Using STRIPS for automated computer game scene generation

Benjamin Schnieders

July 10, 2008

Abstract

To be able to provide harmonious artificial game worlds, game designers have to write huge, unflexible scripts defining NPC behavior. This paper investigates to what extend it is possible to use a STRIPS planner to generate NPC plans of a scene automatically during runtime. An experiment was performed in a simulated game environment, and test runs show the ability of generating and executing plans. It was found that even from simple scene settings, many plots can be generated, in which the agents show intuitive, cooperative and emergent behavior. From the results it can be concluded that STRIPS planners are suitable for planning scenes and that the technique presented is an improvement over commonly used scripting techniques.

1 Introduction

During the last decades, the concept of computer controlled players evolved from simple targets to real characters, with own targets that may make them an ally or an enemy to the human player. Usually, the player follows an overall storyline leading the alter ego through the game world. Most interactions with the game world happen through Non-Player Characters (NPCs); these interactions mark the progress of the story. Because of this, the NPC behavior is very important [2, 5]. Making the NPCs more human, rather than harder to defeat, seems to make the player feel more comfortable [23]. Humanlike behavior without player interaction can be shown through interaction between NPCs, such as showing cooperative behavior of agents. To achieve natural NPC behavior, game designers put effort in creating realistic scenes, usually using scripts that the NPCs execute. Writing these scripts does not only take a lot of time, the concept of scripts also introduces a major issue: they are unflexible, thus the acting of the agents cannot be altered after the game is released [11]. Techniques for automatic scene generation can help saving time while creating the scenes and, since the planning can be done during execution, bring a huge increase of flexibility [9].

The research question investigated in this paper reads as follows: *Can automatically generated plans provide an improvement over common scripting methods?* Requirements here are variety, flexibility, feasibility and whether these plans can create the illusion of being a result of natural behavior.

2 Background

For creating a living world around the player, NPCs play small scenes or provide substories which can be explored. A scene is defined as a small sequence of actions, embedded in the game world to enhance its impression or make a certain statement. These scenes are usually causally independent from the main story, so that decisions made by the player in these scenes do not alter the main storyline. The fact that scenes are encapsulated allows planners to create scenes automatically, without the danger that the the main storyline could be influenced. The increased number of possible substorylines plus the engaging behavior of computer characters gained by intelligent planning is demanded by players [19]. Subsection 2.1 gives a short overview on how the topic is handled in current games, Subsection 2.2 provides information what STRIPS is and how plans are generated. A summary of related articles can be found in Subsection 2.3.

2.1 Current Games

Scripts in modern computer games usually determine the NPC behavior completely, thus leaving no flexibility for reacting on the player or other non-scripted events. Because of this, scripts may fail during execution since prerequisites were changed by the player. Results of failing scripts can be differ from game crashes to leaving the player in a situation where the game cannot be finished anymore, since certain actions in the script could not be done. Even if the result is not that fatal, it can be enough to destroy the illusion of a consistent and realistic game world, like the physically impossible situation the NPC in Figure 1 is in. Since often players try to explore all possibilities the game might provide, failing scripts are very common. In order to avoid malfunctions in scripts the freedom of the player is limited, NPCs are made immortal or areas are made unaccessible by blocking them through invisible walls.

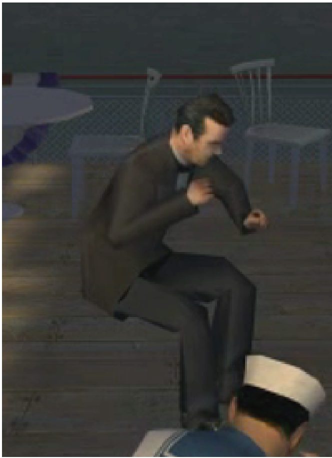

Figure 1: This NPC could not stop the eating action before being placed at another position by a script.

Today, planning agents in computer games use advanced techniques over predefined actions [11]. Scripts and scripted events only create an appearance of intelligence, but the NPCs themselves are not smarter. Predefined behavior can easily be exploited, especially if there is no alternation of scripts used for a certain scene, players can replay that scene and already know what will happen next. Because of these downsides, game developers changed the artificial agent intelligence to planning systems. For example, the F.E.A.R. game-engine uses a modified STRIPS-planner as part of the agent intelligence system [17].

2.2 Planning with STRIPS

STRIPS, the Stanford Research Institute Problem Solver, is a basic planning language [10]. It uses First Order Logic statements to express its current state of the world in boolean relations. STRIPS planners find plans from one situation, given in FOL, to another. All actions have to be defined including preconditions necessary to perform the action and effects the action will have. Solving complex problems within very limited time is a situation that can occur in scenes with multiple agents and many possible actions. The original STRIPS planner is not capable of planning in complex scenarios in realtime, but modern implementations like Graphplan [3] give a significant improvement in speed [14]. Also, Graphplan can distinguish between actions that have to be performed sequentially and those that might be performed in parallel, which is an important feature since STRIPS was primarily not designed for possible parallel execution of actions [8].

2.3 Related Work

Related work on creating more humanlike NPCs can be separated into the topics of automated storytelling and agent planning. Automated storytelling is focussed on creating interesting and consistent plots [6, 7, 12], using forward search techniques in plot space to direct the plot in a certain direction. Planning enhances NPCs on the level of intelligence and consistency in their own behavior while following their goals [16, 17, 22], using symbolic planning algorithms. Combining both, NPCs can follow their own goals and still, due to the designer's decisions while setting up the scene, create meaningful plots [4].

3 Experimental Setup

To answer the research question, an example scene is used, in which agents have goals, so that the planner is able to generate plans. Two teams of agents with competing goals use the planner to generate sequences of actions that are performed. Once a team reaches the goald, the planner is finished. When a planned action can not be performed, a new plan is generated from the current state of the world. Subsection 3.1 describes the implementation of the planning engine, Subsection 3.2 gives details on the setup of the example scene. Subsection 3.3 provides details on the execution of the plan.

