University of Amsterdam
Master Profile project Intelligent Systems

GOAP

Goal Oriented Action Planning

authors: **Gideon Maillette de Buy Wenniger** (coll. nr. 0234222)
Chemin des Falaises 3/723
1005 CH Lausanne

Attila Houtkooper (coll. nr. 0274542)
van Alphenstraat 3-1
1053 WE Amsterdam

date: 5 January 2008

Supervisor: **Frank Aldershoff**

Abstract

Game bots for shooter shooter games are researched. A new game bot infrastructure for the popular game Quake 3 Arena is developed based on the original code by J.P. van Waveren and the article on A* planning for F.E.A.R. by J. Orkin. Limitations and advantages of planning are investigated in comparison with other approaches such as Finite State Machines and Hybrid Deliberative/Reactive Architectures which combine planning with reactive behavior. The result is a working system, and a strong framework that can be extended to involve more complex actions and plans. Additionally this work thoroughly discusses how planning may be combined with reactive behavior, in order to take bot intelligence to the next level.

Index

1. Introduction

Creating autonomous robots is the dream of almost every Artificial Intelligence student. What makes this so interesting is that all important fields of AI come together in such projects. Good robots require planning, leaning, adequate computer vision, kinematics, reasoning about uncertainty   virtually all main fields of AI. Software robots for games, or shortly bots are like simpler versions of real world robots. Planning, learning, reasoning about uncertainty are still major issues, but some of the lower level problems such as kinematics and adequate computer vision are nonexistent or at least much simpler for game bots. Since the game designer has complete control over the game, things such as position determination and complex computer vision can be passed over by just using the information available in the game. At the same time, it is challenging to use the extra information available in a realistic way. It shouldn't be obvious that the bot uses information it couldn't reasonably have if it only used normal sensing and reasoning like the human players do
.
Games form an excellent testing environment for all kind of AI techniques, and game bots for shooter games are arguably the closest thing to autonomous robots in the real world. When the game bots in Q*uake3 Arena* [12] and later *Unreal Tourmanment* [8] were released they were undoubtedly revolutionary in their application of AI to games. Our project builds on the bot-AI of *Fear* [13] which uses A* Planning to determine the actions of the bots. Those bots deal with an easier situation than the tournament bots of *Quake 3 Arena* and *Unreal Tournament*. They are designed for the situation were there is only one opponent, namely the human player, and their movement is also relatively restricted by their placement in the game. In contrast, tournament bots typically have multiple opponents as well as possibly teammates, and their movement ranges of the entire game map. This requires much more complex behavior and planning than the single player frame of *Fear*.

In this project we implemented A* planning as used for *Fear* in *Quake3 Arena*. We wanted to research how the bots might benefit from this new approach, and if we could in some ways improve over the original *Quake 3 Arena* bots. As it turned out, their are many issues that make working with the *Quake3 Arena* code [4] pretty challenging. MingW [6] offers some relief when working with Windows, and a complete documentation [5] of the source was helpful as well.

This report is structured as follows. We begin by giving a short overview of AI in games so far. We then continue by discussing planning for games in the third section. We next give a thorough overview of Finite State Machine in the fourth chapter, comparing it to planning as well. In the fifth section we discuss different approaches to planning, focusing on the implementation with STRIPS and search algorithms. In the sixth section we cover different infrastructures for real world robot, and discuss how the insights reached in normal robotics might be quite relevant to game bots as well. Chapter seven discusses Game Engines and dynamic memory allocation. In chapter eight we discuss our GOAP implementation. Chapter nine covers our results. Finally chapter ten closes with discussion and conclusion.

2. Short overview of AI in Games

Computer game AI started taking of with real time strategy games such as *Command & Conquer.*
The AI of these games was structured using Finite State Machines. *Starcraft* is another good example of a very

popular real time strategy game based on the same concept. *Age of Empires* (1 and 2) show the development of this type of AI for an even much more complex game, with many more units, types of resources and possible strategies. Role Playing Games (RPGs) such as *Baldur's Gate* offered a new challenge, for they required a lot of diverse and preferably unpredictable behavior on side of the opponents. For those games however, still most of the AI was implemented by means of large collections of scripts rather than anything smarter. *Black and White* [7] was pioneering in it's application of *Reinforcement Learning* [16] to Games. *Quake 3 Arena* and *Unreal Tournament* brought at their time of release great improvements of Bot AI for tournament shooter games. Some other bots exists for the popular team play game *Counterstrike* which is build on the even more popular game *Halflife,* however they don't reach the level of sophistication of the Quake3 Arena bots. Particularly the navigation in Quake3 Arena that introduces *Brushes* and *Reachabilities* to perform path planning was a great improvement over the *waypoint* technology that was used in previous systems (and is sometimes still used for simpler bots). *Fear* pioneered in making planning the core of the bot AI.

This overview is short and incomplete. A more extensive overview can be found on a nice site specially dedicated to AI in computer games [15].

3. Planning for Games: Strengths and Weaknesses.

3.1 Dependency on a World Model

Planning has a long tradition in AI. In the 70's , people from Standford used the minimalistic STRIPS language to perform planning that steered the behavior of the famous robot *Shakey*.
The results of this approach were somewhat disappointing. At the time then computers were much slower, and basically the planning just took to much time, making the robot react only very slow and thus inadequate to the environment. Another, more fundamental problem of using planning in the real world came to light then as well. Planning is always based on a certain model, a more simple version of reality. It turns out however that the real world often does not exactly work in the way it has been modeled. Either the model is too simple, or it is just incomplete, and in most cases ignores the intrinsic uncertainty of our world. This is not to say the world isn't deterministic. It merely says that our world is at least so complex that we don't understand it well enough to treat it as a fully deterministic system,. Therefore much more adequate results can be achieved by making uncertainty a central part of our model of the world, and reasoning about uncertainty a central part of our application of the model in performing planning.

3.2 Closed World Assumption

The fact that planning is only feasible if we have a limited model of the world leads to several challenges. To make a limited, yet for the required task sufficient model of the world, system builders have most often opted for making the closed world assumption. This means assuming the world is fully deterministic, and the used model is complete for the task at hand, and therefore it's predictions are flawless and do not suffer from uncertainty. This of course, is a very optimistic as well as unrealistic view of the world, and has proven to be not adequate for most real world tasks. The alternative, as mentioned before is to make the modeling of uncertainty an integral part of the approach, and allow that the outcome of some actions is simply not known or at least not certain.

3.3 The Success of Planning in FEAR

The makers of *F.E.A.R.* made the step of implementing planning in a game. Their STRIPS implementation didn't go much beyond the original one as used in *Shakey*. One difference was that they used procedural preconditions and postconditions, so that appropriate functions could be used to implement them rather than handling everything on a pure symbolic level. Contrary to the results with *Shakey*, their approach was quite successful. It facilitated great game play experiences that really went beyond the AI of earlier games.
Several factors can explain this success. First of all they kept the planning at a high level, so that the actions remained quite abstract and the number of actions remained very limited.
Keeping the actions abstract means complete plans can remain abstract and short. This is computationally advantageous for it limits computation and memory usage. When planning is done using very primitive actions, the whole planning will become intractable or at least very slow. Besides being computationally attractive, an advantage of abstract actions is that they're less likely to fail. Primitive actions such as move forward two units fail easily, because if any small thing is different from what is expected perhaps there is some small obstacle in the way - this causes the action to have a result slightly different from what was planned, and consequently can make the entire plan fail. On the other hand, abstract actions such as move to area 103 are less likely to fail. They only fail if there really exists no sequence of primitive actions that can accomplish them, for the set of primitive actions executed will just be adapted in whatever way is needed to accomplish the abstract goal.

