A PDDL-Based Planning Architecture to Support Arcade Game Playing

Olivier Bartheye and Éric Jacopin

MACCLIA, CREC Saint Cyr, Écoles de Coëtquidan, F-56381 GUER Cedex
{olivier.bartheye,eric.jacopin}@st-cyr.terre.defense.gouv.fr

Abstract. First, we explain the Iceblox game, which has its origin in the Pengo game. After carefully listing requirements on game playing, the contents of plans, their execution and planning problem generation, we design a set of benchmarks to select good playing candidates among currently available PDDL-based planners. We eventually selected two planners which are able to play the Iceblox video game well and mostly in real time. We describe both the predicates and some of the operators we designed. Then, we give details on our planning architecture and in particular discuss the importance of the generation of PDDL planning problems. We wish to report that no planner was tweaked during the benchmarks.

1 Introduction

On one side, the quality of the Artificial Intelligence is part of today's game evaluations and on the other planning systems still wait for going mainstream. Could a close encounter of both be fruitful? Over the years planning has occurred in the video-game domain with some success (e.g. [1]), with the work on the Goal Oriented Action Planning architecture being probably the most sucessful [2]. This time, we would like video-gaming to occur in the planning domain: designing and implementing a game and connect it to today's planners. First, thanks to the International Planning Competition [3], today's planners all accept planning problems written in PDDL [4] which is going to ease the selection of a good planner for video-gaming. Then, it is not only a matter of constructing a plan but also to generate PDDL planning problems from a video-game situation. Third, it is also a matter of executing plans in a video-game. And finally, what shall happen when the dynamics of video-games makes useless the plan currently executed? These matters and questions have rarely been addressed in the case of the close encounter of planning and video-games and it is the purpose of this paper to report the sparks.

This paper is organized as follows. We begin with a description of the Iceblox arcade game and argue why this game is a good choice for testing the performances of PDDL-based planners coping with planning problems from this domain. The next section presents and discusses the necessary requirements on a planning system playing an arcade game such as Iceblox. Then, we report on the selection process of available PDDL-based planners, from a wide perspective (simple Iceblox situations, several planners, various planning problem properties) to a narrow perspective where two planners were tested against more realistic Iceblox situations. Finally, we outline the main procedures of our arcade game playing architecture: the generation of PDDL planning problem and the plan execution process and discuss the importance of path planning.

2 Iceblox

Description. In the Iceblox video game [5], the player presses the arrow keys to move a penguin (named Pixel Pete) horizontally and vertically in rectangular mazes made of ice blocks and rocks, in order to collect coins. But flames patrol the maze (their speed is that of the penguin) and can kill the penguin in a collision; moreover, each coin is iced inside an ice block which must be cracked several times before the coin is ready for collection. The player can push (space bar) an ice block which will slide until it collides with another ice block, a rock or any of the four side of the game. A sliding ice block stops when it collides into another ice block, a rock or any of the four sides of the game and kills a flame when passing over it. If the player pushes an ice block which is next to another ice block, a rock or a side of the game, it cannot slide freely and shall begin to crack. Once cracked, an ice block cannot move any more and thus pushing a cracked ice block eventually results, after seven pushes, in its destruction. An ice block which contains a coin slides and cracks as does an ordinary ice block. A coin cannot slide; it can only be collected once revealed after the seventh push. The player gets to the next, randomly generated, level when all coins have been collected. Killed flames reappear from any side of the game and from time to time, as in Figure 1, it happens that locking the flames is a better strategy than killing them: since there is no constraint on time to