3.1 Planner Implementation

When setting up a STRIPS-based planner for computer games scene logic one encounters two main difficulties. These difficulties and possible solutions are described below. First, STRIPS is just defined for a single Operator. This means that there is no competition and no competitive plans. Since total collaboration usually is not intended in scenes of interest, the problem is solved using the concept of teams: Each team has a different goal and uses the planner to achieve it. All members of a team have the same common goal, and try to reach it by following the steps of the plan. Since the original STRIPS planner does not support multiple agents, the notation of actions does not include information on how many agents can perform an action in parallel. Furthermore, the plan created does not define which actions are dependent or can be performed the same time. Graphplan avoids these problems since it checks the preconditions of planned actions to determine whether they may be executed in parallel.

Second, expressing a scene in first order logic is connected with other problems [15]. Numerical data has to be reduced to certain ranges of interest in order to be expressible in FOL. In computer science, information is often inherited from other data. Since there are no conditions in FOL that could be used for such definitions, all information about an object has to be given. This can be implemented in a way that all objects that can be described in FOL extend a common superclass, having a method that returns a FOL representation of the object.

The inherited classes call the parent class (or classes) for their FOL definition, and add their own data before returning. For an illustration see Figure 2.

```
class FOLObject
{
public:
    FOLObject();
    virtual string getFOLDescription();

protected:
    string name;
};

class UseableObject : public FOLObject
{
public:
    UseableObject();
    string getFOLDescription();

protected:
    string atPos;
};

string FOLObject::getFOLDescription()
{
    return "("+name+")";
}

string UseableObject::getFOLDescription()
{
    return FOLObject::getFOLDescription() +
        "(object " + name + ")" +
        "(at " + name + atPos +")";
}
```

Figure 2: An example how inheritance can be expressed in FOL.

3.2 Scene Setup

For testing and evaluation of the planner, an example scene was generated, visually presented in Figure 3. This scene takes place in different regions, separated by water, which the agents can only cross by using bridges. The agents' goal is to have an object *gold* in the end. To achieve this goal the agents have to move to and pick up a key which is then used to open a chest containing the gold. The scene expressed in First Order Logic can be found in Appendix A, Figure 10.

Important information about each scene is knowledge about which agents are participating in the scene and which agents are cooperating with other agents. This information can be given through the concept of teams. All agents in a team try to reach the same goal and will collaborate to achieve it. Other teams, even if their goals define them as competitors, are not necessarily enemies.

Figure 4 shows an example scene file. The first line denotes the number of teams in the scene. The following two lines state the operations file for each team, all actions the agents of this team can execute are listed here, with preconditions and effects.

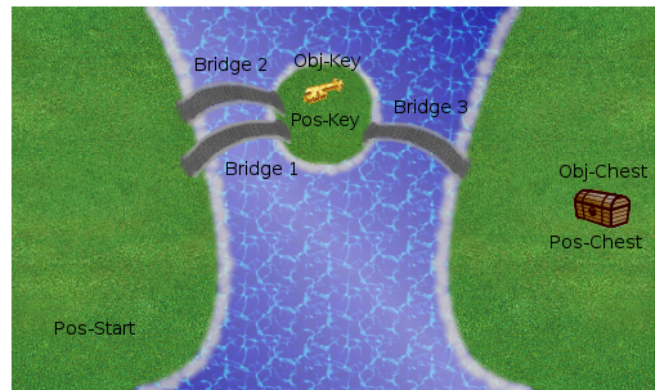

Figure 3: A map of the example scene.

```
1  2
2  team1.ops team1.facts (have_Hero1_obj-gold)
3  team2.ops team2.facts (have_Badguy1_obj-gold)
4  4
5  1 Hero1 pos-startteam1
6  1 Hero2 pos-chest
7  2 Badguy1 pos-startteam1
8  2 Badguy2 pos-chest
```

Figure 4: An example of a scene file

The operations are expressed as shown in Figure 5. First, the name of the operator, so the name of the function, is given, followed by a list of parameters. These are the variables the planner will try to match. The list of preconditions has to match the state of the world in order to make the execution of the function possible. The effects, finally, are added to the state of the world, unless the keyword *del* is specified, which is used to delete information.[1] In the example a function to pick up an object is described: The preconditions dictate that the first parameter is of type *agent*, the second of type *object*, and that both are at the same position *pos*, which is variable and thus part of the parameters.

In the effects section a relation *have* on the agent and object is set true, and the relation specifying where the object was before is deleted. For the example scene, 7 simple actions are used, shortly introduced in Table 1. A complete operations file can be found in Appendix A, Figure 11.

After the operations file, the facts file is specified, providing a representation of the scene in First Order Logic from the perspective of the current team. In this file, first all entities are declared, see lines 1-6 in Figure

[1] As in STRIPS relations can only be defined to be true, they are deleting instead of setting them to false [10].

```
(operator
 PICKUP-OBJECT
 (params (<ag>) (<obj>) (<pos>))
 (preconds
       (agent <ag>) (object <obj>)
       (at <ag> <pos>) (at <obj> <pos>)
 )
 (effect
       (have <ag> <obj>)
       (del at <obj> <pos>)
 )
)
```

Figure 5: A function to let an agent pick up an object

Action	Used for
GOTO	moving within one region to a different position
CROSS-BRIDGE	crossing a bridge
OPEN-CHEST	open a chest with a key and take an object out of it
PICKUP-OBJECT	picking up an object lying on the ground
DROP-OBJECT	dropping an object from the inventory
ATTACK	attacking an enemy agent, resulting in the enemies' death
LOOT	obtaining an object from the inventory of a killed agent

Table 1: The actions used in the example scene

6. The planner can only match variables that have been declared before. After the declarations, all existing relations between objects are expressed and lastly the goal is specified, line 23 in Figure 6.