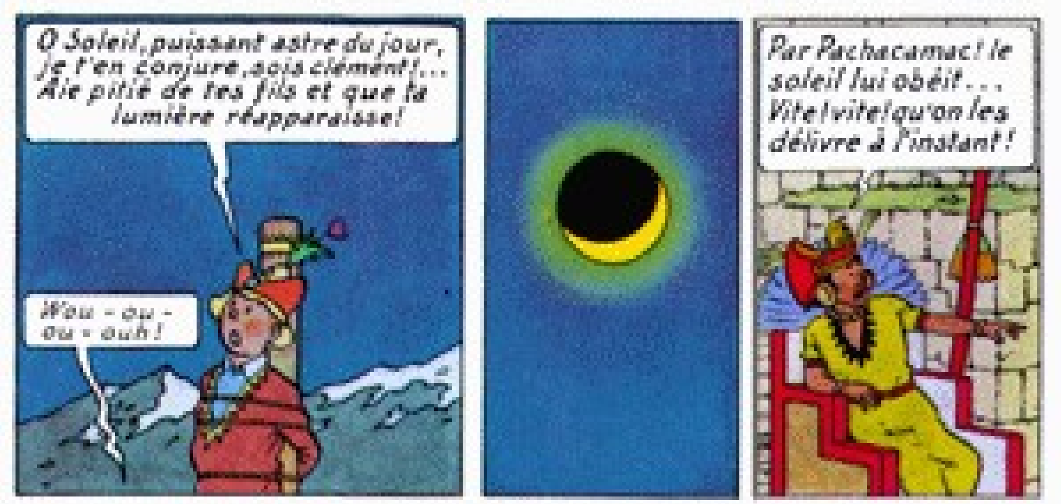

O Soleil, puissant astre du jour, je t'en conjure, sois clément !... Aie pitié de tes fils et que ta lumière réapparaisse !
Wou - ou - ou - ouh !
Par Pachacamac ! le soleil lui obéit . . . Vite ! vite ! qu'on les délivre à l'instant !

Figure 1: The power of unknown information as illustrated by Hergé's Tintin Prisoners of the Sun

Another big advantage any game designer has over a robot programmer is that while the real world is not easily manipulated, in a a game one has complete control. That means in a game one can simply make the environment behave as one predicts it to do. In other words the game AI model and the Game Engine model can easily be made the same , so that nothing unexpected ever happens.
This is a sort of cheating, but it one can even go further in the usage of control, and in doing so improve the game experience greatly. As an example of this, in Fear the player is made to believe that dying soldiers call for reinforcements and that this has a real impact on the game, since those also arrive afterwards. What happens actually is that reinforcements are only called for when they are arriving already anyway. This is a good example

of (ab)using complete knowledge to trick people with incomplete knowledge into believing one has more power or intelligence than one actually has. As such it is not unlike the way Tintin saves his and captain Haddocks life in Tintin Prisoners of the Sun (seeFigure 1) by pretending he is a sorcerer who has control over the sun, when in fact he merely knows there will be a solar eclipse, of which the Aztecs are unaware.

4.Finite State Machines

4.1 Definition of Finite State Machines

A Finite state machine (FSM) or simply a state machine is a model of behavior composed of a finite number of states, transitions between those states, and actions. As such it may well be modeled as a directed graph with a set of nodes S corresponding to the states, a set of directed edges T corresponding to the transitions between the states and a set of labels A corresponding to actions. To give a richer representation of Finite State Machines that allows an unambiguous representation of actions and their character, so called State Diagrams are used (see *Figure 2*).

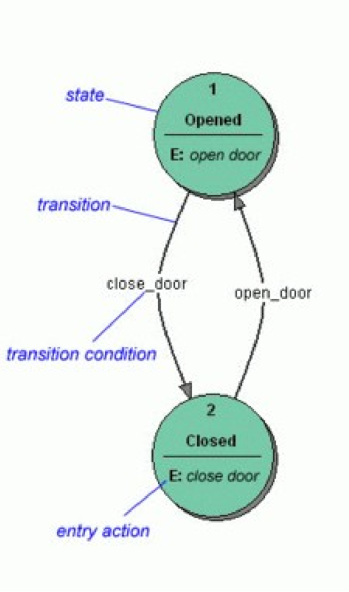


Figure 2: State diagram , source: Wikipedia

Actions come in different forms. There are **entry actions** and **exit actions** that correspond to actions that are executed when entering or leaving a certain state. Additionally there are **input actions** which are both triggered and given form depending on the present state and certain input parameters. Last there are **transition actions** which are performed when certain transitions are made.

4.2 Applications

Finite State Machines are used in diverse areas such as electrical engineering, linguistics and computer science. In electrical engineering they can be used to model simple circuits that control systems, such as vending machines. In linguistics they are mostly used to model language generation, and have proven very useful particularly for speech recognition and language understanding [21], for which problem the more specific form of *Hidden Markov* Models is used. Additionally Finite State Machines are used often in computer games to model and structure the game AI Logic. This is the application of our interest and we'll next give a more detailed description of it.

4.3 Application to Games

In Games Finite State Machines are mostly used to structure the game logic by separating it in a few sensible states. Those states correspond to distinct game situations and modules responsible for handling those situations. Every state is roughly independent of the other states in the resolution of game situations for which it takes responsibility. Only thorough global variables and input/export of information upon state transitions do different modules interact.
In *Quake3 Arena* [12] for example, there are states corresponding to attacking an enemy, chasing an enemy, retreating, finding an objective etc. The game logic is not at all limited to the states and state transitions.
Within every state a lot of scripting is taking place, a large part of which is dependent on the *BotState*. This means that the Finite State Machine is not very pure in the sense that much of its logic and scripting depends on things that are not modeled as a part of it. This however, is generally the case in games. There Finite State Machines are not meant to order all the game logic and keep all information about the game state, rather they function to roughly structure the code in a few sensible modules that can function more or less independently. In this way, the work of the AI programmer becomes separable into a few distinct cases, and code development, maintenance and modularity all benefit from that.

4.4 Advantages and disadvantages compared with planning

Finite State Machines have the advantage of simplicity and intuitiveness, but also the clear disadvantages of limited modularity and reusability compared to planning and other more thorough approaches towards game AI. When a programmer wants to adapt the AI behavior, the whole Finite State Machine must be changed, and often one will need to make similar adaptations at multiple places. In planning and other methods this problem is much smaller, as we will see in the next section. Another clearly undesirable feature of Finite State Machines is that the somewhat arbitrary state transitions much resemble the Spaghetti-code style of programming of the 80's with all those *goto* statements. That kind of programing makes it completely unclear how the flow of control is ordered, at least to everybody but the original programmer. When Finite State Machines grow in size by adding more states and transitions, the problem becomes more serious. At the same time, hacking much of the game logic within the states itself outside of the Finite State Machine is a worse option even. This approach is taken in *Quake3 Arena* were most of the actual work is done by calling specialized methods through System Calls. Those methods lack a good description, work with many variables that only exist locally and influence the game logic thoroughly by directly adapting the game state (*BotState*) which makes it very difficult for non specialists to understand what actually happens in the code, and even more difficult to adapt this.

Contrary, in planning methods there is a clear separation between the general planning method, and the problem specific actions and goals. Furthermore there can be made a nice separation between the declarative/symbolic description of actions and their effects and the actual game scripts corresponding to the execution of those actions. As such both may be adapted independent of each other, which greatly facilitates modularity and maintainability.

5.Planning and Search

5.1Approaches to Planning

Planning can be approached in a wide range of different ways. Those ways all have in common that they require a model of the world, most commonly defined in terms of states and actions over states, and specifically initial states and goal states. How actions and states are described and how the actual planning is performed can vary a lot for different problems, planning algorithms and concrete systems.
States can be discrete or continuous, they can be very specific and low level or abstract and high level. They can be large, containing many variable or small, containing only a few. Furthermore the states can be described in different formalisms. Often used for this purpose are propositional logic or first order logic. Alternatively knowledge description languages can be used, such as XML or the more formally defined Semantic Web infrastructures of RDF, OWL [22]. But states can be also much simpler, such as positions in space (which might be sufficient for robotic path planning) or simply lists of boolean attributes with values. Furthermore, some planning problems might require to model probability explicitly and somehow assign probability to states so that the uncertainty of actions can be reflected in the states.
Similar to states, actions can come in different forms as well. They can be as simple as adding a few integers to some variables, or changing the value of some booleans, or they can be as complex as requiring full simulations to be performed to compute the new state. They can be described purely on the level of logic in terms of changes of variables, or they may be generally defined by any function that manipulates the state. As such, only the input state and output state are important as far as planning is concerned, the details of the mapping between those states may be regarded as a black box.