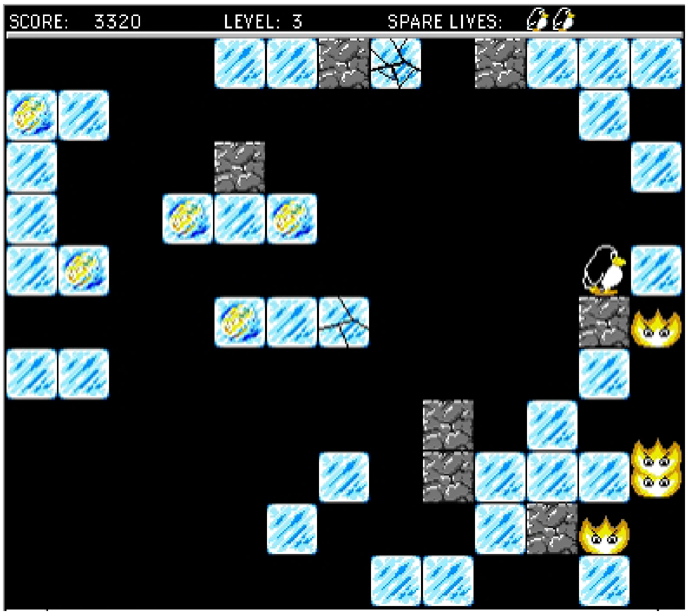


Fig. 1. An Iceblox game in progress

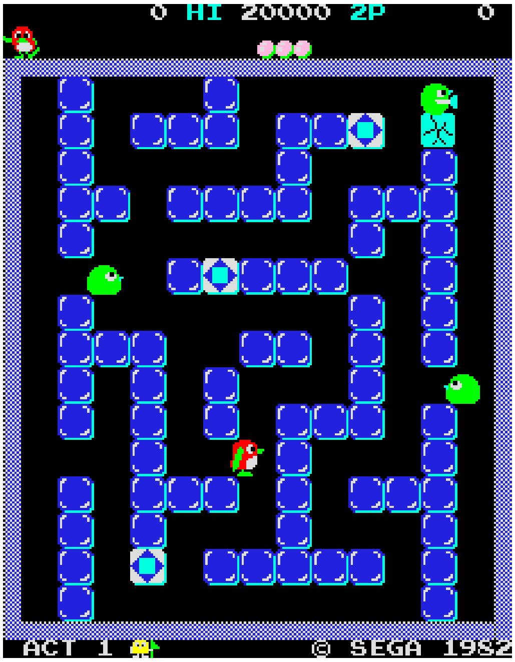


Fig. 2. A Pengo game in progress [7]

finish any level of Iceblox, this strategy gives the player all the latitude to collect coins. The number of flames and rocks increase during several levels and then cycle back to their initial values. Finally, the player gets 200 points for each collected coin, 50 points for each killed flames and begins the game with three spare lives.

Why Iceblox?. Iceblox is an open and widely available java implementation [6, pages 264–268] of the Pengo arcade game, published by Sega in 1982 (see Figure 2). In Pengo, there are pushable and crackable ice blocks but neither rocks nor coins; the main objective of the game is to kill all the bees patrolling the ice blocks mazes. Bees hatch from eggs contained in some ice blocks; the destruction of an ice block containing eggs results in the destruction of these eggs. In Pengo, bees can be killed with a sliding ice block and by a collision with the penguin when they are stunned. Pushing a side of the game when a bee is next to this side shall stun the bee for some short time. The player is given 60 seconds to kill all the bees of any of the 16 levels of the game (after the sixteenth level, the game cycles back to the first level). Bees accelerate when reaching the 60 seconds time limit and the player is given a short extra time to kill them before they vanish, thus making the level impossible to finish. There also are pushable diamonds which cannot be collected but must be aligned to score extra points. As in any arcade game of the time, there are many bonuses, making the high score a complex goal; and both the killing of the bees and the alignment of the diamonds score differently according to the situation. We refer the reader to [7] for further details about the game.

Iceblox obviously is easier than Pengo: no time constraint, uniform speed and scoring, no bonuses,... Main targets (coins) fixed rather than moving (bees or flames), flames cannot push ice blocks and fighting them is optional,... However, the locking of flames in a closed maze of ice blocks and rocks possesses a geometrical spirit similar to that of the alignment of diamonds in the Pengo game.

Consequently, we chose Iceblox because it is a simpler start in the world of arcade video games than the commercial games of the time.

