LGOAP: adaptive layered planning for real-time videogames

Giuseppe Maggiore*, Carlos Santos*, Dino Dini*, Frank Peters*, Hans Bouwknegt*, and Pieter Spronck†
*NHTV University of Applied Sciences, Breda, NL
E-mail: {maggiore.g,santos.c,dini.d,peters.f,bouwknegt.h}@nhtv.nl
†University of Tilburg, NL
E-mail: p.spronck@uvt.nl

***Abstract*—One of the main aims of game AI research is the building of challenging and believable artificial opponents that act as if capable of strategic thinking. In this paper we describe a novel mechanism that successfully endows NPCs in real-time games with strategic planning capabilities. Our approach creates adaptive behaviours that take into account long-term and short term consequences. Our approach is unique in that: *(i)* it is sufficiently fast to be used for hundreds of agents in real time; *(ii)* it is flexible in that it requires no previous knowledge of the playing field; and *(iii)* it allows customization of the agents in order to generate differentiated behaviours that derive from virtual personalities.**

***Keywords*—Games, AI, planning**

I. Introduction

Non-playing characters (NPCs) in modern games often exhibit behavior that is strategically and tactically inferior. In general, NPC behaviour is clearly artificial and unrealistic. This leads to a loss of immersion on the part of the player, as whenever the player pays close attention to an NPC, he must conclude that the NPC just wanders around aimlessly. Such NPC shortcomings are found even in high-profile games. In *Mass Effect 3* [1], for example, NPCs outside of battle areas remain where they are, with the rare exception of NPCs that move along pre-programmed paths, only to reach their destination and then do nothing further. In *Fable III* [2], similarly, NPCs move randomly among the buildings in the various cities and villages, without any apparent goal. In both *Mass Effect 3* and *Fable 3* the NPCs' AI can be called inferior because the NPCs: *(i)* are not interacting with the objects of the game world in a way that is similar to the player; *(ii)* do not try to achieve a long-term goal; *(iii)* act mainly as moving decorations; and *(iv)* do not follow any of the implicit rules of the game world such as the need to sleep, eat, earn money, etc. Enemy NPCs are able to fight the player or patrol certain locations, but they do not do so for any specific reason beyond "being programmed to do it", nor do they take advantage of the environment; for example, NPCs rarely use doors or other items to create barricades, or make strategic use of the ammo or med-packs that are strewn about the levels.

We argue that many games would benefit of having NPCs who are part of the game world as much as the player is, in that they *play* the game rather than *be a passive part* of it. Anything that the player can do, NPCs should be able to do as well. NPCs should be able to recognize the actions available to them, and choose the best course of action because it makes sense according to their internal logic, either to fulfill their needs or to reach some goal.

Simulating such behaviors requires a mechanism for computational reasoning. We believe that the techniques based on planning [3], which are used for the AI of board games such as [4], have the best chance of solving such a problem. Notably, planning is linked with logic programming, one of the four main programming paradigms. Planners enjoy the same expressive power as logic programming itself. However, these techniques have hardly been explored in real-time videogames, where the ability to compute and execute decisions quickly for a changing environment is considered to be more important than the ability to construct good plans.

Our objective in this paper is to lay the groundwork for NPCs that: *(i)* play according to the same rules as the player; *(ii)* try to achieve their own goals in the game; and *(iii)* exhibit adaptive, human-like behaviours. Since our work is aimed at real-time games, the techniques we explore should also be fast to compute: where in chess the computing time for the next move can be minutes, in a real-time game it is milliseconds. Moreover, executing plans in real-time adds a further challenge: the initial conditions that made the planner choose a specific course of action may change, for example because of the unpredictable effects of the player's actions.

We now state the problem statement for this work: **to what extent can planning be used to create adaptive behaviour of NPCs that allows them to achieve long and short term goals in a real-time game or simulation?**

We argue that the use of planning can improve video games as a whole significantly. The first and foremost advantage is a *stronger emotional involvement* from the player. Instead of feeling alone and surrounded by scripted entities, the player could feel part of a virtual reality which NPCs appear to actually *inhabit*. Secondary characters could have their own goals and their own agendas, which they would actively try to achieve. Interesting NPC behaviour could lead to higher playing time of a game, because of the arising of unexpected situations. By influencing the magnitude and frequency of unexpected situations (for example, see [5]), the game could even make use of *procedural story-telling* to generate side-stories to which NPCs would react naturally. This could be an innovative tool in the hand of game designers, who could define non-linear stories and plots.

To describe our approach, we start by giving a uniform notation for representing the game entities, the NPCs, and the actions available to them (Section III). This notation easily allows to change, either in real-time or at design-time, the game world. The NPCs then adapt to the new state of the world. We then give our planning algorithm (Section IV), a *layered* variation of Goal-Oriented Action Planning [7], which we named *LGOAP*. The LGOAP algorithm allows NPCs to find *adaptive plans* in *real-time* in order to achieve their own *goals* given the *actions available* to them, and thanks to the *layering* system the algorithm handles the creation of plans that span a *long period of time*.