5.2 STRIPS

The *STRIPS* [23] language, were *STRIPS* is shorthand for Standford Research Institute Problem Solver is one of the simplest planning languages and has been in the AI community for longtime. Originally it was used for controlling the robot *Shakey* [24]. In *STRIPS* both states and actions are to be described in a limited form of first order logic. States are basically just sets of predicates that describe facts that are holding true in the state, for example: inPosition(shakey, livingRoom). Actions consist of two parts, namely preconditions and postconditions, or results. Preconditions is a list of predicates that must hold true in order for the action to be allowed to execute. Postconditions is a list of predicates that will change as a result of the action. Originally *STRIPS* only used descriptions of preconditions and postconditions in the form of pure logic, and did not allow functional descriptions to be used for them. Allowing functions however generalizes the planning language and makes it more convenient to actually implement and work with. We therefore take this approach, and describe both the postconditions and the preconditions as functions. Furthermore we do not limit ourselves to first order logic, but rather allow the preconditions to put arbitrary constraints on the action. Similarly we allow the postconditions to be any function that can arbitrarily transform the state, as long as the result is still a valid state within our domain.

With this general definitions of States and Actions, we can now start to formally define plans, provided that we first introduce some terminology:

We start by defining states, actions and sequences of actions:

Def1: State

A **State** s is a list of predicates that holds true in the World at a certain point in time. Let States be the set of allowable states in the planning problem.

Def2: Action

An **Action a(s)->s'** is a function that maps a State **s** onto a new state **s'**. Let Actions be the set of allowable actions four the planning problem.

Def3: Action Sequence

An **Action Sequence** $l=[a_{1,}a_{2,}a_{3,.}.,a_n]$ is an ordered list of actions that is serially applied to a state **s** resulting in state **s'**. The result of applying part of an Action Sequence to a state **s** to get a new state **s''**, and then apply the rest of the Action Sequence to **s''** to get **s'** is the same as the result of applying directly the entire Action Sequence to **s**.

Formerly:
Let s' = App(l, s) be the state resulting of the application of l to s.
Then it holds that for every 1 <= i <=n:
App(l,s) = App($[a_{1,}a_{2,}a_{3,.}.,a_n]$,s) = App($[a_i,..,a_n]$,App($[a_{1,.}.,a_{n-i}]$, s))

Next we define the goal set as the the union of all states that are a goal.

Def4: Goals

Let $Goals=\cup_{goalState}\ goalState\epsilon States\wedge goal(goalState)$ be the set of goal States.

We can now give a nice recursive definition of a solution, defining it as any sequence of actions that either directly reaches the goal or can be split in an action that leads to a new state and a sequence of remaining actions that is a solution starting from this new state.

Def5: Solution

An Acion Sequence l is a solution for the planning problem starting from state s if:

1. $l=[a],a\epsilon Actions\wedge s\epsilon States\wedge App(s,a)\epsilon Goals$
2. $l=[a|RemainingActions]\wedge App(s,[a])=newS\epsilon States$
and RemainingActions is a solution for the planning problem starting from newS

5.3Search Algorithms

Having formerly defined the planning problem in terms of states, actions, goals and solutions, we effectively reduced the planning problem to a search problem. This problem is simply to search the sequence of actions that is a solution. For search problems there are a a lot of different Algorithms available, with different properties in terms of time usage, memory usage, optimality and the use of problem specific (heuristic) information. We will shortly review those algorithms next.

5.3.1 Greedy Algorithms

If we want to find a shortest/cheapest path, were all actions have equal length/cost then simple algorithms such as breadth first or iterative deepening [3] can be used. Those algorithms are optimal but quite inefficient, since they require to extensively search the entire search space up to the depth of the solution. If some problem specific information about the problem is available, then heuristics can help a lot to improve the search. Best First search makes use of a heuristic value to first extend the best, most promising partial solutions, in the hope of finding a solution faster in this way. This algorithm however is not guaranteed to be optimal.

5.3.2 The A* Algorithm

The A* algorithm combines the accumulated cost of incomplete solutions with heuristic information to find solutions much faster, while keeping a guarantee of optimality. The A* algorithm evaluates a cost function for every partial solution : f(s) =g(s) + h(s). In this formula g(s) is the accumulated cost of a partial solution, while h(s) gives an heuristic estimate of the remaining cost required to reach the goal when extending this partial solution. It uses this cost function to iteratively find the partial solution with lowest cost, and extend it into a new set of partial solutions which is then added to the list.

An important requirement for the A* algorithm is that that the used heuristic is strictly optimistic, that is never overestimates the remaining cost for partial plans. Provided that this is the case, it is easy to see that the first solution found by the algorithm is guaranteed to be the best one.

A* is a very nice and widely used algorithm, but a drawback are its great memory requirements resulting from the demand of keeping all partial solutions in memory while extending them. For this reason some similar algorithms have been developed that use less memory, namely Iterative Deepening A* - ADA* or Recursive Best First Search   RBFS. We refer to   Artificial Intelligence: A modern approach   [3] for more details on those algorithms.

We now present pseudo code for the A* Algorithm, but first we make a few comments about it.

The algorithm is usually described as a graph-search algorithm, but it is more general than that, and while most search spaces can somehow be represented as graphs, this is not always the most intuitive representation.

We take another viewpoint based on our definitions in section 5.2. We defined Paths as action sequences and Actions as operations that extend those action sequences into new, longer paths. The algorithm is then to be seen a Path-extension search algorithm, that extends incomplete paths by executing allowable actions until a goal state is reached. This is more intuitive in view of our initial discussion of STRIPS and our formal definitions concerning planning, and it also fits much better with our actual implementation.

Another comment is that one should define a maximum on the cost (or search-depth in case of uniform costs for actions), to assure the algorithm will finish. If this is not done, than certain actions may easily lead to cycles in paths, and without a limit on cost / search depth this will give a non-terminating algorithm.

The Algorithm:

```
function Astar(State initialState, CostFunction g, HeuristicFunction h, int
maxCost, Set goalSet)  returns AStarPath or Failure :

      // initialize
      AstarPath initPath = [InitialState];
      SortedPathList = [initPath];

      //loop
      while(nonEmpty(sortedPathList))
            AstarPAth toExtend = removeFirst(sortedPathList);

            if(accumulatedCost(toExtend, g) > maxCost)
                  break;

            if(lastState(toExtend)  ϵ  goalSet)
                  return toExtend;
            else
                  for Action a  ϵ possibleAcions(toExtend):
                        AstarPAth extendedPath  = extendPath(toExtend,a);
                        int cost = g(path) + h(path);
                        insertSorted(sortedPathList, extendedPath, cost);
                  end
            end


      return FAILURE;


function removeFirst(SortedPathList s):
      AstarPath p = remove first element from s;
      return p;


function accumulatedCost(AStarPath p, CostFuntion g):
      return g(p);

function extendPath(AStarPath toExtend,Action a):
      toExtend = [lastState|RestStates];
      State newState = applyAction(lastState);
      toExtend = [newState|toExtend];


function insertSorted(SortedPathList sortedPathList, AstarPAth path, int cost):
      insert path in sortedPathList at position i such that:
            cost( sortedPathList, elementAt(i-1)) <= cost   ∧
            cost( sortedPathList, elementAt(i+1)) >= cost

function elementAt(List l, int pos):
      return element at position pos in List;
```

Note:

With `List = [Head|Tail]` we denote the operation that takes a list
and splits it in the first element of the list (=Head), and the sublist of remaining elements (=Tail).

6. The Robot Approach

6.1 The Hierarchical Paradigm

The field of Robotics is concerned with many issues that are general topics for Artificial Intelligence and as a whole. These issues include Knowledge Representation, Understanding Natural Language, Learning, Planning and Problem Solving, Inference, Search and Vision. In fact research in Robotics, and the great interest the U.S. government had in it, was one of the first main drives for developing AI. This began during WOII with research on telemanipulator systems to be used in the Nuclear Industry to handle radioactive materials, and later continued with research on Robotic Technology to be used in the U.S. Space Program. The first approach in the development of autonomous Robot systems, was the so called **Hierarchical Paradigm**. In this framework Robots are build using a strict sequence of Sense,Plan and Act steps, that is repeated on each iteration. The Plan step relies on extensive models of the world, that are derived from the sensor data possibly combined with the history of previous perceptions and any world knowledge that is put into the model. The big problem with this approach is that it relies on a closed world assumption and thus has to deal with the Frame Problem of AI [19]. Another problem is that the Plan step often requires a lot of computation, which makes these kind of systems very slow and inadequate to respond real time to real events in the world.