3 Requirements

Game playing. On its way to collect coins, the planning system playing the Iceblox game shall badly either fight or avoid flames: disorganized movements or paths longer than necessary shall be allowed as long as they do not prevent Pixel Pete from collecting coins. Both the fighting and avoidance of flames shall be realized in (near) real time: game animation might look sometimes slow or sometimes irregular but shall never stop; in particular, flames shall always be patrolling the maze, even when traditional problem solving (that is, Planning) shall take place.

Plans (what's in a plan?). A plan shall be a set of partially ordered operators which represent actions in the Iceblox domain. What kind of actions shall we allow to be part of a plan?

Let us distinguish five kinds of actions that may happen in an arcade video game such as (Pengo or) Iceblox:

1. Drawing a frame to animate a sprite; this is the pixel kind of action. Animation always takes place, even if the player does nothing: for instance, a flame is always animated in the Iceblox game, even in the case where it cannot move because it is surrounded by ice blocks.
2. Basic moving of one step left, right, up or down and pushing an ice block. This is the sprite kind of action which is not interruptible for a certain number of pixels, usually the number of pixels of a side of the rectangle where the sprite is drawn (in the case of Iceblox, sprites are squares of 30 by 30 pixels). Each action of this kind exactly corresponds to a key pressed by the player: arrows keys to move left, right, up and down and the space bar to push an ice block.
3. Moving horizontally or vertically in a continuous manner, that is making several consecutive basic moves in the same direction; this is the path-oriented kind of action and means following a safe path in the maze.
4. Avoiding, fleeing, fighting or locking the flames in a closed maze of ice blocks and rocks. This is the problem solving kind of action and concerns survival and basic scoring. Ice block cracking until destruction to collect a coin also is a problem solving kind of action.
5. Defining goals and setting priorities on them; this is the game level strategy kind of action and is oriented towards finishing the game with the highest score. In the

case of Iceblox, this means ordering the collection of coins, noticing the opportunity to lock the flames in a maze and setting its importance in finishing the level; or else seizing the opportunity to seek cover behind ice blocks of a geometrical configuration created during game playing.

Out of the five kinds of action, only push actions of kind 2, abstract Moves based on rectilinear moves of kind 3 and the coin collection action of kind 4 shall be represented as operators.

The Plans just described are *not* as simple as they may appear: it is *not* the case that either their construction shall be too simple or their use shall be redundant with the player's actions. First, these Plans look like the plans for the storage domain which is part of the deterministic IPC benchmarks [3]. Second, as Table 1 shows, current planners agree with their non easiness. Third, there are Plans (both constructed and executed [8] or only executed [1]) in commercial video-games which are shorter and simpler.

Plan execution. The plan execution system receives a plan from the Planning system in order to execute it in the Iceblox game. Execution of a plan means executing the players actions (that is, actions of kind 2) corresponding to the operators of the plan, in an order compliant with the partial order of the plan. For instance, the plan execution system shall decide of several basic moves actions to execute a Move operator in the plan; which can include ice block pushing to facilitate the realization of the Move operator. The plan execution system shall also decide of taking advantage of the current Iceblox situation to avoid pushing an ice block when a flame is no longer aligned with it, to choose a new weapon or to avoid unexpected flames. These decision shall result from a local analysis and no global situation assessment shall be realized to take such advantages. The plan execution system shall eventually warn the user when it terminates (whatever the outcome, success or failure). The player can terminate the plan execution at any time by pressing any arrow key or else the space bar.

In case of emergency situations (e.g. no weapon is locally available, fleeing seems locally impossible, etc), the plan execution system shall call for re-Planning.

Planning. Planning refers to the plan formation activity. The player shall intentionally start the planning activity by pressing a designated key (e.g. the "p" key). Once running, planning shall not stop the Iceblox game: flames shall patrol the maze and sliding ice blocks shall continue to slide while the plan formation activity is running. The user shall be warned when the activity ends, either with success or else failure (e.g. a different sound for each case). If any of the arrow keys or else the space bar (push) is pressed before the end of the planning activity, then it is terminated. The user shall be able to play at any time.

Planners. The main objective of the work reported in this paper is to determine whether any of today's planners is able to play the Iceblox game: no planner shall be specifically designed to play Iceblox and no existing planner shall be tweaked to play Iceblox.