The next line in the scene file denotes the number of agents in the scene, which are listed subsequently, each preceeded by the number of the team the agent is in and followed by the position the agent is at in the beginning. Since each agent in a team has the same goal, all allied agents follow the same plan. The designer can choose in which situation the scene execution should stop. A candidate that can always occur is that no team is able to fulfil the plan, in this case the scene should always be stopped since no plan is generated any more. The designer might want to stop some scene once the first team reaches its goal, others may run until a specific team wins, or even all teams win. Many combinations are possible, the designer has to choose what fits best for the scene. Sometimes other events can be used for scene termination, such as timers, counters or specific actions that happened in the plan.

3.3 Planning and Plan Execution

The STRIPS planner creates a chain of actions to connect the scene settings and the goal situation. Graphplan creates additional information about which actions have

```
(Hero1)
(pos-key)
(pos-chest)
(obj-key)
(obj-gold)
(obj-chest)

(preconds
        (agent Hero1)
        (at Hero1 pos-key)

        (object obj-key)
        (keyfor obj-key obj-chest)
        (at obj-key pos-key)

        (object obj-gold)
        (at obj-gold obj-chest)

        (object obj-chest)
        (contains obj-chest obj-gold)
        (at obj-chest pos-chest)
)
(effects (have Hero1 obj-gold))
```

Figure 6: A sample facts file for a simple scene

to be executed one-by-one and which are independent of the order, or may be executed in parallel by multiple agents. Using this information the scene plan can be decomposed to sequential steps and raw actions, where one step holds all actions that can be executed in parallel. The Planner finally tries to schedule all agents in the team to the actions of the current step, in order to guarantee a fast plan execution. Implementing threads from the Qthreads library [21] allows the program to do multiple actions at the same time. This way the actions can be separated from the planning procedures and the routines that collect the state of the world in FOL representation.

When the team thread schedules the execution of the next action on the list multiple things have to be checked:

- Does the agent have a corresponding method?
- Are all preconditions met?
- Is another agent currently using an object which is a parameter here?

If the last last case is true, the agent will have to wait until the object is available again. As soon as an agent has succeeded in performing an action, the action marked as done. Once all actions of the current step are done, the next step is processed, until there are no more steps. When an action cannot be done since the preconditions are not met, although the action was scheduled by the planner, it means the planner is out of synch with the game world. In this case a replan is scheduled, which means the current team thread is stopped and the current state of the world is written to a file the planner is

fed with. Once the new plan exists, the thread is started again and agents go back to work. Whenever an action is finished the plans of all other teams have to be checked whether they are still synchronized with the world data, since performing actions change the state of the world.

4 Results

The planning and plan execution program was tested on the example scene and other simple settings. Running the planner even on these simple scenes as described before and given in Appendix A generates interesting plots. This section shows the results, beginning with the variety of plans that can be generated, in Subsection 4.1. Subsections 4.2 and 4.3 present cooperative and competitive behavior. Finally, Subsection 4.4 provides a summary of the results.

4.1 The variety of generated plans

Considering a plot different when the sequence of "important" actions, this is actions except moving, differ, four distinct plots can be generated from the example scene. These are:

1. Pick up the key, open the chest
2. Pick up the key, fight for the chest, open the chest
3. Fight for the key, pick up the chest, open the chest
4. Fight for the key, fight for the chest, open the chest

These are the four independent and different ways on how to win the scene. Each team has the possibility to reach the goal on every way. The more actions exist, the more possibilities arise for the agents to reach their goal, and thus more plots can be generated. More agents and more teams also contribute to create more plots. When the actions that are performed stay the same, the dramatic value is equal, unless the player has a certain relation to the agents appearing.

Naturally, the plan can not always be performed in the way it was intended. A lot of restrictions and effects that are not given in FOL decide on the further outcome of the scene. It might be that an agent from team 1 is attacking an agent of team 2 in order to loot an object from him afterwards, but the attacking agent dies in the attempt because the enemy is too strong. For the planner, these are considered nondeterministic effects, since they cannot be predicted. Also, plans of other teams are unknown, and thus their effects as well. Another source of nondeterministic effects is the human player. These nondeterministic effects will lead to more complex plots, where replanning is often needed. Especially on more complex scenes the influence of nonpredictability rises, the more actions have to be performed the more actions can fail during execution.

Let the complexity of a scene be defined by the number of actions it takes to finish the scene. Taking these nondeterministical effects into account, the number of possible plots is growing exponentially with the number of operations and the number of objects/agents used. This means, the more complex scenes are given, exponentially more possible storylines are provided. The real number of possible storylines is not growing as fast since not all nodes in the plan are reachable and others may not be considered worth exploring by the planner, still the increase is exponential. The increase in workload for the planner due to replanning grows linear with the complexity of the scene, and quadratic with the number of teams, since each team has to replan when another team is changing the state of the world. Since most planners support caching of subplans, replanning is in average much faster than creating a complete plan, especially when the new state of the world is quite similar to the former one.

Running the planner 20 times on the example scene yielded 16 different plots, of which five can be found in Appendix B, Figure 12 to 16. These are, obviously, mostly differing in the order of (or the positions where) the actions from the four basic plots happen.