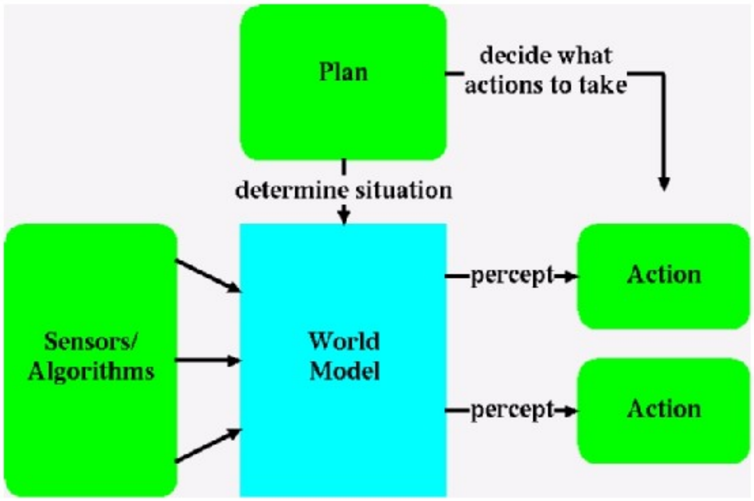


Figure 3: Schema of ***Hierarchical Paradigm***

6.2 The Reactive Paradigm

As a reaction on the many problems encountered in the Hierarchical approach, many researchers in Robotics started building Robotic systems using a Reactive approach, which is inspired by observations from biology and ethology. The systems within this new Reactive Paradigm made use of a direct coupling between sensing and acting, implementing complex emergent behaviors as a combination of many primitive behaviors that are fast,robust and computationally inexpensive.

In this paradigm perception is largely direct, and uses Affordances. Affordances are perceivable potentialities for action inherent in the environment (see [20] for details on this).

These Affordances are the direct releasers for the Behaviour which is executed by the actuators or motors of the robot.

Within the Reactive Paradigm there are two main approaches to combining the simple behaviors.

The first one is called **Subsumption** and was practiced by Brooks [17],[18] who is one of the most well known practitioners of the Reactive Paradigm. In a Subsumption Architecture higher layers can subsume or inhibit behaviors in lower layers. One main advantage of the Subsumption architecture is that it can effectively be directly implemented into hardware.

The second approach makes use of **Potential Fields** to trigger and combine behaviors of the robot.

Potential Fields are Vector fields, which can be directly computed from the perceptions (sensor readings) of the robots. One big advantage of Potential Fields is that the resulting actions are easy to model and combine as the sum of several Potential fields belonging to different behaviors, for example movement behaviors. This can give the Robot builder a good overview of the expected Behavior in all different situations, whereas in the Subsumption architecture such an overview is harder to visualize. The main point to notice however that in terms of expressiveness and potential for programming Robotic behavior these two architectures are basically identical.

The main disadvantage of the reactive approach is that whereas the programming of simple behaviors such as movement is straightforward, the extension to more complex procedural behaviors that can not easily be modeled as a sum of simple reactive behaviors is difficult if not impossible.

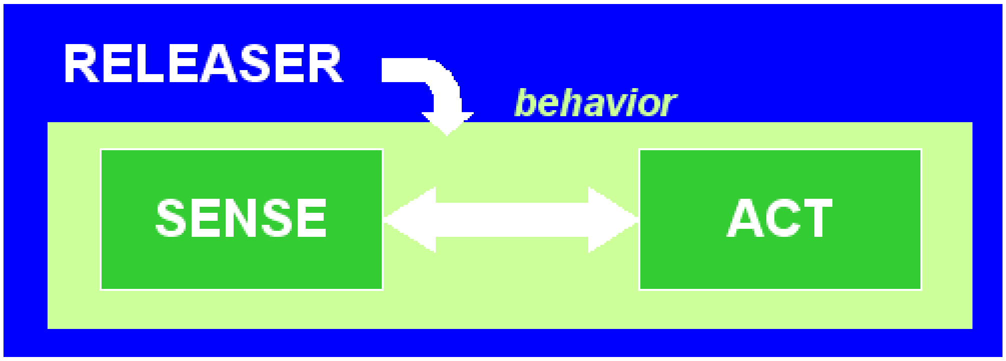


*Figure 4: Schema of **Reactive Paradigm***

6.3 The Hybrid Deliberative/Reactive Paradigm

To accommodate the problems of the Reactive Paradigm and allow for complex Behavior, while maintaining the great advantages of robustness, modularity and speed from the reactive paradigm, a new paradigm called **the**

Hybrid Deliberative/Reactive Paradigm came into being that intended to combine the good properties of both the Hierarchical and the Reactive approach. This paradigm can be described as: **Plan**, then **Sense-ACT**. The Plan box is concerned with all deliberation and global world modeling. The Robot first plans how to accomplish a mission (using a global world model) or a task, and then instantiates a set of behaviors (**SENSE-ACT)** to execute the plan or a portion of it.

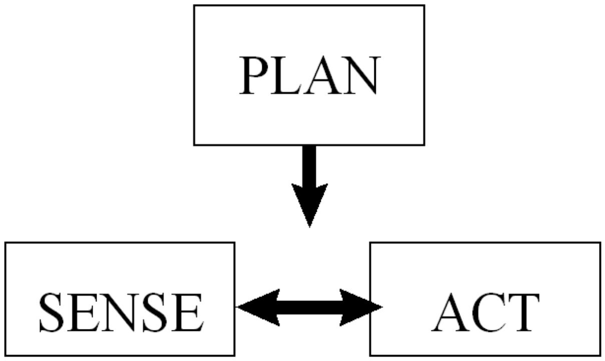


*Figure 5: Schema of **the Hybrid Deliberative/Reactive Paradigm***

Although most applications so far have been implemented using the Reactive Paradigm, more complex applications require a Hybrid Architecture in order to enable the integration of many complex modules including many systems (Communication Systems, Planning Systems, Position Determination Systems) that don't even have directly to do with the most basic functions of the robot. This strongly holds for complex game bots as well. While a game-bot should be reacting fast and real time to its environment, it also needs to plan complex behaviors at a high level in order to reach the its abstract goals. For these reasons the **Hybrid Deliberative/Reactive Paradigm** is most fit it.

6.4 Application to Quake and game-bots

From the above discussion it becomes quite clear that game-bots might profit a lot from the Hybrid approach that combines planning with reactive behavior with planning. It must be acknowledged that the Quake AI module is in fact already doing this in a way, but the problem with the implementation is that there are no neat separations between reactive modules and a planning system. The whole AI is structured around the Finite State machine which is implemented within *ai_dmnet.c* . In a sense the AI used in*F.E.A.R.* is more naive than the AI of *Quake3 Arena*, for it uses only planning and no reactive behavior, while *Quake3 Arena* uses both, though in an ugly way. In the ideal case both planning and reactive behavior are used in a modular and extendable way. The high-level planning is then taken care of by a general planning engine, that makes abstract plans and instantiates reactive action-execution modules that take care of the low level execution of the plan. The reactive modules are well able to deal with unforeseen small changes in the game world. They handle all events that don't need or don't afford planning to react to, since planning would become too complex or just too slow to facilitate them.

7.Game Engines: Dynamic Memory Allocation

7.1 The lacking availability of dynamic memory in Game Engines

Most programming languages provide the programmer with some form of dynamic memory. In Java and Python this is done at a very high level. This implies most of the details are hidden from the user, and memory allocation and freeing is taken care of by a virtual machine. On the contrary, in C and other lower level languages memory must be explicitly allocated and freed by the user himself, by calling specific methods. In C the methods *malloc*, *calloc*, *realloc* and *free* are available for this purpose, the first three for (re)allocating memory and the last one for freeing it.