Due to the on-going effort of the International Planning Competition (IPC) [3], many planners are available today and most of them accept Planning data (i.e. states, operators) written in the PDDL language. Because of this wide acceptance, the overhead of both generation and processing of PDDL shall first be ignored; writing directly to a planner's data structures shall be considered only if game playing requirements are not met. Consequently, the Planning activity in the Iceblox domain shall take as input an initial state, a final state and a set of operators all written in the PDDL language.

Available planners are in general more ready to process PDDL text files than ready to be connected to a video game. We shall thus begin with the design of a set of Iceblox planning problems of increasing difficulty, in the spirit of the IPC: executable files for the planners, PDDL Iceblox problems files and scripts files to automate this Iceblox benchmarking process. If a planner fails these off line tests then it shall not be a good candidate for Iceblox video game playing.

What shall demonstrate that a planner is a good candidate for a connection to Iceblox? Solving the Iceblox benchmarks fast enough seems the obvious answer. Rather, the question should be: what is the time limit beyond which the current Iceblox situation is so dangerous that an action has to be taken right away, thus changing the initial state of the planning problem and consequently asking for re-planning? Iceblox has a good playability when the frame rate is about 30 frames per second, that is, when the flames coordinates (in pixel) are updated about 30 times per second. Luckily, the sprites are 30 by 30 pixels; it then takes about 1 second for a flame to move from one crossroad to another. According to the Iceblox code, a flame gets a random new direction between 1 and 4 crossroads, while keeping in mind that the new direction at the fourth crossroad might just be the same than the previous one. We can then consider that the limit is when Pixel Pete is 4 crossroads away from a flame, which gives a plan search runtime of at most 4 seconds. If a plan has been found then it needs to be executed, which undoubtedly takes time to trigger. Consequently, the flame should not reach the fourth crossroad and since movement between two crossroads is not interruptible, the flame should not reach the third crossroad before the plan search ends, which gives us a time limit of 3 seconds.

A planner shall be a good candidate for Iceblox video-game playing if it can (off line) solve Iceblox planning problems within the time limit of 3 seconds. This time limit sorts out dangerous flames from harmless flames.

4 Minimal Iceblox Situations

Three examples. We here describe three Iceblox planning problems of increasing difficulty which we designed in order to get a set of good candidate planners. Both plan length and planning problem size shall be used as criteria of difficulty: the larger number of predicates describing both the initial and the final states and the larger number of operators in the plan solution, the more difficult the problem.

It is necessary to collect a coin in each of the three problems. Figures 3, 4, 5 contain a screen shot of an iceblox planning problem and its PDDL code. See Figure 3 for the simplest of our three problem; this is obviously a "Welcome-to-Iceblox-world!" problem. In Figure 4, we introduce danger from one flame with only one weapon to kill this flame. In Figure 5, we set the case for two weapons to kill a flame guarding a coin. The properties of these three problems are the following:

See Figure	3	4	5
Requires flame fighting?	No	Yes	Yes
Number of weapons	0	1	2
Number of predicates (initial and final states)	7	10	13
Number of typed objects	4	8	10
Number of paths leading to the coin	1	1	2
Length of solution plan	2	4	4 or 5

Why these three problems? First because their number of predicates all are very small and yet they cover the basics of Iceblox game playing: these situations are so simple that if a planner fails at these problems then it is certainly not able to play iceblox. Moreover, although more problems have been designed, they confirm the results of Table 1. Given as Iceblox levels to an Iceblox planning system, these problems can be solved in real time; screen shots of the final Iceblox situations are gathered in Figure 6.