4.2 Cooperative Behavior

Graphplan prefers plans with actions that can be executed in parallel over those without, since it tries to find the plan with the least number of parallel steps. As a result, since parallel actions are executed by different agents, (if available) one can see agents harmonically cooperating and sharing workload. Figure 7 shows the plan for the example scene with just one team, and one agent. The plan consists of eight steps that have to be executed consecutively, leading the agent first to the key and then to the chest, which is opened.

```
1 GOTO Hero1 from pos-startteam1 to pos-bridge1end1
2 CROSS-BRIDGE Hero1 bridge1
3 GOTO Hero1 from pos-bridge1end2 to pos-key_reg-island
4 PICKUP-OBJECT Hero1 obj-key1
5 GOTO Hero1 from pos-key to pos-bridge3end1
6 CROSS-BRIDGE Hero1 bridge3
7 GOTO Hero1 from pos-bridge3end2 to pos-chest
8 OPEN-CHEST Hero1 obj-chest1 receiving obj-gold
```

Figure 7: The example scene with one agent, taking 8 steps

Creating a plan for the same scene with 2 agents gives a different result. Although the total length of the plan is now 11 actions, it just consists of 6 steps, since most actions can be done in parallel. Looking at the actions in more detail reveals that the agents perform a very intuitive and thus humanlike worksharing: Since

agent *Hero2* starts at the position the chest is located, it picks up the object and brings it to agent *Hero1*, which is collecting the key in the meantime. Both agents meet in the middle, and agent *Hero2* drops the chest so that it can be opened by agent *Hero1*. Whenever possible, Graphplan will parallelize work, and with it stimulate cooperative behavior.

```
1 GOTO Hero1 from pos-startteam1 to pos-bridge1end1
1 PICKUP-OBJECT Hero2 obj-chest1
2 CROSS-BRIDGE Hero1 bridge1
2 GOTO Hero2 from pos-chest to pos-bridge3end2
3 CROSS-BRIDGE Hero2 from bridge3
3 GOTO Hero1 from pos-bridge1end2 to pos-key
4 GOTO Hero2 from pos-bridge3end1 to pos-bridge1end2
4 PICKUP-OBJECT Hero1 obj-key1
5 GOTO Hero1 from pos-key to pos-bridge1end2
5 DROP-OBJECT Hero2 obj-chest1
6 OPEN-CHEST Hero1 obj-chest1 receiving obj-gold
```

Figure 8: The example scene with two cooperating agents, finished after 6 steps

4.3 Competitive Behavior

If multiple teams follow a goal that can only be reached by one of them, competitive behavior emerges.

Unlike the cooperative scene setup, two teams of agents want to have one object gold in their inventory in this scene. In a real game environment the speed of the agent would determine which agent reaches the key first, in the simulation it is decided randomly by the CPU which thread is allowed to perform the action first. If an agent of one team has the key, the updated plan of the other team will include a method to retrieve the key, since it can not be just picked up any more. The example operations list just provides one sequence of actions to get an object from the inventory of an enemy (obviously many more can be thought of immediately), this is to kill the enemy and then loot it for the wished object. These two actions are now part of the plan the agents will follow. The same happens if one team has picked up the chest or even has the gold already. In the example, the scene was stopped after the first team reached the goal conditions so the gold was never looted.

In Figure 9 one can see both cooperation and competition. The agents within one team both move to a point close to the middle, and in line 13 Hero2 is attacked by Badguy2. This does not directly contribute to the goal, as Badguy1 is the leading character in the team, but, since Badguy1 is currently busy, saves some steps namely those it would Badguy1 take to kill and loot Hero2.

```
Badguy1 moving to position Bridge2Left.                     1
Hero2 picking up object obj-chest1.                         2
Badguy1 crossed Bridge2 and is now at Bridge2Right.         3
Hero1 crossed a Bridge1 and is now at Bridge1Right.         4
Hero1 moving to position pos-key.                           5
Badguy1 moving to position pos-key.                         6
Hero2 moving to position Bridge3Right.                      7
Hero1 picking up object obj-key1.                           8
Badguy2 moving to position Bridge3Right.                    9
Badguy1 moving to position Bridge3Left.                     10
Hero1 moving to position Bridge3Left.                       11
Hero1 crossed a Bridge3 and is now at Bridge3Right.         12
Badguy2 attacks agent Hero2.                                13
Badguy1 crossed a Bridge3 and is now at Bridge3Right.       14
Badguy2 loots agent Hero2 and gets object obj-chest1.       15
Badguy1 attacks agent Hero1.                                16
Badguy2 dropped object obj-chest1.                          17
Badguy1 loots agent Hero1 and gets object obj-key1.         18
Badguy1 took object obj-gold out of the chest obj-chest1.   19
```

Figure 9: The example scene showing competing agents

4.4 Results Summary

The results show that the requirements for the generated plans are fulfilled. Variety is given, due to nondeterministic effects with exponential increase with the complexity of the scene, flexibility is implemented by the possibility to replan once the state of the world changed. The planner only creates feasible plans, and cooperative and competitive behavior lets the plans that are executed seem very natural.

5 Discussion

By creating the example scene it is shown that scenes can be expressed in First Order Logic. The planner supports multiple agents and generates and plays plans for them. The number of possible plans for scenes is growing with the complexity. Also, there are some limitations of the system, which are to be discussed in Subsection 5.1. Subsection 5.2 deals with advantages of the STRIPS architecture, and Subsection 5.3 provides details about the usefulness of the plan generation.

5.1 Weaknesses of STRIPS

The language limitations of STRIPS are rather strict. All clauses that are not defined are assumed to be false, instead of just being unknown. This limitation might make the creation of scenes in which the agents gather information about their environment during execution complicated, since all unknown information is assumed to be false. A related problem is the lack of negated preconditions for operations. This can be avoided as well by defining every relation twice - once the usual way and once that it means the opposite - but this workaround is unhandy, since all effects have to be rewritten and twice the amount of data is used. The Action Description

Language ADL can, as an extension of STRIPS, deal with both negated literals and unknown variable states [18].

Still, neither STRIPS nor ADL have native support for multiple agents. This problem can be circumvented, as shown in Section 3, but still the planner has to be called for each team separately, and, since the competing plans are not taken into account, more replanning is needed: Each action of competitors is not included in the current plan, so that replanning is needed to by in synch with the world data again. Switching to agent centered programming languages as for example 3APL could solve this problem [13].