Game engines could be seen as a sort of virtual machines or kernels in their own right, as the whole game is written and executed on top of them. This implies also that most functionality should be accessed through game engine specific interface methods, and for various reasons not all standard methods and packages may be available, or at least not in their normal form. In particular game engines tend to implement their own methods for memory management, as well as for giving input to and generating output from the game. The second issue is sometimes annoying, it simply means methods at *printf* are not available, one must import and use game specific counterparts to test things.

The first issue is far more serious. It means basic memory allocation methods such as *malloc* and *free* are not available, and the programmer is forced to use some game specific memory allocation procedure that may well not give the desired functionality. Performance is the reason Game designers use such limited, centralized memory allocation and management. But for everybody who wants to extend or change a game it is very problematic if only some specific data structures may be dynamically allocated, which is the case for quake. Furthermore the interface for generating or destroying such bounded-form objects in quake is not easy to use. Lastly, in quake many methods including those for memory management are only available at certain levels in the code, since the corresponding headers are only included in some of the c files. To change this, one basically has to adapt the whole makefile and with that the structure of the code. This makes it a cumbersome task to adapt the game based on the native methods. If one does not like the limited form of dynamic memory available, nor the complicated procedures required to access it, there is actually another solution available.

This other option is implementing an own memory allocation mechanism within quake on top of a big chunk of statically reserved memory. This is discussed in the next section.

7.2 Implementing Memory Allocation Mechanisms from the scratch

Malloc and *free* are basically just low level methods that can implemented in ways different from the standard available in c. In fact, some famous algorithms exist for memory management beyond the standard c-implementation of the *malloc* package. One of these is the *Doug Lea's* Malloc package [25], or *dlmalloc* for short. It does a good job at minimizing space and time of the algorithm, while considering portability, compatibility and maximizing locality at the same time. As such it is much better than the standard of many c environments.

Memory allocation methods can also be self-programmed from the scratch. While they normally use dynamic memory, one can also just allocate a big chunk of static memory and build a dynamic memory allocator on top of this. In this way one simulates dynamic memory but requires only static, which is exactly what is also done in the game engines themselves. Such an approach is good when no real dynamic memory is available, but wasting some memory by statically declaring it is not a big issue. When running somewhat older games on modern machines these premises hold.

In the book Computer Systems A programmers perspective [26] a nice minimalistic implementation of *malloc* and *realloc* is given, which we adapted to use in our system. We adapted the routines to use a heap based on staticly declared memory (a big char array). Originally it was still using dynamic memory as a basis, which in our case would imply missing the point of the whole exercise. Besides wasting some memory, our simple implementation has the disadvantage of being much less efficient than the normal *malloc* or thoroughly optimized *dmalloc* package. Still, for our purposes it is quite adequate. Our biggest gain is that we're freed from being obliged to use the *Quake 3 Arena* format of dynamic memory, which is a quite restricted one. Rather we can now create and work with the data structures and methods most natural to our application, without worrying how to convert them into some artificial format suitable for the dynamic memory allocation of the quake engine.

7.3 Assuring thorough freeing of memory in C

When working with C, and creating multi layered structures, one is confronted with the challenge of allocating and freeing correctly all memory used by those multi layered structures. When one deletes an object it is not enough to only free the pointer storing a structure at the highest level,since this structure might contain pointers containing sub-structures which would not automatically be freed this way. To tackle this problem satisfactory, a rigorous approach is needed. The approach we take is to supply all our structs with a pointer to a specific freeObject method, which should be called with the object as argument to free the object. Then, since all objects contain this method it is not necessary to know the object or its type to know how to free it. Different objects will have pointers to different freeObject methods, but these details are hidden to the user.
We must assure that any object that has multiple layers, meaning it contains pointers to other objects, will have a freeObject method that also takes care of freeing the lower level objects. This is achieved by simply using for every lower level object the contained freeObject method pointer to free it. As a last step the highest level object can be freed using the atomic *free* method on the pointer that stores it.

We thus have a systematic, recursive, transparent mechanism that assures all objects can be thoroughly freed, by using a uniform freeObject method stored in the same field in all objects. This approach can be easily extended to incorporate new structs. All one has to do is create adequate struct specific free methods when introducing those, and putting those in the freeObject field of the object when the new object is created. This is conceptually not too different from the more self-evident need for adequate struct constructors that apply the right sort of (possibly layered) memory allocation and field instantiation to create new structures of a certain type.

8. GOAP Implementation

8.1 Code overview

8.1.1 General Philosophy

The GOAP project was tackled by first defining the desired functionality of the planning system, then analyzing the existing *Quake3 Arena* code, and finally making a list of required subcomponents to achieve this functionality. The Quake code itself turned out to be quite complex an difficult to adapt. We therefore split the project in two parts. One part consisted of getting things working in quake. To this end we started with the implementation of the much simpler *Pandemonium* [27] architecture. This consists basically just of a set of behaviors implemented as scripts for the bot, as well as a set of activation functions that determines which behavior is executed at any time in the game. The second part consisted of implementing the symbolic planning with *A** in C.

8.1.2 Vector code

After the general plan was sketched out, everything was developed in a bottom up way, creating basic building blocks and solving new problems as they arrived. Since C does not contain such a nice set of standard methods and Data Structures as Java does, we needed to create a lot of code from the scratch. We build our own Vector struct containing all the methods to generate a Vector and add, retrieve and remove elements from it.

8.1.3 Sorting Plans

For the *A** Algorithm we needed a sorting algorithm to keep the list of incomplete plans sorted on cost. We found an efficient *Heapsort* algorithm for this purpose on the internet [2], which we adapted to our purpose. It is noted that different algorithms and Data Structures, might give better performance than a naive re-sorting of the whole list after the insertion of the new paths. Concretely, red-black trees [1] could be a better alternative because they allow for both O log(n) insertion and O log(n) removal of maximum/minimum elements from the Data Structure. This requires a lot of extra work however, and only gives achieves performance gain on part of the sorting efficiency, which is really a side issue in our project, so we did not implement this.

8.1.4 A* and Towers of Hanoi

We developed a implementation of *A** using the very simple state space of the Towers of Hanoi as our testing environment. While we started with very simplified problem, we set up everything in a very general way, allowing arbitrary functions for preconditions and postconditions. We use actions and States. Actions contain pointers to functions that check whether preconditions hold and pointers to other functions that compute the postconditions. This abstract implementation made it easy to adapt the planning for the *Quake3 Arena* world, or indeed to any other planning problem.

The implementation of the *A** algorithm is based on its discussion in Russell & Norvig.
[3]. Currently our implementation only supports for one goal to be planned for.
In case we want to plan for different goals, we either have to run our planning problem once for every goal, or we must adapt it so that it continues searching until paths have been found for goals have been reached or a certain path length is reached. After this is done, it remains to chose a best path amongst the different solutions that solve different goals. This can be done by either assigning an absolute value to different goals, or use some cost/benefit ratio, possibly taking the current bot status into account. Either way it is clear one can extend in many ways beyond basic *A** with just one goal, but unfortunately, we already had our hands more than full with the basic implementation.

8.1.5 Pandemonium

The finite State machine in Quake works by keeping a pointer to a function in the *BotState* called ainode that implements a certain state of the quake FSM. To adapt it we created new AINode_* and AIEnter_* functions in ai_dmnet.c and ai_dmnet.h, which correspond to the actual code of the state and a method to enter the state. We furthermore adapt ai_dmq3.c to immediately enter our new GOAP node AINode_Pandemonium once a new bot is set up. From within this new node we then call all our other functions. Our Pandemonium implementation consists of a a simple array of behavior scripts and an associated array of activation-value functions (shrieks). The behavior scripts execute basic actions such as jumping, moving to an opponent, attacking an opponent and moving to an picking up an item.