Planning problems predicates. The following predicates are used to describe both the initial and final states of the planning problems of Figures 3, 4 and 5 (crossroad$_{i,j}$ denotes the location at the intersection of line i and column j):

- (position i j): a sprite (Pixel Pete, ice block, iced coined, weapon) is at the crossroad$_{i,j}$.
- (extracted i j): the coin at the crossroad$_{i,j}$ has been collected by the player.
- (guard i_1 j_1 i_2 j_2): the flame at the crossroad$_{i_1,j_1}$ guards the coin at the crossroad$_{i_2,j_2}$. The idea of guarding a coin is linked to the 3 seconds time constraint (see the planners requirements): a flame makes dangerous the collection of a coin when it is within a range of 3 crossroads.
- (iced-coin i j): there is an ice block at the crossroad$_{i,j}$ which contains a coin.
- (protected-cell i j): there exists a path towards the crossroad$_{i,j}$. This path is safe: no flame makes this path dangerous.
- (reachable-cell i j): there exists a path towards the crossroad$_{i,j}$; there exists at least one flame putting this path in danger.
- (weapon i_1 j_1 i_2 j_2 i_3 j_3): there exists a weapon at the crossroad$_{i_2,j_2}$; Pixel Pete should push this weapon from the crossroad$_{i_1,j_1}$. The weapon shall stop sliding at the crossroad$_{i_3,j_3}$; this is useful information when an ice block needs more than one push before the final kick at a flame.

Operators use two more predicates:

- (blocked-path i j): there is an ice block on the path to crossroad$_{i,j}$.
- (blocked-by-weapon i_1 j_1 i_2 j_2): the weapon at crossroad$_{i_2,j_2}$ is on the path to crossroad$_{i_1,j_1}$.

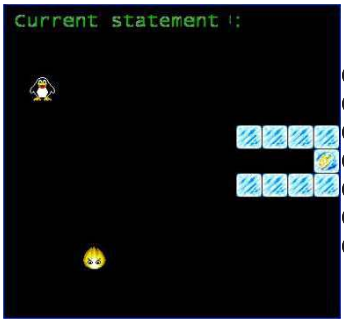
Over all the operators we designed, 4 proved to be critical for Iceblox game playing: move-to-crossroad, destroy-weapon, kick-to-kill-guard and extract. We hope their names are self explanatory. There are more (e.g. pushing a block to lock flames in a maze of ice blocks and rocks) but basic Iceblox playing is impossible if you don't get those 4 operators. Due to space limitations, we only give the PDDL code of kick-to-kill-guard and extract; these operators are given to the planners for benchmarking and playing:

```
(:action extract
    :parameters (?coinx - coord-i ?coiny - coord-j )
    :precondition (and (protected-cell ?coinx ?coiny)
            (iced-coin ?coinx ?coiny) (position ?coinx ?coiny)
            (reachable-cell ?coinx ?coiny))
    :effect (and (extracted ?coinx ?coiny)
            (not (iced-coin ?coinx ?coiny))
            (not (protected-cell ?coinx ?coiny))
            (not (reachable-cell ?coinx ?coiny))))

(:action kick-to-kill-guard
    :parameters
            (?reachablewx - coord-i ?reachablewy - coord-j
             ?weaponx - coord-i ?weapony - coord-j
             ?newweaponx - coord-i ?newweapony - coord-j
             ?guardx - coord-i ?guardy - coord-j
             ?coinx - coord-i ?coiny - coord-j
             ?blockedx - coord-i ?blockedy - coord-j)
    :precondition (and (iced-coin ?coinx ?coiny)
            (position ?reachablewx ?reachablewy)
            (guard ?guardx ?guardy ?coinx ?coiny)
            (weapon ?reachablewx ?reachablewy ?weaponx ?weapony
                ?newweaponx ?newweapony)
            (reachable-cell ?weaponx ?weapony)
            (protected-cell ?weaponx ?weapony))
    :effect (and (position ?weaponx ?weapony)
            (protected-cell ?coinx ?coiny)
            (blocked-by-weapon ?blockedx ?blockedy ?newweaponx ?newweapony)
            (reachable-cell ?newweaponx ?newweapony)
            (protected-cell ?newweaponx ?newweapony)
            (not (reachable-cell ?weaponx ?weapony))
            (not (protected-cell ?weaponx ?weapony))
            (not (reachable-cell ?blockedx ?blockedy))
            (not (guard ?guardx ?guardy ?coinx ?coiny))
            (not (weapon ?reachablewx ?reachablewy ?weaponx ?weapony
                ?newweaponx ?newweapony))))
```