Another important issue is that none of the planners have a concept of time it may take to execute an action. A rough estimate may be enough, just that the planner knows that moving to a distant position takes longer than picking up or dropping an object. Since this information is not known to the planner, each action is considered to take equally long. Still a better workload between cooperating agents can be reached without heuristic for duration when after each finished operation a replan is scheduled. This way one agent can do a time-intense task and another may perform multiple short tasks meanwhile. For large-scale scenes this is not an optimal solution, since replanning all the time might be too computationally expensive. Any other numerical value has to be quantized in order to be expressible in first order notation. So far, there are few numerical planners available, being capable of taking all kinds of numerical constraints into account, like O-PLAN [20]. These systems are quite complex and in general too slow for real-time planning.

5.2 Advantages of STRIPS

STRIPS is simple. There are not many errors one can make during setting up a game world for an implementation of the planner. Because of its simplicity the planning process is a lot faster than for more complicated languages. Using intelligent search algorithms the planning speed can be increased, and also make the planner independent of for example the goal order [3]. Although the STRIPS notation is known to be insufficient to represent scenes in the real world due to the amount of noise, this approach works well in computer simulated environments, such as games.

5.3 Applicability

Game designers may want to create scenes without random components, for example in cutscenes. In current games, rendered cutscenes are played without usage of artificial intelligence, important characters made immortal, each step is predetermined. The same can be done with this system, by just saving a plan that was generated before, and disabling the replanning during runtime.

Summarized, the generated plots can make the game more worth playing. Mostly the planner creates humanlike step-by-step plans, but sometimes it can exploit certain holes in the logic, which will certainly amaze players. So in the example scene every *object* can be lifted up and stored in inventory by the agent - this also works for objects it was not originally meant for. The chest, for example, extends the *object*, and any agent is thus able to pick it up, which might look surprising to the player. Sometimes, a player might even learn tactics from watching agents perform scenes.

6 Conclusions

An experiment was performed in which STRIPS was used to model a scene and create plans for teams of agents. The results of the experiment runs show that STRIPS can be used to model scenes, that planner implementations like Graphplan create plans, and that a multithreaded program can execute these plans, yielding a variety of different plots. The system is easy to implement and to maintain. One wonders why it is not broadly used in current games [1]. The system is flexible, plots are made from the original intentions of all teams, but regarding side effects the team may have to use a completely different approach after reaching the goal halfway. The generated plans are always feasible, if this state changes, a new plan is made. A large number of plots can be generated, since not all information is known to the planner, and thus often plans change and create new storylines. Generated scenes played by agents can improve the illusion of a realistic gameworld and humanlike behavior of NPCs, since competitive and collaborative behavior is observed as being very natural. In conclusion, generated plans have proven to be an improvement over common scripting methods.

In future work, the planner can be extended to a full Partial-Order Planner, and enhanced with additional information about the agent, that can be used as heuristics for action selection. However, the basic system presented in this paper already is superior to commonly used techniques. Therefore, when implemented in an existing game engine, it may demonstrate its commercial viability in the very near future.

References

[1] AiGameDev.com (2008). Thursday theory: A New Approach to the Application of Theorem Proving to Problem Solving. http://aigamedev.com/theory/strips-theorem-proving-problem-solving.

[2] Baillie - de Byl, Penny (2004). *Programming Believable Characters for Computer Games*. Charles River Media.

[3] Blum, Avril L. and Furst, Merrik L. (1997). Fast planning through planning graph analysis. *Artificial Intelligence*, Vol. 90, pp. 281–300.

[4] Cavazza, Marc, Charles, Fred, and Mead, Steven J. (2001). Agents interaction in virtual storytelling. Technical report, School of Computing and Mathematics, University of Teesside.

[5] Cavazza, M., Charles, F., and Mead, S. (2002a). Interacting with virtual characters in Interactive Storytelling. Proceeedings of the Autonomous Agents Conf., AAMAS'02. Bologna, Italy, 2002.

[6] Cavazza, Marc, Charles, Fred, and Mead, Steven J. (2002b). Character-based interactive storytelling. *IEEE Intelligent Systems*, Vol. 17, No. 4, pp. 17–24.

[7] Charles, Fred, Lozano, Miguel, Mead, Steven J., Bisquerra, Alicia Fornes, and Cavazza, Marc (2003). Planning formalisms and authoring in interactive storytelling. *1st International Conference an Technologies for Interactive Digital Storytelling and Entertainment, Darmstadt, Germany.*

[8] Cocosco, Chris A. (1998). A review of STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving.

[9] Cutumisu, Maria, Szafron, Duane, Schaeffer, Jonathan, McNaughton, Matthew, Roy, Thomas, Onuczko, Curtis, and Carbonaro, Mike (2006). Generating ambient behaviors in computer role-playing games. *IEEE Journal of Intelligent Systems*, Vol. 21, pp. 19–27.

[10] Fikes, R. E. and Nilsson, N. J. (1971). STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, Vol. 2, pp. 189–208.

[11] Funge, John David (2004). *Artificial Intelligence for Computer Games*. AK Peters, Ltd, Wellesley, MA.

[12] Göbel, Stefan, Spierling, Ulrike, Hoffmann, Anja, Iurgel, Ido, Schneider, Oliver, Dechau, Johanna, and Feix, Axel (eds.) (2004). *Technologies for Interactive Digital Storytelling and Entertainment, Second International Conference, TIDSE 2004, Darmstadt, Germany, June 24-26, 2004, Proceedings*, Vol. 3105 of *Lecture Notes in Computer Science*. Springer.

[13] Hindriks, Koen V., Boer, Frank S. De, Hoek, Wiebe Van der, and Meyer, John-Jules Ch. (1999). Agent Programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, Vol. 2, No. 4, pp. 357–401.