8.1.6 Action Scripts

Our action scripts, which are both used by the Pandemonium as well as by the planning implementation, are based on lower level *Quake 3 Arena* functions that do all the actual work.
Those lower level functions often have a quite limited functionality. The function to get an item assumes one always wants the best item, were best is defined by some other hidden lower level functions. Since many data structures are only defined locally in the file were they are used, it is quite complex to move functionality from those lower level functions up to a higher level, so that one can get more control over what is done. In some cases this is what we did however. In ai_dmq3.c we created a new function *GOAPBotFindEnemyWithEntityNum(bot_state_t *bs, int entitynum)* that comes in place of the function *BotFindEnemy(bot_state_t *bs, int curenemy).* The old function gave no way to specify what enemy to go form, in the new function this can be specified by means of an *entitynum,* which is a descriptor for the most general Quake structure entity .
To get an item, there is no straightforward approach either. What must be done is to create an item goal, and then to execute this goal with another function. The function for creating an item goal, *BotChooseLTGItem(int goalstate, vec3_t origin, int *inventory, int travelflags),* is defined inside another obscure file called be_ai_goal.c inside the folder botlib . This function calls another function that computes fuzzy weights for items, and then chooses the best item based on those weights. To recap, in the original code it is possible to get an item, but not possible to specify which item at a high level in the code.

If we want to take planning seriously, we should have a list of all items and decide which one to get at the level of planning, instead of hidden in some low level obscure function. Problem is, getting a list of items can only be conveniently done within the file be_ai_goal.c , since the relevant Data Structures that keep track of the items in the game are only defined within this file. Of course we could, and probably should adapt the makefile to get these data structures to be defined globally, but this implies changing the whole code structure and is quite hard to do since it requires a lot of expert knowledge about makefiles, compilation and linking. Unfortunately there is no real alternative.

8.1.7 Planning in Quake

The final planning in Quake is effectuated by combining the Planning code with the developed Action scripts. To this end Action structs contain a pointer to the actual action execution script that corresponds to the symbolic action. We created a function execDeamon_Planning that checks whether actions have been planned and if so executes the next action, and if not plans new actions. It uses two helper function planActions and executePlannedActions to do this. Our states and actions are currently limited to the positions and health of opponents and the bot itself, and planning is only concerned with finding the best opponent and moving to and killing this opponent. While this is a limited model, it suffices as a proof of concept. Clearly it would be more interesting to have goals for getting items as well, or for retreating when one is almost dead, or even more complex things. We didn't got to implement those extensions yet, but we did create a complete framework that should make their implementation a lot easier, even though some of those improvements would require far reaching adaptations of the code structure as discussed in the previous section.

8.2 Module Structure

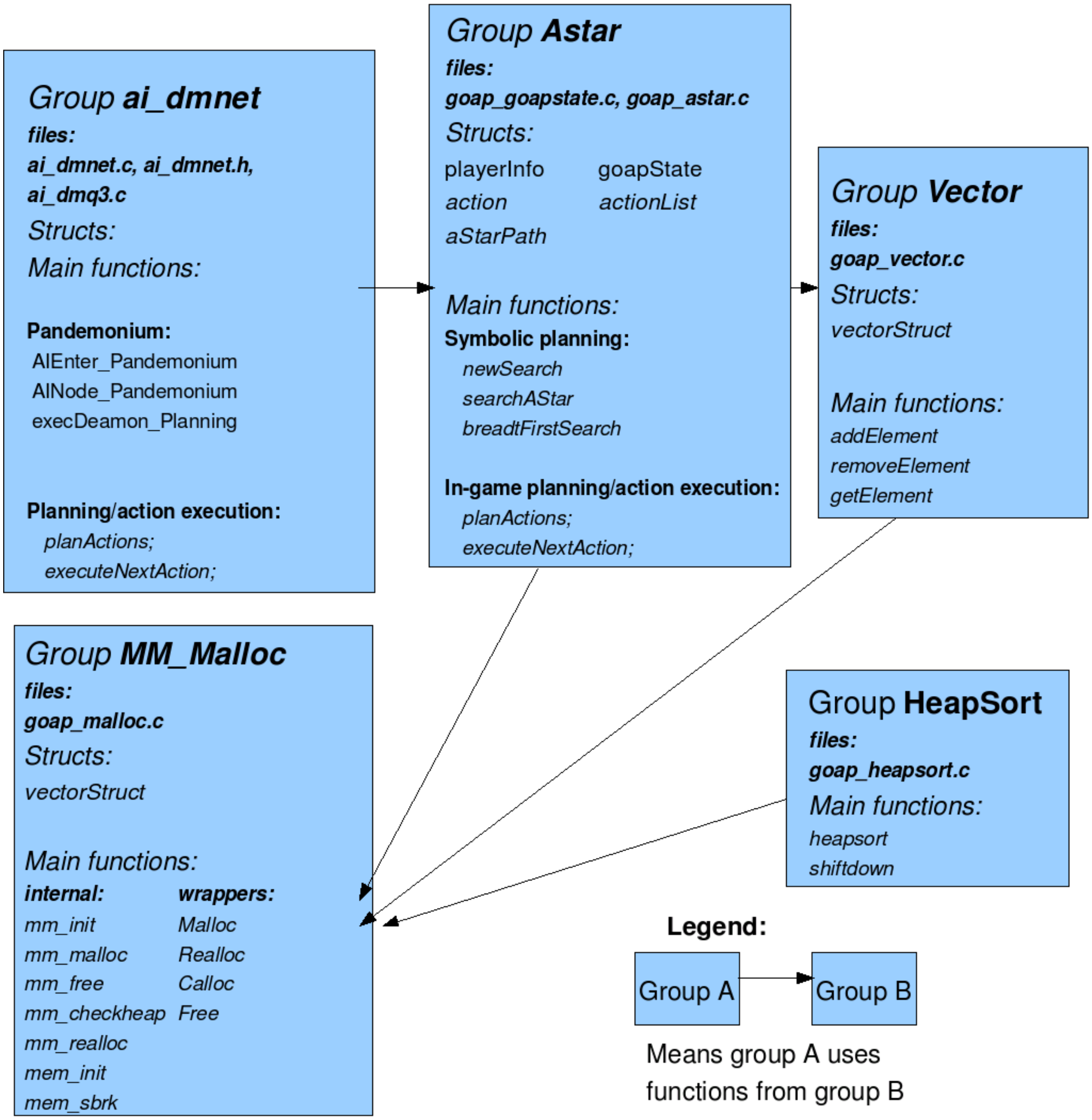
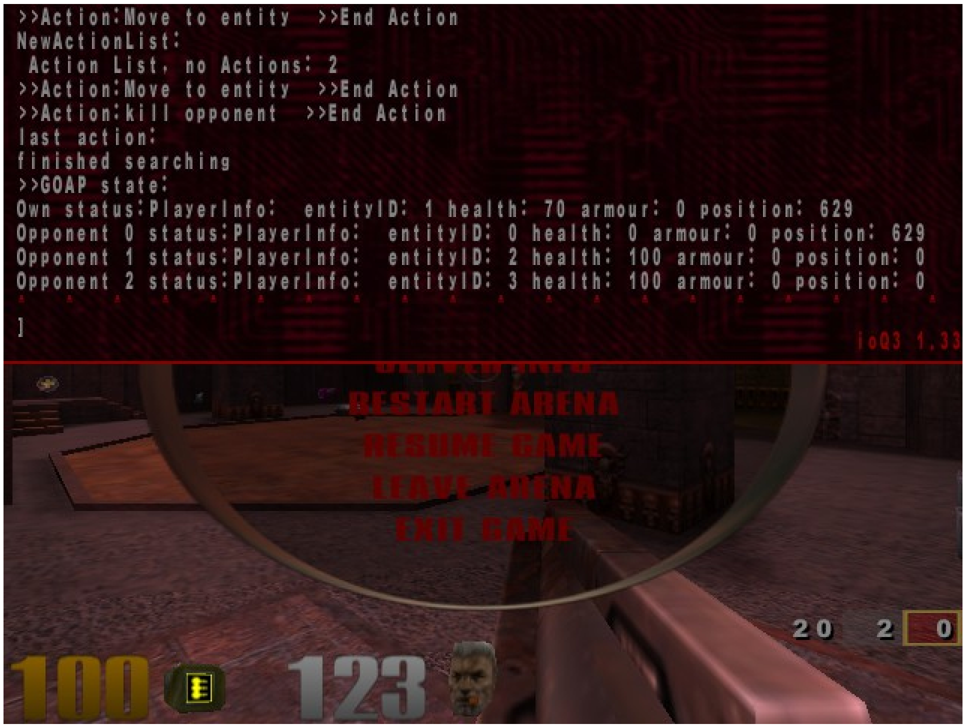


Figure 7: Schema of the structure of the code

9. Results

We first tested the Pandemonium architecture and the planning module separately and verified their compliance to our specifications. After this we integrated the planning with the quake pandemonium code, and after a lot of debugging got this working as well. A problem we encountered, as also mentioned in the section on Game engines, is that many standard c-functions are not available in the quake framework. Thus, in order to get things working again within quake we had to replace all kind of basic functions like *printf* and *memcpy* with their respective quake counterparts. At first we ran planning on every iteration. Apart from being ideologically unsatisfying, this also strongly slowed down the game. We then changed this, by keeping a flag that indicates whether a plan is available, or planning should be done. On every iteration the bot executes one action or does the planning.



