, if it is not, it makes no sense to add a waypoint there. The last things we add are the bomb target zones. Here we use a special *CInterestingZone* class, because we don't bother if the EF_NODRAW flag is set or not when checking all entities, like already stated above. Therefore *CInterestingZone::isValid(CEntity &Entity)* always returns true.

The standard *CIEntity* functions look like this :

```
Vector CIEntity::getWPOrigin(CEntity &Entity){
    return Entity.getOrigin();
}
```

```
const Vector &CIEntity::getOrigin(CEntity &Entity){
    return Entity.getOrigin();
}

long CIEntity::addWP(CEntity &Entity){
    Vector VOrigin = getWPOrigin(Entity);

    UTIL_TraceToGround(VOrigin);

    VOrigin = VOrigin + Vector(0,0,38);

    if(g_Map.m_Waypoints.getNearest(VOrigin,false,false,0,50) == -1)
        g_Map.m_Waypoints.add(VOrigin);

    return 1;
}

bool CIEntity::compare(CEntity &Entity){
    ////////////////////////////////////////////////////////
    // CPerceipt *CIEntity::compare(CEntity &Entity){
    ////////////////////////////////////////////////////////
    // check if that entity matches this IEntity
    ////////////////////////////////////////////////////////

    if(!isValid(Entity))
        return false;

    if(strcmp(m_szClassname,Entity.getClassName()))
        return false;

    return true;
}

void CIEntity::assign(CEntity &Entity, CPerceipt *pNewPerceipt){
    // assigns perception data to perceipt.
    [....]
}

bool CIEntity::isValid(CEntity &Entity){
    if (Entity.getEntity()->v.effects & EF_NODRAW)
        return false;
    return true;
}
```

Example: The *CInterestingWeaponboxWeapon* class: we need to change some functions, because we have other conditions here to check if the entity is really a dropped weapon and the real origin of the dropped weapon is not the standard origin, but the origin of its owner. This is just a speciality of the HL/Counterstrike code.

```
bool CInterestingWeaponboxWeapon::isValid(CEntity &Entity){
    if(!Entity.getOwner()) // no owner ... so what's that ? at least no weaponbox
        return false;
    if(strcmp(Entity.getOwner().getClassName(),"weaponbox")) // it's not in a weaponbox
        return false;

    return true;
}

const Vector &CInterestingWeaponboxWeapon::getOrigin(CEntity &Entity){
    if(!Entity.getOwner())
        return g_VZero;                // return zero if we got a problem here

    return Entity.getOwner().getOrigin();
}
```

4.6.2 Percepting Entities

Percepting entities is now quite easy :

```
while(Entity = UTIL_FindEntityInSphere(Entity,VOrigin,_ENT_PERC_RADIUS)){ // loop thru entities around here
    iter_ientities = begin_ientities;
    while(iter_ientities != end_ientities){
        // check this entity
        if( (*iter_ientities)->compare(Entity) ){
            if(m_pBot->m_pPlayer->views((*iter_ientities)->getOrigin(Entity))){
                pAllocPerc = new CPerceipt; // create a new perceipt

                (*iter_ientities)->assign(Entity,pAllocPerc);

                pAllocPerc->m_fDistance = (VOrigin-pAllocPerc->m_VOrigin).length();
                m_LNewPerceptions.push_back(pAllocPerc);
                break; // there arent double classname IEntities !? am I right ?
            }
        }
        iter_ientities ++;
    }
}
```

4.7 Behaviours

Now we have a waypoint system, a nice pathfinder and a perception system, but still no system which actually uses all this. We need to define how the bot should react on different perceipts, what it should do even without any perceipts.
JoeBOT XP has the following class :

```
class CBehaviour{
public:
    CBehaviour(CBaseBot *);
    virtual ~CBehaviour();

    virtual void reset(void);
    char *getName(void){return m_szName;}

    virtual void evaluate(list<CGoal> *,CPerceipt* p=0);
    virtual void initialize(CGoal *);
    virtual void execute(CGoal *);
    virtual void deinitialize(CGoal *);

    void percept(int i){m_iPercept = i;}

    enum PerceptionTypes{
        P_NONE = 0,
        P_STMem = (1<<0),
        P_LTMem = (1<<1),
        P_Command = (1<<2)
    };
    int m_iPercept; // what type of perceipts to percept :)
protected:
    char m_szName[80];
    CBaseBot *m_pBot;

    int m_iExecuteFSMS; // State of execute FSM
    int m_iEvaluateFSMS; // State of evaluate FSM
```

```
    float m_fLastExecution; // time of last execution
};
```

This class is the base class for all 'Behaviours' of ***JoeBOT XP*** . Each time the bot 'thinks', all behaviours which are stored in a list in each bot, are called. First the evaluate function is called: This function gets a pointer to a perception from the perceipt list, if the *m_iPercept* variable is set accordingly. Using this variable, some of the perceipts don't even get to the behaviour, which can save a lot of processing time. If the behaviour wants to do something, it produces a goal, an instance of the *CGoal* class. This class contains a pointer to the behaviour, a pointer to the related perception, the priority and the type of action. All those goals are collected and when all evaluation functions have been called, the goalfinder decides which of the goals to execute. We have three types of goals, which can be executed in parallel: 'Look', 'Move' and 'Action' -goals. For all behaviours to be executed, the *execute* function is called.
Sometimes it is useful to know when a function is called the first time and when it isn't called any more. For this purpose we have the initialize and deinitialize functions.