[14] Kambhamapati, Subbarao, Parker, Eric, and Labmbrecht, Eric (1997). Understanding and extending graphplan. *Recent Advances in AI Planning*, Vol. 1348, pp. 260–272, Springer Berlin / Heidelberg.

[15] Orkin, Jeff (2004). Symbolic representation of game world state: Toward real-time planning in games. Technical report, AAAI Challenges in Game AI Workshop.

[16] Orkin, Jeff (2005). Agent architecture considerations for real-time planning in games. *Proceedings of the AIIDE 2005*, pp. 105–110.

[17] Orkin, Jeff (2006). Three States and a Plan: The AI of F.E.A.R. *Game Developers Conference Proceedings 2006*.

[18] Pednault, Edwin P. D. (1989). ADL: exploring the middle ground between STRIPS and the situation calculus. *Proceedings of the first international conference on Principles of knowledge representation and reasoning*, pp. 324–332, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[19] Penelope Drennan, Peta Wyeth, Stephen Viller (2004). Engaging Game Characters: Informing Design with Player Perspectives. *Entertainment Computing*, pp. 355–358.

[20] Russell, Stuart and Norvig, Peter (2003). *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition.

[21] Trolltech (2008). Thread Support in Qt. http://doc.trolltech.com/4.4/threads.html.

[22] Young, R. Michael and Riedl, Mark O. (2005). Integrating plan-based behavior generation with game environments. *ACE '05: Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, pp. 370–370, ACM.

[23] Zubek, Robert and Khoo, Aaron (2000). Making the human care: On building engaging bots. Technical report, Northwestern University Computer Science.

A Example Scene

This appendix provides complete files from the example scene.

Figure 10 shows the facts file for team 1. All agents from team 1 are of the type *agent*, all competing agents of type *eagent*. This is the very beginning of the scene, all agents and objects are at their starting positions. Since it is the facts file of team 1, the goal in line 55 is that of team 1, namely that agent *Hero1* has the gold.

```
1  (Hero1)
2  (Hero2)
3  (Badguy1)
4  (Badguy2)
5  (reg-left)
6  (reg-island)
7  (reg-right)
8  (pos-bridge1end1)
9  (pos-bridge1end2)
10 (pos-bridge2end1)
11 (pos-bridge2end2)
12 (pos-bridge3end1)
13 (pos-bridge3end2)
14 (pos-startteam1)
15 (pos-key)
16 (pos-chest)
17 (obj-key1)
18 (obj-gold)
19 (obj-chest1)
20
21 (preconds
22         (agent Hero1)
23         (agent Hero2)
24         (eagent Badguy1)
25         (eagent Badguy2)
26         (bridge pos-bridge1end1 pos-bridge1end2)
27         (bridge pos-bridge2end1 pos-bridge2end2)
28         (bridge pos-bridge3end1 pos-bridge3end2)
29         (object obj-key1)
30         (object obj-gold)
31         (object obj-chest1)
32
33         (keyfor obj-key1 obj-chest1)
34         (contains obj-chest1 obj-gold)
35
36         (at pos-startteam1 reg-left)
37         (at pos-bridge1end1 reg-left)
38         (at pos-bridge2end1 reg-left)
39         (at pos-bridge1end2 reg-island)
40         (at pos-bridge2end2 reg-island)
41         (at pos-bridge3end1 reg-island)
42         (at pos-key reg-island)
43
44         (at pos-bridge3end2 reg-right)
45         (at pos-chest reg-right)
46         (at obj-chest1 pos-chest)
47         (at obj-key1 pos-key)
48         (at obj-gold obj-chest1)
49
50         (at Hero1 pos-startteam1)
51         (at Hero2 pos-chest)
52         (at Badguy1 pos-startteam1)
53         (at Badguy2 pos-chest)
54 )
55 (effects (have Hero1 obj-gold))
```

Figure 10: The example scene expressed in First Order Logic

The complete operations file for team 1 is given in figure 11. Note that the operation *CROSS-BRIDGE* is given twice, with just a swap in the position parameters for the bridge. Since the planner takes any matching function, this resembles a logical OR, allowing to cross the bridge in both directions.

```
(operator
 GOTO
 (params (<ag>) (<pos1>) (<pos2>) (<reg>))
 (preconds (agent <ag>) (at <ag> <pos1>) (at <pos1> <reg>) (at <pos2> <reg>))
 (effect (del at <ag> <pos1>) (at <ag> <pos2>))
)

(operator
 CROSS-BRIDGE
 (params (<ag>) (<bridgepoint1>) (<bridgepoint2>))
 (preconds (agent <ag>) (at <ag> <bridgepoint1>) (bridge <bridgepoint1> <bridgepoint2>))
 (effect (del at <ag> <bridgepoint1>) (at <ag> <bridgepoint2>))
)

(operator
 CROSS-BRIDGE
 (params (<ag>) (<bridgepoint1>) (<bridgepoint2>))
 (preconds (agent <ag>) (at <ag> <bridgepoint1>) (bridge <bridgepoint2> <bridgepoint1>))
 (effect (del at <ag> <bridgepoint1>) (at <ag> <bridgepoint2>))
)

(operator
 OPEN-CHEST
 (params (<ag>) (<thechest>) (<thekey>) (<objinchest>) (<pos>))
 (preconds (agent <ag>) (at <ag> <pos>) (at <thechest> <pos>) (keyfor <thekey> <thechest>)
           (have <ag> <thekey>) (contains <thechest> <objinchest>))
 (effect (have <ag> <objinchest>) (del contains <thechest> <objinchest>))
)

(operator
 PICKUP-OBJECT
 (params (<ag>) (<obj>) (<pos>))
 (preconds (agent <ag>) (object <obj>) (at <ag> <pos>) (at <obj> <pos>))
 (effect (have <ag> <obj>) (del at <obj> <pos>))
)

(operator
 DROP-OBJECT
 (params (<ag>) (<obj>) (<pos>))
 (preconds (agent <ag>) (object <obj>) (at <ag> <pos>) (have <ag> <obj>) )
 (effect (del have <ag> <obj>) (at <obj> <pos>))
)

(operator
 ATTACK
 (params (<ag1>) (<ag2>) (<pos>))
 (preconds (agent <ag1>) (eagent <ag2>) (at <ag1> <pos>) (at <ag2> <pos>))
 (effect (del eagent <ag2>) (dead <ag2>))
)

(operator
 LOOT
 (params (<ag1>) (<ag2>) (<pos>) (<loot>))
 (preconds (agent <ag1>) (dead <ag2>) (object <loot>) (at <ag1> <pos>) (at <ag2> <pos>) (have <ag2> <loot>))
 (effect (del have <ag2> <loot>) (have <ag1> <loot>))
)
```