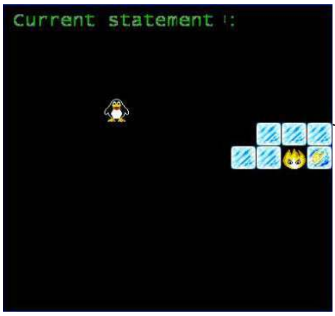
These operators can be much simpler and indeed we designed and used very simple versions of them. But of course, there is a compromise between simplicity which questions the overall utility of all this, and complexity which gets the planners nowhere. It is very easy to put more information in the operators than necessary; and then: bye bye runtimes!

Results. Several planners have been tested against these three problems (and others), with a 3.2 GHz Pentium 4 PC, loaded with 1Gb of memory. The runtimes, in seconds,



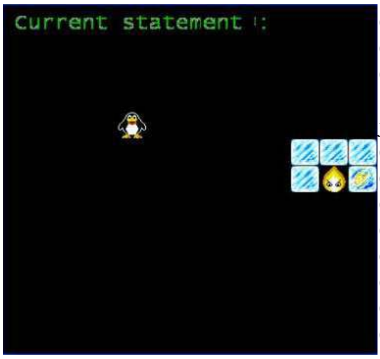
```
(define (problem first-case)
(:domain iceblox-strips)
(:objects i-1 - coord-i i-12 - coord-i j-1 - coord-j j-4 - coord-j)
(:init (position i-1 j-1) (iced-coin i-12 j-4)
(protected-cell i-1 j-1) (reachable-cell i-1 j-1)
(protected-cell i-12 j-4) (reachable-cell i-12 j-4))
(:goal (extracted i-12 j-4)))
```

Fig. 3. The danger is away, the path is obvious: you're fired if you can't solve this one



```
(define (problem second-case)
(:domain iceblox-strips)
(:objects i-4 - coord-i i-12 - coord-i i-11 - coord-i j-2 - coord-j
j-4 - coord-j j-3 - coord-j j-11 - coord-j j-10 - coord-j)
(:init (position i-4 j-2) (iced-coin i-12 j-4)
(protected-cell i-4 j-2) (reachable-cell i-4 j-2)
(guard i-11 j-4 i-12 j-4) (protected-cell i-11 j-3)
(weapon i-11 j-2 i-11 j-3 i-11 j-10 i-11 j-11)
(reachable-cell i-11 j-3) (reachable-cell i-12 j-4))
(:goal (extracted i-12 j-4)))
```

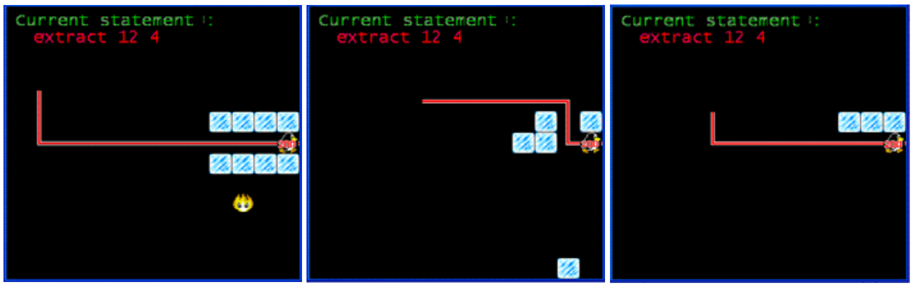
Fig. 4. There is danger, but the fight is easy



```
(define (problem third-case)
(:domain iceblox-strips)
(:objects i-4 - coord-i i-12 - coord-i i-11 - coord-i i-9 - coord-i i-
10 - coord-i j-2 - coord-j j-4 - coord-j j-3 - coord-j j-11 - coord-j
j-10 - coord-j)
(:init (position i-4 j-2) (iced-coin i-12 j-4)
(protected-cell i-4 j-2) (reachable-cell i-4 j-2)
(guard i-11 j-4 i-12 j-4) (protected-cell i-10 j-4)
(weapon i-9 j-4 i-10 j-4 i-11 j-4 i-12 j-4)
(weapon i-11 j-2 i-11 j-3 i-11 j-10 i-11 j-11)
(protected-cell i-11 j-3) (reachable-cell i-10 j-4)
(reachable-cell i-11 j-3) (reachable-cell i-12 j-4))
(:goal (extracted i-12 j-4)))
```