We describe (Section V) how plans are carried out by comparing the NPCs' expectations to the actual results in the game world. We then discuss the use of our planning system to animate NPCs in a randomized virtual city, which acts as our case study. This case study allows us to evaluate (Section VI) our approach to verify that our planner can be used in real time, for large numbers of NPCs concurrently, and with good resulting behaviors.

We compare our LGOAP algorithm to related approaches (Section VII) before we draw conclusions (Section VIII). Finally, we describe our future work into significant extensions to our approach (Section IX), such as planning for social interactions, long-term planning through the addition of layers, and even planning based on incomplete information on the game world and learning.

II. The idea

Our approach rests on one main idea: *create and execute, in real-time, plans that consist of sequences of actions*. Actions are defined as transitions in the game world state; multiple actions are available to an NPC in a given state, but only one can be chosen for the final plan. We define the game world as a distribution of resources over entities and agents, and actions cause only changes to that distribution. This allows us to consider actions as transitions in a graph, across which the planner finds an optimal path in order to reach the desired state.

This approach as presented requires answering some further questions:

- how do we model an abstract game world so that the approach works on any concrete game world without modification?
- what search algorithm can we use that can be executed sufficiently fast?
- how can we deal with NPC personalization?
- how can we deal with the fact that an NPC can only plan his own actions, while the future world state depends on the actions of all NPCs?

In the next sections we will answer these questions and we will also provide with an evaluation that quantifies the quality of our answers.

III. Entities, actions, and agents

We start by defining a uniform model of the game world. An adaptive AI needs such a model in order to be able to react to changes made to the game world.

A. Game world

We define the game world as a series of *entities* (such as a table, a chair, a gun, a gem, etc.) and a series of *agents* (the playing and non-playing characters). Entities may change dynamically: their locations may vary and they may be added or removed from the game world. One aspect that is fixed for all entities is that they all contain *resources* (or *stats*). The same resources are also stored for each agent. When an agent interacts with an entity, it does so by activating an *action*. An action is responsible for moving resources between entities and agents; this movement may be in both directions; that is: both the agent and the entity may increase or decrease their internal resources as a result of the action. Actions may also modify the location of the entity or the agent, and may take time to be executed. Actions may change dynamically and may be added or removed.

B. Example scenarios

Consider an example scenario: a game where the agent must reach the door at the end of a linear dungeon. The resources of the game entities are `life` and `attack`. The entities of the dungeon are the `sword`, which increases the attack resource, an `ogre` that guards the exit, and a series of `tiles` that can be traversed by the agent. The shape of the dungeon is the following:

```
L=life        Sword      Agent      Ogre
A=attack       L=1        L=5        L=5
*=entity       A=2        A=0        A=1       Goal
-=tile          *----------*----------*----------*
```

The available actions in this scenario, which are attached to the appropriate entities, are:

- `Sword.pickup`, which increases the attack rating of the agent who picks it up and decreases the life of the sword;
- `Tile.walk` that changes the current location of an agent;
- `Ogre.fight` that subtracts the attack resource of the agent from the life resource of the ogre and vice-versa.

Our framework for defining games allows for extensions without much effort. For example, we could add a new resource, `key`, which represents ownership of keys. We can now add a room to the dungeon which requires a key in order to obtain the sword. The new shape of the dungeon becomes:

```
L=life                 Agent      Ogre
A=attack     Key        L=5        L=5
K=key        K=1        A=0        A=1       Goal
*=entity      *----------*----------*----------*
_=tile                   |
                         | Door
                         *  K=1
                         |
                         | Sword
                         *  A=2
```

The new actions are `Key.pickup`, which adds to an agent a key resource, and `Door.open`, which subtracts from the door key resource the agent's key resource.

C. Logic programming

The encoding of the problem as described above is inspired by constraint logic programming [6]. Actions change the resources of the various entities of the game world if some preconditions are true. For example, `Sword.pickup` would be defined as:

```
Agent.L > 0 ∧ Sword.L > 0 ∧ Agent.P = Sword.P →
    Sword.L := 0 ∧ Agent.A := Agent.A + 2
```

Thanks to this unified framework, it becomes easier to define an adaptive planner that may combine general purpose actions without needing knowledge of them. Also, specifying a goal becomes the definition of a series of conditions on NPC resources.

IV. Planning algorithm

In our approach, a game state is no more and no less than a distribution of resources over available entities and agents. Actions, executed by the player or by NPCs, move the game world from one state to another. A problem of general-purpose NPCs is that they blindly pick one action at a time, without considering the medium and long term consequences of their decisions. Planning solves this problem by giving the AI the ability to select *sequences* of actions. In this section we show how to adapt planning to real-time games.