Figure 11: The example scene expressed in First Order Logic

B Plan Runs

This appendix contains five of the 16 different plots. These five plots are typical, the other plots more or less correspond to these, only differing in details. The plots were generated by each team following its goal and changing the plan when necessary. The output is directly taken from the program, without changes except skipping debug output. All output posted by *[Agent.cpp]* is written by the agents itself, in the corresponding functions which, except writing the message to the screen, just change the state of the world as they are supposed to.

The first plot, Figure 12, shows a plot that is dominated by actions of agent *Badguy1*. Although both agents of team 1 are faster in the beginning and collect the chest and the key, they are both killed by agents of team 2, agent *Badguy1* loots the enemy agents for the objects and finally gets the gold.

```
1  [Agent.cpp] Agent Hero1 moving to position pos-bridge1end1.
2  [Agent.cpp] Agent Badguy1 moving to position pos-bridge2end1.
3  [Agent.cpp] Agent Hero2 picking up object obj-chest1.
4  [Agent.cpp] Agent Badguy1 crossed a bridge and is now at pos-bridge2end2.
5  [Agent.cpp] Agent Hero1 crossed a bridge and is now at pos-bridge1end2.
6  [Agent.cpp] Agent Hero1 moving to position pos-key.
7  [Agent.cpp] Agent Hero1 picking up object obj-key1.
8  [Agent.cpp] Agent Hero1 moving to position pos-bridge3end1.
9  [Agent.cpp] Agent Badguy1 moving to position pos-bridge3end1.
10 [Agent.cpp] Agent Hero1 crossed a bridge and is now at pos-bridge3end2.
11 [Agent.cpp] Agent Hero1 moving to position pos-chest.
12 [Agent.cpp] Agent Badguy1 crossed a bridge and is now at pos-bridge3end2.
13 [Agent.cpp] Agent Badguy2 attacks agent Hero1.
14 [Agent.cpp] Agent Badguy1 moving to position pos-chest.
15 [Agent.cpp] Agent Badguy1 loots agent Hero1 and gets object obj-key1.
16 [Agent.cpp] Agent Badguy1 attacks agent Hero2.
17 [Agent.cpp] Agent Badguy1 loots agent Hero2 and gets object obj-chest1.
18 [Agent.cpp] Agent Badguy1 dropped object obj-chest1.
19 [Agent.cpp] Agent Badguy1 took object obj-gold out of the chest obj-chest1 using key obj-key1.
```

Figure 12: An example plot, dominated by agent *Badguy1*

The second plot, Figure 13, shows a plot in which team 1 wins. Agent *Badguy1* reaches the key first and picks it up, but is later killed by agent *Hero2*, which loots *Badguy1*, opens the chest and brings the gold to agent *Hero1*. This scene is different from the others since not the agent which should have the gold in the end opens the chest.

```
1  [Agent.cpp] Agent Hero1 moving to position pos-bridge1end1.
2  [Agent.cpp] Agent Badguy1 moving to position pos-bridge2end1.
3  [Agent.cpp] Agent Hero2 picking up object obj-chest1.
4  [Agent.cpp] Agent Hero1 crossed a bridge and is now at pos-bridge1end2.
5  [Agent.cpp] Agent Badguy1 crossed a bridge and is now at pos-bridge2end2.
6  [Agent.cpp] Agent Badguy1 moving to position pos-key.
7  [Agent.cpp] Agent Badguy1 picking up object obj-key1.
8  [Agent.cpp] Agent Badguy1 moving to position pos-bridge3end1.
9  [Agent.cpp] Agent Badguy1 crossed a bridge and is now at pos-bridge3end2.
10 [Agent.cpp] Agent Badguy1 moving to position pos-chest.
11 [Agent.cpp] Agent Hero2 dropped object obj-chest1.
12 [Agent.cpp] Agent Hero1 moving to position pos-bridge3end1.
13 [Agent.cpp] Agent Hero2 attacks agent Badguy1.
14 [Agent.cpp] Agent Hero2 loots agent Badguy1 and gets object obj-key1.
15 [Agent.cpp] Agent Hero1 crossed a bridge and is now at pos-bridge3end2.
16 [Agent.cpp] Agent Hero1 moving to position pos-chest.
17 [Agent.cpp] Agent Hero2 took object obj-gold out of the chest obj-chest1 using key obj-key1.
18 [Agent.cpp] Agent Hero2 dropped object obj-gold.
19 [Agent.cpp] Agent Hero1 picking up object obj-gold.
```