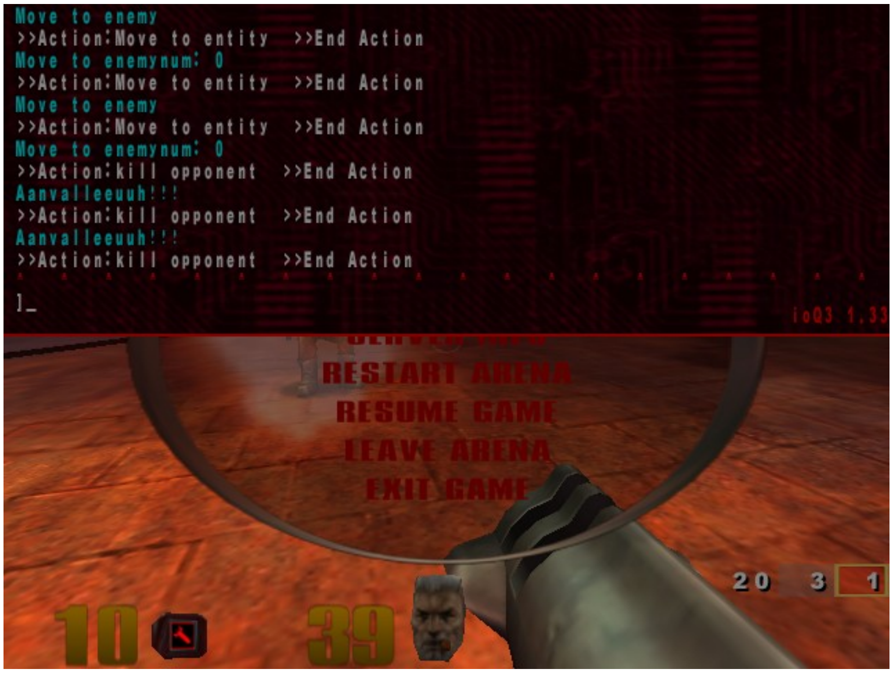
```
>>Action:Move to entity  >>End Action
NewActionList:
 Action List, no Actions: 2
>>Action:Move to entity  >>End Action
>>Action:kill opponent  >>End Action
last action:
finished searching
>>GOAP state:
Own status:PlayerInfo:  entityID: 1 health: 70 armour: 0 position: 629
Opponent 0 status:PlayerInfo:  entityID: 0 health: 0 armour: 0 position: 629
Opponent 1 status:PlayerInfo:  entityID: 2 health: 100 armour: 0 position: 0
Opponent 2 status:PlayerInfo:  entityID: 3 health: 100 armour: 0 position: 0

]
```

Figure 8: Result of A planning. The final action list (=plan) consists of the actions move to entity and kill opponent.*

Typically a game action, such as moving to an opponent, has to be repeated many times before it's desired effect is reached, in this case standing next to the opponent. For this reason actions are repeated until their desired effect is achieved (see *Figure 9* and *Figure 10*).

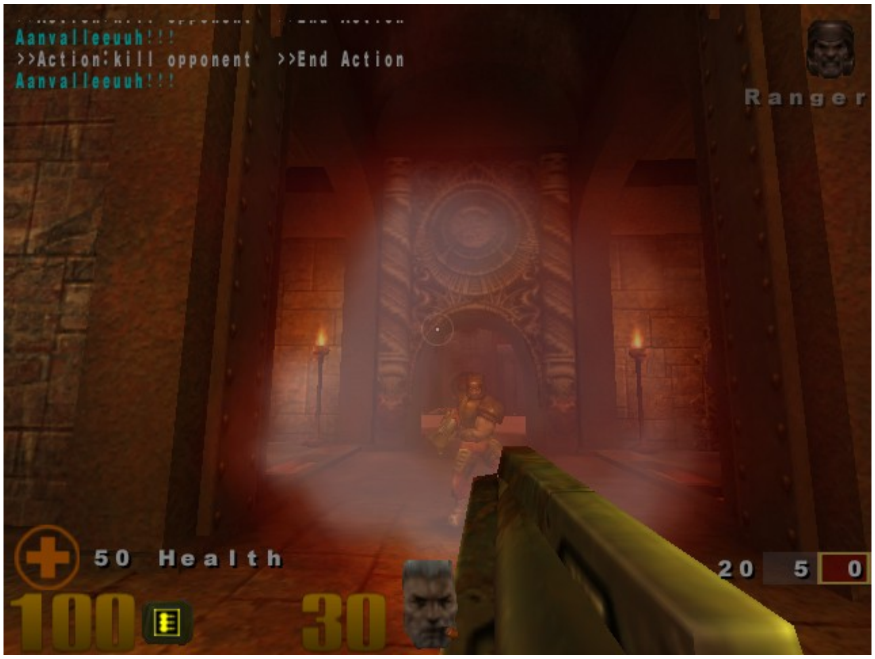
We finally notice that our program shows the advantages of planning, but also it's strong disadvantages when applied in it's pure form. When moving to an opponent, or executing any other action, the bot is doing just that and is not adequately reacting to any other thing in the world. Perhaps another opponent is attacking from behind, or an item lies just nearby and could have easily be picked up on the way?



```
Move to enemy
>>Action:Move to entity  >>End Action
Move to enemynum: 0
>>Action:Move to entity  >>End Action
Move to enemy
>>Action:Move to entity  >>End Action
Move to enemynum: 0
>>Action:kill opponent  >>End Action
Aanvalleeuuh!!!
>>Action:kill opponent  >>End Action
Aanvalleeuuh!!!
>>Action:kill opponent  >>End Action

]_
```

Figure 9: Actions are implicitly repeated until they are completed. Here move to enemy is completed and the attack action is commenced.



```
Aanvalleeuuh!!!
>>Action:kill opponent  >>End Action
Aanvalleeuuh!!!
```

Figure 10: The attack is repeated until the enemy is dead.

The way to resolve this disadvantage is not to make planning more complex, but rather to combine it with reactive behavior. This means planning should only take place at a very abstract level, and then reactive modules should be instantiated to execute those plans. Those reactive modules ideally have a lot of flexibility in dealing with unforeseeable events and opportunities, so that the best of both planning and pure reactive behavior are united. This is exactly the hybrid paradigm as discussed in section 5, and should be the next step in bot AI.

10. Discussion and Conclusion

Our project started with the relatively simple idea of implementing planning for *Quake3 Arena*. On thew way however, we encountered and had to resolve many side issues such as memory management, the inherent complexity of the Quake source, path sorting,and the construction of many required Data Structures in c from the scratch. Consequently, we learned a lot about c programming and makefiles as part of the project. Overall then, this project which seemed simple at startup, turned out to be very complex and challenging as well as time consuming in practice.

We would have liked to further implement more complex forms of planning, involving multiple goals, as well as to increase the repertoire of actions available to the bot. We do feel however, we did a very good and thorough job at creating the main infrastructure required to do planning within Quake, and certainly hope that others might benefit from our work to take this to the next level.

One thing our project clearly showed is that planning alone is not enough. It might still work in the simpler situation of single player shooters with limited freedom of movement on part of the bots, but for tournament bots it simply doesn't suffice. As mentioned in our section on Robotics and else in the report, we think the next move would be to combine the planning with reactive behavior, as has already been done for normal robots, but not very explicitly and cleanly for game bots. We look forward to see the results of such an approach.