A. Naïve planning

Planning in a game can be done naïvely with backtracking on all the possible sequences of actions. An example of such a naïve implementation is "Goal-Oriented Action Planning" (GOAP) [7]. Such backtracking ends after a satisfactory plan (one that reaches the goal while ensuring the agent's survival) is found, or when the best plan is discovered. As an acceptable plan not always exists, appropriate failure conditions must be taken into account; for example, an upper bound over the maximum number of actions must be used to avoid exploring plans which are too long.

Back-tracking is a simple forward chaining state space search [8] which is effective but inefficient. Assume that at every step there are at least A actions available, and that plans with more than N actions are discarded. Every possible sequence of actions of length N may be explored, and so the complexity of the algorithm is $O(A^N)$. This number will quickly become too large to be feasible, as in our experience there are rarely less than 10 actions available at all times (and often many more, as actions such as movements are often granular) and to ensure good results the agent should plan as far ahead as possible. Depending on the average length of an action, an NPC is expected to plan at least 10 to 20 actions ahead, but possibly even more.

This results in the fact that backtracking is not feasible in practice.

B. Heuristic optimizations

Optimizations may be chosen so that the algorithm is less wasteful, i.e., some inferior plans are not explored at all. This approach is known as a heuristic search [9], and can reduce the total number of plans that are considered by a large factor, thereby yielding faster run-times. This is achieved by taking into account the resource constraints of the possible actions [10], in order to steer the planner towards plans that maintain desirably high levels of certain important resources, or which increase a resource towards a given goal. For example, a plan that reduces the `health` of an NPC to zero can safely be ignored, as it is deadly and thus of no interest.

Such a planning algorithm is essentially a path-finder. It explores a large graph in which the nodes represent all the valid states of the game world, and the actions represent transitions from one of the valid states into another one. The new state of the game world will be forward in time, and as a result of the actions used it will have different resources, entities, and available actions. Consequently, graph exploration techniques can be considered as of interest to planning.

A graph containing all reachable game worlds is too large to warrant straightforward exploration. Instead of using an algorithm such as Dijkstra's minimum pathfinding algorithm, we can use variations such as *iterative deepening depth-first search* (IDDFS) [11]. IDDFS is a search strategy where each iteration increases the depth of exploration until the shallowest goal state is reached. IDDFS is a form of breadth-first search, but it uses less memory. In particular, we use IDA* [12], which is a variation of the A* algorithm that uses iterative deepening to keep memory usage lower than A*.

IDA* is an informed search, because it expands the reached nodes according to some heuristic. Since a node represents a possible state of the game world, the *most desirable* state is picked. The desirability of a game world is a heuristic which may vary depending on the specific scenario; in many cases, an ordering of the NPC resources is defined so that the most important resources are kept as high as possible. This heuristic drive has two side-effects: *(i)* it finds plans that not only satisfy the goal, but which do so by optimizing some resources; and *(ii)* it may speed up the search process significantly by focusing exclusively on promising states of the game world, and ignoring reachable but pointless states. Moreover, by trimming the set of reachable game worlds so that it does not exceed a maximum size, we obtain a further speed up: plans that reach inferior intermediate states are discarded right away. Reducing the size of the set of explorable game worlds represents a tradeoff between the throughness of the search and its speed. The more we reduce the size of this set, the faster the algorithm becomes. Unfortunately, an excessive reduction in size poses dangers: in particular, we may remove (bad) intermediate states that must be considered by the planner in order to find a good solution. Thus some care is in order when applying such a heuristic. Care in this case means testing and experimentation to ensure that good solutions can still be found after the trims.

The resulting complexity of the algorithm now is $O(N \times M \times A)$, where M is the maximum size of the queue.

Algorithm	Complexity	Steps
Naïve	$O(A^{T_{tgt}/T})$	10^{100}
Fast	$O(\frac{T_{tgt}}{T} \times M \times A)$	100000
Layered	$O(L \times \sqrt[L]{\frac{T_{tgt}}{T}}^2 \times M \times A)$	15000

TABLE I. STEPS PER ALGORITHM FOR $M = 100$, $A = 10$, $T_{tgt}/T = 100$, $L = 3$

C. Layered planning

The optimizations discussed above can partially mitigate the slowness of the algorithm, but they do not change the fact that a large number of plans must be considered before finding the best one. Moreover, by having the maximum number of steps as either an exponent or a multiplicative factor, increasing the number of actions for the searched plan (the value of N) steeply increases the computation time required. Unfortunately, having a long plan is desirable, because the longer the plans, the more effective and capable of long-term reasoning the NPCs become.