Figure 13: An example plot, dominated by team 1

The plot in Figure 14 was already given and explained in Section 4.3, Figure 9.

```
1  [Agent.cpp] Agent Hero1 moving to position pos-bridge1end1.
2  [Agent.cpp] Agent Badguy1 moving to position pos-bridge2end1.
3  [Agent.cpp] Agent Hero2 picking up object obj-chest1.
4  [Agent.cpp] Agent Badguy1 crossed a bridge and is now at pos-bridge2end2.
5  [Agent.cpp] Agent Hero1 crossed a bridge and is now at pos-bridge1end2.
6  [Agent.cpp] Agent Hero1 moving to position pos-key.
7  [Agent.cpp] Agent Badguy1 moving to position pos-key.
8  [Agent.cpp] Agent Hero2 moving to position pos-bridge3end2.
9  [Agent.cpp] Agent Hero1 picking up object obj-key1.
10 [Agent.cpp] Agent Badguy2 moving to position pos-bridge3end2.
11 [Agent.cpp] Agent Badguy1 moving to position pos-bridge3end1.
12 [Agent.cpp] Agent Hero1 moving to position pos-bridge3end1.
13 [Agent.cpp] Agent Hero1 crossed a bridge and is now at pos-bridge3end2.
14 [Agent.cpp] Agent Badguy2 attacks agent Hero2.
15 [Agent.cpp] Agent Badguy1 crossed a bridge and is now at pos-bridge3end2.
16 [Agent.cpp] Agent Badguy2 loots agent Hero2 and gets object obj-chest1.
17 [Agent.cpp] Agent Badguy1 attacks agent Hero1.
18 [Agent.cpp] Agent Badguy2 dropped object obj-chest1.
19 [Agent.cpp] Agent Badguy1 loots agent Hero1 and gets object obj-key1.
20 [Agent.cpp] Agent Badguy1 took object obj-gold out of the chest obj-chest1 using key obj-key1.
```

Figure 14: The example plot that was already discussed in Section 4.3. This plot nicely shows cooperative behavior in both teams.

The plot in Figure 15 can still go on with different ending conditions. Since the scene was ended after the first team had succeeded, it was stopped after agent *Badguy1* received the gold. Still, there is a plan for team 1 to win, which would include killing agent *Badguy1.* This plot does nearly not make use of agent *Badguy2* and *Hero2*, which can be explained by the actions done taking a random amount of time. Thus agent *Hero2* is still busy with picking up the chest, while the other agents move. Since access to the ressource *obj-chest* is locked and agent *Badguy2* certainly also tries to pick it up, agent *Badguy2* has to wait until the action is finished. These artifacts should not occur in a non-simulated game environment, where picking up an object only takes a reasonable amount of time.

```
1  [Agent.cpp] Agent Hero1 moving to position pos-bridge1end1.
2  [Agent.cpp] Agent Badguy1 moving to position pos-bridge2end1.
3  [Agent.cpp] Agent Hero2 picking up object obj-chest1.
4  [Agent.cpp] Agent Badguy1 crossed a bridge and is now at pos-bridge2end2.
5  [Agent.cpp] Agent Badguy1 moving to position pos-key.
6  [Agent.cpp] Agent Hero1 crossed a bridge and is now at pos-bridge1end2.
7  [Agent.cpp] Agent Badguy1 picking up object obj-key1.
8  [Agent.cpp] Agent Hero1 moving to position pos-key.
9  [Agent.cpp] Agent Badguy1 moving to position pos-bridge3end1.
10 [Agent.cpp] Agent Badguy1 crossed a bridge and is now at pos-bridge3end2.
11 [Agent.cpp] Agent Hero1 moving to position pos-bridge3end1.
12 [Agent.cpp] Agent Badguy1 moving to position pos-chest.
13 [Agent.cpp] Agent Badguy1 attacks agent Hero2.
14 [Agent.cpp] Agent Badguy1 loots agent Hero2 and gets object obj-chest1.
15 [Agent.cpp] Agent Hero1 crossed a bridge and is now at pos-bridge3end2.
16 [Agent.cpp] Agent Badguy1 dropped object obj-chest1.
17 [Agent.cpp] Agent Badguy1 took object obj-gold out of the chest obj-chest1 using key obj-key1.
```

Figure 15: An example plot, dominated by agent *Badguy1* again

The plot in Figure 16 is a plot dominated by agent *Hero1.* Often agent *Hero1* or *Badguy1* dominate the scene, since the goal of both teams it that these agents have the gold. If the planner sees no increased efficiency when using both agents, just one agent is used. This might be changed by allowing both agents to have the gold to win the scene.

```
1  [Agent.cpp] Agent Badguy1 moving to position pos-bridge1end1.
2  [Agent.cpp] Agent Hero1 moving to position pos-bridge2end1.
3  [Agent.cpp] Agent Badguy2 picking up object obj-chest1.
4  [Agent.cpp] Agent Badguy1 crossed a bridge and is now at pos-bridge1end2.
5  [Agent.cpp] Agent Badguy1 moving to position pos-key.
6  [Agent.cpp] Agent Hero1 crossed a bridge and is now at pos-bridge2end2.
7  [Agent.cpp] Agent Badguy1 picking up object obj-key1.
8  [Agent.cpp] Agent Hero1 moving to position pos-key.
9  [Agent.cpp] Agent Hero1 attacks agent Badguy1.
10 [Agent.cpp] Agent Hero2 attacks agent Badguy2.
11 [Agent.cpp] Agent Hero1 loots agent Badguy1 and gets object obj-key1.
12 [Agent.cpp] Agent Hero1 moving to position pos-bridge3end1.
13 [Agent.cpp] Agent Hero1 crossed a bridge and is now at pos-bridge3end2.
14 [Agent.cpp] Agent Hero1 moving to position pos-chest.
15 [Agent.cpp] Agent Hero1 loots agent Badguy2 and gets object obj-chest1.
16 [Agent.cpp] Agent Hero1 dropped object obj-chest1.
17 [Agent.cpp] Agent Hero1 took object obj-gold out of the chest obj-chest1 using key obj-key1.
```

Figure 16: An example plot, dominated by team 1 after team 2 having a good start.