An example of a behaviour:

```
class CBVLT_HLDM_AttackSnipe:public CBehaviour,CCallback_onPathCreation,CCallback_onReachedDestination{
public:
    CBVLT_HLDM_AttackSnipe(CBaseBot *pBot):CBehaviour(pBot){
        strcpy(m_szName,"CBVLT_HLDM_AttackSnipe");
        percept(CBehaviour::P_LTMem);}

    virtual void evaluate(list<CGoal> *,CPerceipt* p=0);
    virtual void execute(CGoal *);
    virtual void initialize(CGoal *);
    virtual void onPathCreation(AStarBase *);
    virtual void onReachedDestination(CPlayer *);
};
```

This behaviour was written to make the bot search for a sniper place, if it has a sniper weapon, if it knows from his long term memory that there was somewhere an enemy which is still alive, and camp there.
This class is derived from 3 different classes. First one is the normal *CBehaviour* class, second is a class used to provide a callback function for the *AStarTaskManager* once the pathfinding process is finished. The last class this behaviour is derived from is another class providing a callback function : this function is called once the bot has reached his destination.
In the constructor we can see that this behaviour should only be called with long term memory perceipts.
The *evaluate* function checks if the long term memory perceipts are suitable, i.e. if the enemy has already been attacked by this bot, if the enemy is still alive etc. Then a check is

done whether the bot has a sniper weapon at all. If all this is the case, a *CGoal* instance is created and added to the list of goals.
The *execute* function does nothing but updating the last time the behaviour was executed.
The *initialize* function searches a place to snipe from. This place should be not too far away from the bot's current position, not too near to the enemy's last known position and it should overlook quite a few waypoints at once. Searching for this waypoint is mainly done by using 'AND' operations on bitfields. If such a waypoint is found, an *AStarMachine* is created and added to the *AStarTaskManager*.
When the pathfinding is finished, the *onPathCreation* function is called. Here we just tell the bot's basic path movement behaviour which path to take.
When the bot has reached the end of this path, the *onReachedDestination* function is called and we can tell the bot to camp there for a while.
A variant of the *CBehaviour* is the so called *CEmotion* class. It is derived from *CBehaviour*, but produces no goals, therefore needs no *execute* function. But it got two other functions : *preprocess* and *postprocess*. Those two functions are called before and after looping throught all perceipts. These 'emotions' can be used to analyse certain perception - in the simplest case just to count the visible enemies.

4.8 Teamplay

Teamplay is an important aspect of a lot of games, and it becomes more and more important. Users expect NPC's to coordinate their attacks, stupid and selfish agents are not needed any more.
JoeBOT *had no real teamplay, but some basic squad behaviour was achieved by adding task to the taskplanner to follow one of their teammates. One of the main problems here is, that this system is useless in fights. Second, since the bots cannot really estimate where the player to follow wants to go, cohesion is difficult to get. Extrapolating the players speed was sucessful here, unfortunately this led to delays in the bots movement especially at the round start, when the bots had to 'sort themselves'. In general this system works fine and produces nice movements when you see it on an overview.*

4.8.1 Current Implementation

JoeBOT XP has a separated squad and team system. Each bot in a team player mod is a member of a team, and then also added as a member of a squad.

The bots of ***JoeBOT XP*** itself have only very limited capabilities of navigation, they can hide to reload, but not much more. Most navigational tasks are performed by the squad. The squad then creates perceipts for the perception system of each single bot to make them go to some location. The squad itself is organized just like a bot, it has a perception system, a goalfinder and behaviours. Perceptions are for example radio commands. The behaviours and the goalfinder work the same way like in the bot.
The team structure processes possible radio commands of team members, avoids too frequent radio messages in the mod and determines the sizes of the squads. It also contains an array of the time a waypoint has been last seen, so that the team's members don't have to look all the same direction when they run around a corner together. This approach is motivated by the fact that a team of humans also percept where the other are looking and try to have the whole terrain in sight as a team.
The destinations of the squad members to go to are stored inside the squad. A function checks if the bot is going there, and creates an A* Task to be executed, if needed. The bots do **not** get their destination just as an waypoint index, but the command already contains the whole path. Otherwise it would have been difficult to coordinate the pathfinding processes to avoid getting stuck : Normally all squad members get the command to go somewhere at the same moment. At this point the AStarTaskManager makes things worse, since the needed pathfinding processes might now run in parallel. The squad executes them now sequentially, thus knowing the paths of preceeding squadmembers. Those paths are now used to increase the cost of this waypoints, making the other bots avoid this route and take one which is not exactly the same. Doubling the cost of a waypoint on the path of another squad member produces resonable results. Using this system, 2 bots from the same squad will for example pass by a crate on different sides or take 2 different narrow paths if they are relatively close together.

Main behaviours of the squad :

- *Cohesion*
  This behaviour checks if all bots are running to the same spot, and then checks how far they are from the average distance to the goal. If a bot is significantly ahead, it gets the command to wait a bit, ideally at safe places. This function also takes the type of weapon the bot is carrying into account, since we do not want a bot with a sniper weapon to run in the first line.
  This behaviour looks pretty intelligent in gameplay and it increases the probability to be able to attack an enemy with multible teammates.

- *Avoidance*

  After introducing the penalty for running the same way in a squad, this behaviour has become almost useless. It detects if 2 bots are currently heading to the same waypoint, and then tells one of them to wait a bit. In narrow hallways and similar locations this behaviour is still very useful, though.

- *RunAround*

  The basic navigational behaviour : Just run around, pick a random waypoint and head there. This behaviour is active when no other tasks have to be fulfilled.