To reduce the complexity of the algorithm we have adopted a hierarchical system similar to [13]. This hierarchical system uses multiple planners nested inside one another. The highest level planners plan actions that span a long time each and which involve large changes in resources. As lower level planners are activated, their plans are created so that they respect the constraints given by the higher layers. The actions of a layer are more concrete, take less time, and generally involve smaller resource exchanges than the actions of the layers in higher levels. Considering three layers, examples of actions for the example seen in Section III could be: *(i)* `complete quest X`, `acquire sword`, `fight ogre` for the highest layer; *(ii)* `use item X` for the middle level layer; and `move to X` for the lowest level layer. Of course, depending on the problem domain, different layers and actions may need to be defined.

The layered planning system creates plans by activating a lower level layer to specify the actions that must be taken in order to complete a higher level action. The final planning system works by invoking the fast planning function multiple times, each time with a different set of available actions (those identified by the current layer) and with a different goal (the one specified by the current higher layer action). Suppose that L is the number of layers, that $\texttt{find\_plan\_fast}_L$ plans for the highest level layer (that is, using the global goal of the agent), and that $\texttt{find\_plan\_fast}_{l,action}$ plans for the l^{th} layer and uses $action$ as a goal. The algorithm then is:

```
find_plan_layered(world,agent) =
  plan = find_plan_fast$_L$(world,agent)
  for l = L-1 downto 0 do
    new_plan $\leftarrow$ []
    for (world,agent,action) in plan
      new_plan $\leftarrow$ find_plan_fast$_{l,action}$(world,agent)
           :: new_plan
    plan $\leftarrow$ new_plan
  return plan
```

Thanks to the use of layers, our planner can now plan for much longer periods of time, while the planning requires a significantly decreased computational load in comparison. The complexity of the above algorithm is given by invoking the fast planning algorithm once for every layer, that is L times, and for each action found by each layer, that is N times: $O(L \times N^2 \times M \times A)$. The advantage is that the final plan, entirely composed of lowest level actions, will now be much longer than before, that is N^L actions.

To understand the advantage of this new formulation, suppose that each of the lowest layer actions takes T seconds *of simulated time* to complete. Also suppose that our target planned time is T_{tgt} seconds *of simulated time*. The total number of actions required for such a plan are T_{tgt}/T. The number of steps that each algorithm takes are summarized in Table I, and show that the layered planner is the most effective by a large margin.

A layered approach offers a further reduction in computational load, which applies to non-deterministic simulations where planning may fail due to unforeseen circumstances. Instead of computing all the steps needed by all the layers, we can employ a lazy evaluation [14] approach where plans are refined only as the previous actions get completed. This way, if a plan fails halfway, we can skip computing the lower level plans for the successive actions which are discarded anyway.

D. Memory of old plans

In an effort to improve the performance and believability of our NPCs, in our approach we provided them with a small memory of old plans, so that a set of useful plans that were found and used in the past may be stored for each layer. This process is known as *memoization* or *tabling* [15], and has been used in conjunction with planners such as [16]. Each plan contains not only a sequence of actions, but also the circumstances it was activated in (time, location, NPC resources). If the current circumstances match the original circumstances that the plan was activated in, then the plan is considered a candidate for re-activation. The benefits for the plan are then computed from the current configuration of the game world, and if the result achieves the current goal then the plan is used. When a plan is used, then its score is increased according to some criterion, for example the amount of time elapsed since last reuse. When a new plan is formulated, then it is memorized, but if there are too many plans already in memory then the lowest scoring plans are removed since they have been used the least.

The memory system can be described by the following algorithm:

```
recall_plan(world,agent) =
  for plan in agent.memory do
    if similar(plan.world,world) $\wedge$
       reaches_goal(plan) then
       increase_score(plan)
       return plan
  return null
```

A hazard of the memory system is that the circumstances in which an agent builds up his memory of past plans may change suddenly, thereby obsoleting all memorized plans right away. For example, an NPC may change jobs, move to another city, etc. We employ a simple heuristic to address this problem: when an agent cannot recall a useful plan for a long period of time, for example a few days, and memory is full, then we simply erase its memory. This amounts to resetting the search of the optimal memory away from a bad local solution. While this solution is not especially refined and could be improved significantly (though doing so may amount to a whole separate paper), it does have the advantages of simplicity and high performance.

E. NPC personalization

Our framework allows the differentiation of NPCs, so that different NPCs may make different decisions depending not only on their current circumstances, but also on their preferences and their internal constitution.

By modifying the goals of NPCs, we let them make decisions according to a personality which makes them favour certain (valid) plans over other (still valid) plans. The planner will still have as its implicit goal the survival of the agent, that is plans that load to the death of the agent will be ignored, no matter how appealing to the agent personality they may be. Hoewever, where there multiple plans that reach the goal exist, agents with different personalities will choose different plans. Also, the priority given by the heuristic that selects the next action to expand may be steered by NPC preferences. For example, and NPC that favors magic may try to find plans that solve a battle with magic fighting actions rather than using physical tools.