References

[1] Michael T. Goodrich, Irvine Roberto Tamassia, Data Structures and Algorithms in Java, 2nd Edition , *John Wiley & Sons, Inc. ISBN: 0471383678 ©2001*

[2] Ariel Faigon. A library of internal sorting routines .©*1987* http://www.yendor.com/programming/sort/

[3] Stuart Russell, Peter Norvig. Artificial Intelligence: A Modern Approach, 2nd edition .©*2003 Prentice Hal ISBN: 0137903952*

[4] Zachary Slater, Aaron Guyes et. al. Quake 3 Arena Source http://ioquake3.org/

[5] Quake 3 Arena Source documentation *http://www.nanobit.net/quake3_doxy/index.html*

[6] MingW Minimalist GNU for Windows . http://www.mingw.org/

[7] Black & White . http://www.lionhead.com/bw/

[8] Epic Game developers. Unreal Tournament http://www.epicgames.com/

[9] G. Kaminka, S. Schaffer, C. Sollitto , R. Adobbati, A. N. Marshall, A. Scholer, S. Tejada . Gamebots: A 3D Virtual World Test-Bed For Multi-Agent Research *In Proceedings of the Second International Workshop on Infrastructure for Agents, MAS, and Scalable MAS, (Montreal, Canada, 2001).*

[10] Laird, J. Using a Computer Game to Develop Advanced AI .*Computer, 34 ,7 (Jul. 2001), 70-75.*

[11] J.Orkin, 3 States and a Plan: The AI of F.E.A.R. . Game Developer's Conference Proceedings (2006)

[12] J. P. van Waveren. The Quake III Arena Bot *Master thesis 2001 Delft University of Technology.*
http://www.kbs.twi.tudelft.nl/Publications/MSc/2001-VanWaveren-MSc.html

[13] ID Software developpers. Official Quake Site .*http://www.idsoftware.com/*

[14] Vivendi Universal Games, Inc. F.E.A.R. © *2006 Monolith Productions, Inc. All rights reserved.*
http://int.whatisfear.com

[15] S. M. Woodcock Games Making Interesting Use of Artificial Intelligence Techniques .
http://www.gameai.com/ai.html

[16] R.S. Sutton, A.G. Barto, Reinforcement Learning: An Introduction ,*MIT Press, Cambridge, MA, 1998*

[17] Brooks,R., Challenges for Complete Creature Architectures , proceedings of First International Conference on Simulation of Adaptive Behaviour, pp 434-443

[18] Brooks,R.A., A Robust Layered Control System for a Mobile Robot ,*IEEE Journal of Robotics and Automation*, vol. 1, no.1, 1986, pp.1-10.

[19] Dennett, D. (1987), Cognitive Wheels: The Frame Problem in Artificial Intelligence .
in Pylyshyn (1987).

[20] R.Murphy (200), Introduction to AI Robotics ,*MIT Press 2000*

[21] D. Jurafsky and J. H. Martin. SPEECH and LANGUAGE PROCESSING: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition . Prentice-Hall, 2000
ISBN: 0-13-095069-6

[22] World Wide Web Consortium (W3C). Resource Description Frameword (RDF) .
http://www.w3.org/RDF/

[23] R. E. Fikes, N.Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving .*Artificial Intelligence, 5(2):189 208* (1971).

[24] Arrtificial Intelligence Center . List of Shakey Publications *http://www.ai.sri.com/shakey/*

[25] D. Lea. A Memory Allocator .*http://g.oswego.edu/dl/html/malloc.html*

[26] R.E. Bryant, D. R. O'Hallaron. Computer Systems: A Programmer's Perspective
©2003 Prentice Hall. ISBN-10: 013034074X

[27] Selfridge, O.G. 1959. Pandemonium: A Paradigm for Learning *In Proceedings of the Symposium on Mechanization of Thought Process. National Physics Laboratory.*

Appendix A :Installing Quakevolution

If Windows is used, using *MingW* [6] is the best option, much simpler than working with Visual C++ in our experience. A good description on how to install stuff can be found under [4] as well. In fact, using *MingW*, the installation under Windows is simpler than under Linux. One drawback remains is that one has to copy the newly compiled files from the source director to the game directory that contains the .pak files and the executable, every time the code is changed and recompiled. In Linux one does not have this problem.

The following gives an idea of how Quake3 Arena can be installed on a Linux machine. Some things will need to be adapted, depending on you're specific system. Note that we finally were forced to work with windows ourselves, because neither of us could get the video card working properly under Ubuntu Linux to support 3d graphics as required for *Quake3 Arena*.

Installation under Linux, General plan:

1) Save the next lines in an .sh file:

```
#!/bin/sh
echo ""
echo "Quake 3 automatic installer"
echo "Made by Laurence Muller and based on file modifications by Jeroen Bédorf"
echo "Adaptation by Attila Houtkooper"
echo ""

TARGETINSTALLDIR=`pwd`
echo "Getting file from http://staff.science.uva.nl/~aldersho/GameProgramming/Quakevolution.zip"
cd $TARGETINSTALLDIR
wget http://staff.science.uva.nl/~aldersho/GameProgramming/Quakevolution.zip
echo "Unzipping file (Quakevolution.zip)"
mkdir Quakevolution
chmod -R u+w Quakevolution
unzip Quakevolution.zip
```

```
        cd Quakevolution
        echo "Unzipping file (qvsource.zip)"
        unzip qvsource.zip
        mv qvsource/* --target-directory=$TARGETINSTALLDIR/Quakevolution/
        mv quake3-latest-pk3s/* --target-directory=$TARGETINSTALLDIR/Quakevolution/
        echo "Fixing filenames and more..."
        dos2unix compile.sh
        mv $TARGETINSTALLDIR/Quakevolution/include/al/ $TARGETINSTALLDIR/Quakevolution/include/AL/
        mv $TARGETINSTALLDIR/Quakevolution/src/code/QV/qv_fileio.c $TARGETINSTALLDIR/Quakevolution/src/code/QV/qv_fileIO.c
        mv $TARGETINSTALLDIR/Quakevolution/src/code/QV/qv_fileio.h $TARGETINSTALLDIR/Quakevolution/src/code/QV/qv_fileIO.h
        mv $TARGETINSTALLDIR/Quakevolution/src/code/QV/qv_geneticalgorithm.c $TARGETINSTALLDIR/Quakevolution/src/code/QV/qv_geneticAlgorithm.c
        mv $TARGETINSTALLDIR/Quakevolution/src/code/QV/qv_geneticalgorithm.h $TARGETINSTALLDIR/Quakevolution/src/code/QV/qv_geneticAlgorithm.h
        mv $TARGETINSTALLDIR/Quakevolution/src/code/QV/qv_testenvironment.c $TARGETINSTALLDIR/Quakevolution/src/code/QV/qv_testEnvironment.c
        mv $TARGETINSTALLDIR/Quakevolution/src/code/QV/qv_testenvironment.h $TARGETINSTALLDIR/Quakevolution/src/code/QV/qv_testEnvironment.h
        dos2unix $TARGETINSTALLDIR/Quakevolution/src/code/tools/lcc/src/dagcheck.md
        chmod 777 compile.sh
        chmod 777 $TARGETINSTALLDIR/Quakevolution/src/code/tools/lcc/build-linux-i386/lburg/lburg
```

2) Change all paths in /tmp/ahoutkoo/Quakevolution/compile.sh from /scratch/ to /tmp/ahoutkoo/

4) Change all paths in /tmp/ahoutkoo/Quakevolution/src/makefile from /scratch/ to /tmp/ahoutkoo/

5) Save the following in an .sh file and execute:#!/bin/sh

```
        TARGETINSTALLDIR="/tmp/ahoutkoo"
        export TARGETINSTALLDIR
        $TARGETINSTALLDIR/Quakevolution/compile.sh
        cd $TARGETINSTALLDIR/Quakevolution/baseq3
        echo "Getting pak0.pk3 file from http://student.science.uva.nl/~lmuller/pak0.pk3"
        wget http://student.science.uva.nl/~lmuller/pak0.pk3
```

```
cd $TARGETINSTALLDIR/Quakevolution/
dos2unix run.sh
chmod u+x run.sh
cd $TARGETINSTALLDIR/
rm Quakevolution.zip
echo "Done!"
echo "To get normal game speed remove +set timescale 15 from run.sh"
```

6) Change all paths in $TARGETINSTALLDIR/Quakevolution/run.sh from /scratch/ to $TARGETINSTALLDIR/