Fig. 5. Two unsafe paths: one shorter, one longer…

averaged over ten runs, are gathered in Table 1. The planners appearing in Table 1 reflect all the cases which happened during the tests: some planners rejected part or all of the PDDL files, reporting PDDL mistakes and some planners found no solution to



```
Current statement :
  extract 12 4
```

```
Current statement :
  extract 12 4
```

```
Current statement :
  extract 12 4
```

Fig. 6. When off line benchmarking becomes real time Iceblox playing. The red and white track marks the path followed by the penguin during the plan execution.

Table 1. Performances of several planners on PDDL files of the 3 problems of Figures 4, 5 and 6. A typed and untyped coordinates version was tested for each problem. All times, averaged over ten runs, are in seconds; planners were given 120 seconds to search for a solution although the required time limit is 3 seconds (see the requirements for planners).

	Typed			Untyped		
	Fig. 3	Fig. 4	Fig. 5	Fig. 3	Fig. 4	Fig. 5
FF [9]	0.01	0.49	1.83	0.01	> 120	
HSP2 [12]	PDDL Syntax errors					
Metric-FF [13]	0.01	0.33	1.07	0.01	> 120	
Qweak [10]	0.01	0.02	0.02	0.02	0.02	0.02
SGPlan [14]	0.01	0.35	1.14	0.01	> 120	
STAN [15]	0.18	PDDL Syntax errors		0.2	> 120	
TSGP [16]	0.03	25	> 120	0.05	> 120	
YAHSP [17]	0.01	No plan found		0.01	No plan found	

the problems. Hopefully, others correctly read the PDDL files; among those planners, some found solutions within the time limit and others did not. All the solutions found were correct: no planner we tested returned an incorrect plan.

Four planners, FF, Metric-FF, Qweak and SGPlan compose the set of good candidates for playing Iceblox. We eventually reduced the set to the well known and successful FF [9] and the less known and fastest but exotic Qweak [10][11].

Again, *no* planner was tweaked during both benchmarking and game playing (see the requirements for planners).

As you can notice, ice blocks in this design can only be pushed for killing flames guarding an iced coin and there is no action to crush an ice block. Indeed, we implemented the crushing of ice blocks in the execution module. Then, by carefully designing our levels, both FF and Qweak were able to play large Iceblox levels (i.e. 12 lines by 12 columns with at most 3 flames and several ice blocks) almost in real-time.

In the next section, we see how to plan the pushing and crushing of ice blocks with FF and Qweak.

5 Game Playing Situations

The previous game situations might be perfect for the selection of PDDL-based planners, but they are far from real iceblox game playing situations. In this section we carry on testing and see how more realistic game playing situations can be handled by the planners FF and Qweak. Looking at figure 1 we can observe that putting more objects in an Iceblox game level produces more realistic game situations: more ice blocks, more coins, more flames, more weapons,... We need problems with more objects, and consequently more predicates and more actions. However, FF's instantiation mechanism is sensible to more actions with more parameters. Consequently, if we propose more actions to the planners, we should be careful with the number of objects of our problems.

The first four problems we designed have the following properties:

See figure 7	*top-left*	*top-right*	*bottom-left*	*bottom-right*
Requires flame fighting?	No	Yes	Yes	Yes
Number of weapons	0	1	1	4
Number of predicates (initial and final states)	7	10	10	19
Number of typed objects	5	8	8	15
Number of paths leading to the coin	1	1	2	4
Length of solution plan	2	4	5	7

As you can observe, the size of these problems (except the *bottom-right*) correspond to the size of the problems of figures 3, 4 and 5. However, our domain file now has 7 actions instead of 4 and our new kick-to-kill-guard action now possesses 11 parameters while our push action has 6 parameters; the crush action, given below, only possesses 4 parameters.