V. Execution of plans

In this section we discuss how, after the planning phase, plans are actually put into action. While planning, NPC's read and modify temporary copies of the game world state. The game world itself is only modified through the execution of the final, selected plan.

When planning in real-time, special care must be taken in order to ensure that the executed plans remain sensible throughout their execution. This cannot be guaranteed while computing a plan, since we cannot predict with total accuracy how lower level plans will achieve goals, how other agents will act, and how random events in the game world will affect the state. This requires plan executions to be interrupted in order to plan again whenever appropriate. Moreover, there can be benefit in searching for new plans while waiting for the execution of the current plan to be completed. In particular, this minimizes the time an NPC is idle while waiting for its planner to complete.

a) Simple planning and execution: The simplest scheme just alternates planning and execution. Planning happens in a single tick of the simulation, and as such is finished instantaneously with respect to the simulation time. After planning, the execution phase is triggered. Executing a plan according to this scheme is straightforward, and it just executes the plan actions in order.

b) Compensating for plan errors: The more planning is long-term or layered, the more likely it is that the situation expected while planning will not occur as anticipated. Even more so if the game world is non-deterministic and may change according to randomness or other unpredictable factors. Planning errors may be benign (a plan is yielding higher benefits than expected) or malign (a plan is costing more than anticipated); negative effects may add up to the point that an NPC following such a plan blindly may find himself with resources so low as to die. This is undesirable, but we can correct the situation with additional checks to determine if an action can be performed safely before actually starting it:

```
execute_plan(world,self,plan) =
  for action in plan.Actions do
    if action.cost < agent.resources then
      execute_action(world,self,action)
    else
      return
```

Additionally, while planning we *simulated* the actions, and so we can keep track inside a plan of the expected results of each action in terms of the configuration of the game world and the NPC. We can then further check so that an NPC keeps running a plan only under the original assumptions:

```
execute_plan(world,self,plan) =
  for action in plan.Actions do
    if action.cost < agent.resources ∧
       matched(action.expectations,world,agent) then
      execute_action(world,self,action)
    else
      return
```

With such an implementation the agents will not blindly follow a previous plan which might now be dangerous due to changes in the world state since planning. Moreover, while long plans suffer from approximation errors caused by the layered planners and lack of determinism of the game world, with this safeguard NPCs can safely try and plan ahead for even multiple days or weeks of simulation time: replanning will be triggered before an execution is completed if needed.

c) Planning in real time: Up until now we have assumed that planning is virtually instantaneous, done in a single tick of the game's clock. However, as we add more and more planning NPCs to the simulation or the simulation grows more complex and thus computationally expensive, the cost of planning becomes too high for a single tick. When this happens, we must split the computation of plans across multiple ticks. NPCs will then factor in the time required for completing the current action (which is also the time dedicated to planning) into the plan itself. Suppose that the amount of time that NPCs take for planning is T_{plan}, then the planning algorithm will simulate that T_{plan} seconds have elapsed, and then will try to plan across multiple ticks of the simulation, but for no more that T_{plan} simulated seconds. If the planner cannot find a plan in T_{plan} seconds, then T_{plan} is increased by a factor K so that the next planning phase will take longer [1]:

```
while true do
  world',self' = simulate_after(world,self,$T_{plan}$)
  plan = find_plan(world',self') | wait($T_{plan}$)
  if plan $\neq$ null then
    $T_{plan} \leftarrow T_{plan}$ / K
```

[1]We run the planner concurrently to a simple waiting routine; we denote concurrent execution with the pipe operator. A | B denotes that A and B are started concurrently, and the first one thread that terminates stops the other.

```
    execute_plan(world,self,plan)
  else
    $T_{plan} \leftarrow T_{plan} \times$ K
```

The function that finds a plan will now suspend its execution after each iteration of its main loop, so that it will be resumed at the next tick of the simulation:

```
find_plan_fast(world,agent) =
  Q = {(world,agent)}
  C = 0
  do
    ... (*update Q of explored plans*)
    suspend()
  while Q $\neq \emptyset \wedge$ Steps < N
  return null
```

It is possible to define even more aggressive schemes of suspension, for example once every few simulated actions, depending on the simulation.

d) Planning while executing: Plans may be computed during execution. Doing so allows us to take even more than the allotted time of T_{plan}, because if we plan during an action that is sufficiently long, then we may take all the time that such action will take for planning. Planning during an action requires that the plan is formulated starting from the point that the action has finished executing [2]:

```
execute_plan(world,self,plan) =
  for action in plan.Actions do
    if is_last(action) then
      world',self' = simulate_after(world,self,
          action)
      return find_plan(world',self') &
          execute_action(world,self,action)
    else
      execute_action(world,self,action)
```

Executing a plan may now return a new plan. In such cases we will not trigger a new planning phase, and instead we will just execute the returned plan:

```
while true do
  plan = find_plan(world,self)
  do
    plan $\leftarrow$ execute_plan(world,self,plan)
  while plan $\neq$ null
```

VI. EVALUATION

In this section we describe our case study and its use for evaluating our planning system. We built a virtual city populated by NPCs and filled with roads, buildings, parks, and usable items inside parks and buildings. Example buildings can be homes, factories, restaurants, gyms, etc. Example entities inside the buildings can be a bed, a kitchen, a television, an exercise machine, each with its own actions available for using them. Each NPC can only access a certain home and work place, but all NPCs can use public buildings and means of transportation. The layout of the city, depicted in Figure 1, is generated procedurally. This city is built using the Casanova game development framework [17]. The city is populated with multiple NPC's with behaviours computed with the technique presented in the previous sections. The challenge for NPC's as presented by the city lies in the fact that time and money are heavily constrained. Moreover, activities are only available when the corresponding buildings are open; for example, grocery shops close in the late afternoon, while offices close a bit earlier. Good behaviours must balance the need for respecting opening times, money to pay for bills, all in addition to maintaining good statistics of the NPC.

The evaluation focuses on assessing three aspects: *(i)* the ability of NPC's to survive through planning in an unknown environment; *(ii)* the high performance of our technique; and *(iii)* the ability to customize NPC's so that they build personalized behaviours.

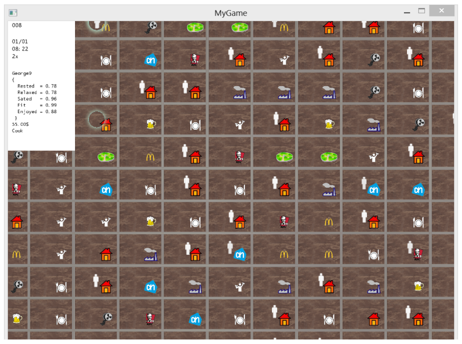
Behaviors are adaptive, in that the NPCs have no hard-coded knowledge of the city layout (which is randomly generated) or the available actions (which depend on the city layout). For example, we added night-time jobs and activities such as pubs and movie theaters that open late in the evening, and our agents immediately started using them in their plans. Similarly, increasing or decreasing the costs or benefits of an action immediately causes different behaviours from the agents. Moreover, the changes can be effectuated both statically or dynamically at run-time: the planner will adapt anyway. The different layers ensure that short and long term goals are achieved. The fundamental long-term goal that we tested was survival; after running the simulation for whole simulated years, we stopped it: the NPCs showed to be capable of generating survival behaviors for potentially indefinite periods of time. Medium- and short- term goals are also achieved.

Figure 2 plots the results of planned behaviours. The different resources of an NPC are grouped into: *vital*, such as health, hunger, or tiredness; and *non-vital*, such as fitness, fun, or social interaction. Two agents start from the same configuration, one with the goal of earning 500 units of money, the other without specific goals besides survival. Both agents find plans where the vital resources are highly optimized: the figure shows that vital resources do not go below 60%; non-vital resources are optimized as well, but their lesser importance is reflected by the fact that they never go below 40%. The agent with the goal of earning money also reaches his goal, meaning that the non-vital resource of money is optimized more aggressively than all the others.

The planner performance is sufficiently high as to allow the simulation to run in real time on very large cities with large numbers of NPCs. Specifically, the simulation runs so fast that we added a speedup factor to force the planner to run more often. A city with one hundred agents, a surface of thirty-six square kilometers, and more than a hundred buildings that the NPCs can use runs at an interactive frame rate even with a speedup of ten simulated minutes per actual second. For the simulation to run at less than an interactive framerate we have to multiply the number of agents threefold, and use a city surface of eighty-one square kilometers. The performance benchmarks are listed in Table II.

Agents can also be customized so that their behaviours reflect preferences. Figure 3 shows the resources of two agents

[2]we denote by (&) the operator which runs two threads in parallel until both complete



```
008

01/01
08: 22
2x

George9
{
  Rested  = 0. 78
  Relaxed = 0. 78
  Sated   = 0. 96
  Fit     = 0. 99
  Enjoyed = 0. 88
 }
55. 00$
Cook
```

Fig. 1. The first virtual city

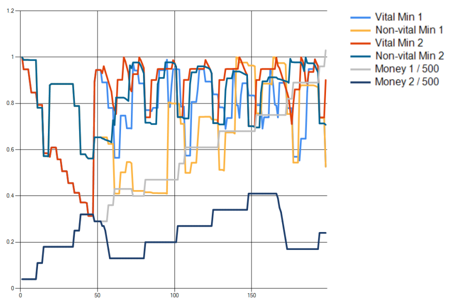


Fig. 2. The resources of two agents: first with the goal of survival plus earning 500 units of money, second with just the goal of survival

who start from the same configuration, but one of them favours fitness-increasing actions. Once again, the two agents find different, but valid, plans. One agent maintains his fitness level higher than 95% at all times, while the other's fitness level occasionally reaches lower values.

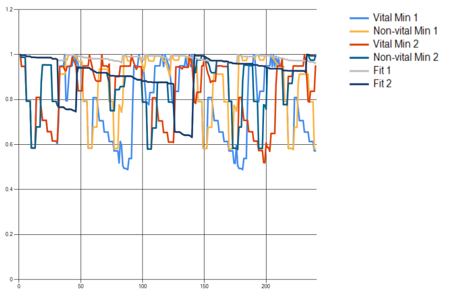


Fig. 3. The resources of two agents: first is a fitness lover, second is a regular NPC

Num agents	City size (sq. km)	FPS	Speedup (min. / sec.)
100	6×6	30	10
150	6×6	30	5
200	6×6	30	5
250	7.5×7.5	30	5
300	9×9	15	2

TABLE II. PLANNER PERFORMANCE

VII. RELATED WORK

There are multiple techniques used in AI for games; these techniques range from planning systems to heuristics, and they all offer different blends of computational requirements, expressive power, etc.

e) Traditional planning systems: LGOAP borrows and heavily extends the central concepts of previous work such as STRIPS and GOAP [3], [7]. The main difference between our approach and those is the use of layering to provide a scalable mechanism to cope with the explosion in the search tree that occurs when searching for long sequences of actions. By using layers we effectively tame the complexity of the algorithm, without giving up the original power of the techniques.

When compared with STRIPS and traditional planners in particular, the most important difference becomes evident: the focus on real-time planning instead of offline decision-making. When compared with GOAP in particular, another important difference lies in how our technique deals with very long plans, whereas GOAP only concerns itself with short plans that only span the immediate future of the NPC.

f) Planning through genetic algorithms: Genetic algorithms (GAs) are a powerful heuristic to deal with searches in combinatory spaces. GAs have been used successfully in a variety of fields, planning included [18]. Through GAs it is possible to generate day-long plans for a very large number of agents to drive a virtual city simulation.

The technique is powerful and promising, but it features a shortcoming that our layered planning system does not have: it requires a carefully crafted utility function. Plans generated this way may reflect implicit biases of the utility function, which may need to be entirely reformulated in case of modifications to the game scenario. GAs for planning are thus less flexible than our technique, and more prone to fall into local minima of the search space.

g) Other hierarchical approaches: Hierarchical approaches such as hierarchical task networks (HTN) [19] or hierarchical state machines (HSM) [20] are used to approximate long-term reasoning in simulations by featuring different layers that represent awareness of consequences and intentions about the future of an NPC.

These approaches are powerful, in that they yield behavior which on occasion is comparable with that of a planner, and at a fraction of the computational cost. Additionally, such techniques involve the *design* of a task network or a state machine: this allows game designers a higher degree of control over the behavior of the NPCs. Unfortunately the resulting behaviors do not adapt to the environment, and the NPCs will suffer from many of the shortcomings described in Section I, namely they will act as if unaware of their surroundings and will not play the game according to the same rules as the player. As such these techniques are simply hand-crafted, heuristic approximations of a planning system with far less flexibility, adaptability, and effectiveness.

VIII. CONCLUSIONS

In this paper we discussed the use of a layered planning technique in order to create non-playing-characters (NPCs) that can actually *live* in a game world rather than simply using rote pre-programmed actions to react to external stimuli. We argue that using layered planning to allow an NPC to achieve its own survival in the game world greatly increases the believability of the game world, and gives additional depth to the game.

We used a planner inspired from forward-chaining constraint logic programming, complemented with an aggressive heuristic for steering the planner and pruning of less promising partial plans, and also using memoization to recycle old but effective plans. Using such a system in real-time poses additional challenges, which we address by tracking expectations in order to make sure that the assumptions made by the planner still hold. Also, the planner takes into account the time it will take for planning, so that the plan does not automatically start late.

We observe that our technique offers multiple positive features, as outlined in Section VI. Our layered planner is *fast* and **effective**, in that it guides hundreds of NPCs to survival in challenging, real-time scenarios for long periods of time. Most benchmarks and tests were actually stopped after various months of simulated time, at which point it is reasonable to say that the NPC could have survived indefinitely. Our planner is **adaptive**, given that the actions it plans and the environment in which it survives are not hard-coded. The behaviors of our NPCs reach short-, medium-, and long term *goals*, such as prolonged survival or obtaining some specific value of some resource. Finally, the behaviors generated by our technique are *customizable* in order to simulate NPCs with different preferences and personalities; this does not impede the achievement of goals but instead chooses among multiple valid plans according to the specific preferences of the NPC.

In conclusion, using planning in real-time games offers different challenges to traditional planning for board-games. When those challenges are solved though, planning proves to be a powerful and reliable technique that enables the generation of high quality NPC behaviours.

IX. Future work

An important aspect of the approach presented in the paper is that it offers numerous opportunities for extension, from structural improvements such as more learning capabilities to further testing in real games and simulations.

At the moment, our system requires the layers to be designed by hand. By grouping all the available actions, sorting them by their impact on the NPC resources (higher level actions would be those with high impact, and vice versa), and estimating the goals that they give to lower level layers, this design effort could be removed. Similarly, using old plans of a layer as actions for the higher level planners, we could obtain a mechanism to allow NPCs to "learn" the layers.

An additional aspect that could be explored is that of social interactions through cooperative planning between NPCs. By planning to meet or cooperate at given times in the higher level planners, it would be possible to define simple social behaviours such as dates, or even more complex ones such as team combat. Social interaction may allow agents to communicate and even share plans and other useful information. NPCs would then learn faster through such interactions.

Finally, estimating the precise effect of actions in unpredictable scenarios instead of hard-coding them, would make the planner more accurate. This could be accomplished with statistical reasoning, such as Bayesian methods [21], neural networks [22], clustering [23], etc.

References

[1] Bioware, "Mass effect 3," http://masseffect.bioware.com/.

[2] Lionhead, "Fable 3," http://lionhead.com/fable-3/.

[3] R. E. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," *Artificial Intelligence*, vol. 2, no. 3-4, pp. 189–208, 1971.

[4] S. J. J. Smith, D. Nau, and T. Throop, "Computer bridge: A big win for ai planning," 1998.

[5] M. Booth, "Left for dead: Ai director," http://www.valvesoftware.com/publications/2009/ai_systems_of_l4d_mike_booth.pdf.

[6] T. Frühwirth and S. Abdennadher, *Essentials of Constraint Programming*, ser. Cognitive Technologies. Springer, 2003.

[7] J.Orkin, "Three states and a plan: The ai of f.e.a.r." *Proceedings of the Game Developer's Conference (GDC)*, 2006.

[8] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, ser. Prentice Hall Series in Artificial Intelligence. Prentice Hall, 2010.

[9] B. Bonet and H. Geffner, "Planning as heuristic search," *Artificial Intelligence*, vol. 129, pp. 5–33, 2001.

[10] P. Haslum and H. Geffner, "Heuristic planning with time and resources," 2001.

[11] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[12] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artificial Intelligence*, vol. 27, pp. 97–109, 1985.

[13] E. D. Sacerdott, "Planning in a hierarchy of abstraction spaces," in *Proceedings of the 3rd international joint conference on Artificial intelligence*, ser. IJCAI'73. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 412–422.

[14] P. Hudak, "Conception, evolution, and application of functional programming languages," *ACM Comput. Surv.*, vol. 21, no. 3, pp. 359–411, Sep. 1989.

[15] N.-F. Zhou and T. Sato, "Efficient fixpoint computation in linear tabling," in *Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, ser. PPDP '03. New York, NY, USA: ACM, 2003, pp. 275–283.

[16] M. Johnson, "Memoization in constraint logic programming," in *Proc. Intl. Workshop on Principles and Practice of Constraint Programming*, 1993, p. manuscript.

[17] G. Maggiore, P. Spronck, R. Orsini, M. Bugliesi, E. Steffinlongo, and M. Abbadi, "Writing real-time .net games in casanova," in *Entertainment Computing - ICEC 2012*. Springer Berlin Heidelberg, 2012.

[18] D. Charypar and K. Nagel, "Generating complete all-day activity plans with genetic algorithms," *Transportation*, vol. 32, no. 4, pp. 369–397, Jul. 2005.

[19] J. P. Kelly, A. Botea, and S. Koenig, "Offline Planning with Hierarchical Task Networks in Video Games," 2008.

[20] J. Cremer, J. Kearney, and Y. Papelis, "Hcsm: A framework for behavior and scenario control in virtual environments," *ACM Transactions on Modeling and Computer Simulation*, vol. 5, pp. 242–267, 1995.

[21] A. Gelman, J. Carlin, H. Stern, and D. Rubin, *Bayesian Data Analysis*. Chapman & Hall/CRC, 2004.

[22] R. Duda, P. Hart, and D. Stork, *Pattern classification*, ser. Pattern Classification and Scene Analysis: Pattern Classification. Wiley, 2001.

[23] J. A. Hartigan, *Clustering Algorithms*. New York: Wiley, 1975.

[24] U. Technologies, "Unity 3d," http://http://www.unity3d.com/.