New planning problem predicates are the following:

- (guard f i_1 j_1 i_2 j_2): the flame f at the crossroad$_{i_1,j_1}$ guards the coin at the crossroad$_{i_2,j_2}$. This predicate now possesses a new parameter f to record a flame number to facilitate the execution process: we may have already killed this flame, thus making needless to focus on it any longer; or it is another flame which is now dangerous and a new plan must be computed.
- (blocked-by-cell i_1 j_1 i_2 j_2): there is an ice block at the crossroad$_{i_1,j_1}$ which can be either pushed or else crushed when at the crossroad$_{i_2,j_2}$; this predicate is a rewriting of the two predicates (blocked-path i j) and (blocked-by-weapon i_1 j_1 i_2 j_2).
- (crushable i j): the ice block at the crossroad$_{i,j}$ can be crushed so that (reachable-cell i j) becomes true.
- (pushable i_1 j_1 i_2 j_2): the ice block at the crossroad$_{i_1,j_1}$ can be pushed; if pushed, it shall stop at the crossroad$_{i_2,j_2}$.

One operator use one more predicate:

- (connected-with-cell i_1 j_1 i_2 j_2): this new predicate is always used in conjunction with the predicate (reachable-cell i_2 j_2) thus asserting that the cell at the crossroad$_{i_1,j_1}$ also is reachable (see the set-reachable action below). In fact, any crossroad is reachable in the Iceblox game since it is always possible to push or crush an ice block to reach it. So, in theory, we should generate this predicate for many interesting positions we could detect and in particular around weapons. However, the dynamics of the game render this generation useless: flames move and weapons change.

We first implemented ice block crushing in the execution module (see next section) but we soon realized it was not coherent with our design of PDDL actions: crushing an ice block is a 7 push action in the same direction. Consequently, we designed an abstract push action gathering several basic pushes as our move action gathers several basic moves:

```
(:action crush-iceblock
    :parameters
          (?crushablex - coord-i ?crushabley - coord-j
            ?blockedx - coord-i ?blockedy - coord-j)
    :precondition (and (position ?crushablex ?crushabley)
          (reachable-cell ?crushablex ?crushabley)
          (protected-cell ?crushablex ?crushabley))
          (crushable-cell ?crushablex ?crushabley)
          (blocked-by-cell ?blockedx ?blockedy ?crushablex ?crushabley))
    :effect (and (reachable-cell ?blockedx ?blockedy)
          (not (reachable-cell ?crushablex ?crushabley))
          (not (crushable-cell ?crushablex ?crushabley))))
```

As discussed for the new predicate connected-with-cell, we designed an action to compute, through planning, the transitive closure of reachable-cell predicate. In practice, solution plans are rarely generated with this action.

```
(:action set-reachable
    :parameters
          (?cellx - coord-i ?celly - coord-j ?cellu - coord-i ?cellv - coord-j)
    :precondition (and (reachable-cell ?cellx ?celly)
          (connected-with-cell ?cellu ?cellv ?cellx ?celly))
    :effect (and (reachable-cell ?cellu ?cellv))))
```

Results. We tested both FF and Qweak against typed versions of the four problems of figure 7; anticipating bad runtimes, we traded our previous every-office-of-yesterday-machine to a newer 2.20GHz T7500 core duo loaded with 3Gb of memory. FF solves the first two problems in, respectively, 30 milli-seconds and 200ms, and fails to solve the last two, running out of memory: as predicted, FF's instantiation mechanism can be blamed for these failures. We thus decided to test Metric-FF and SGPLan as well, but with no more success than FF. Qweak solves the first three problems at about the same speed of 30ms and solves the last problem in 140ms: a faster machine entails faster runtimes.

Again, no planner was tweaked during these tests, but we did correct a couple of bugs in Qweak which appeared when testing even bigger problems.

Realistic iceblox levels can be solved in real-time. However, to achieve good playability in the video-game domain, everything must be fast: the construction of plans, of

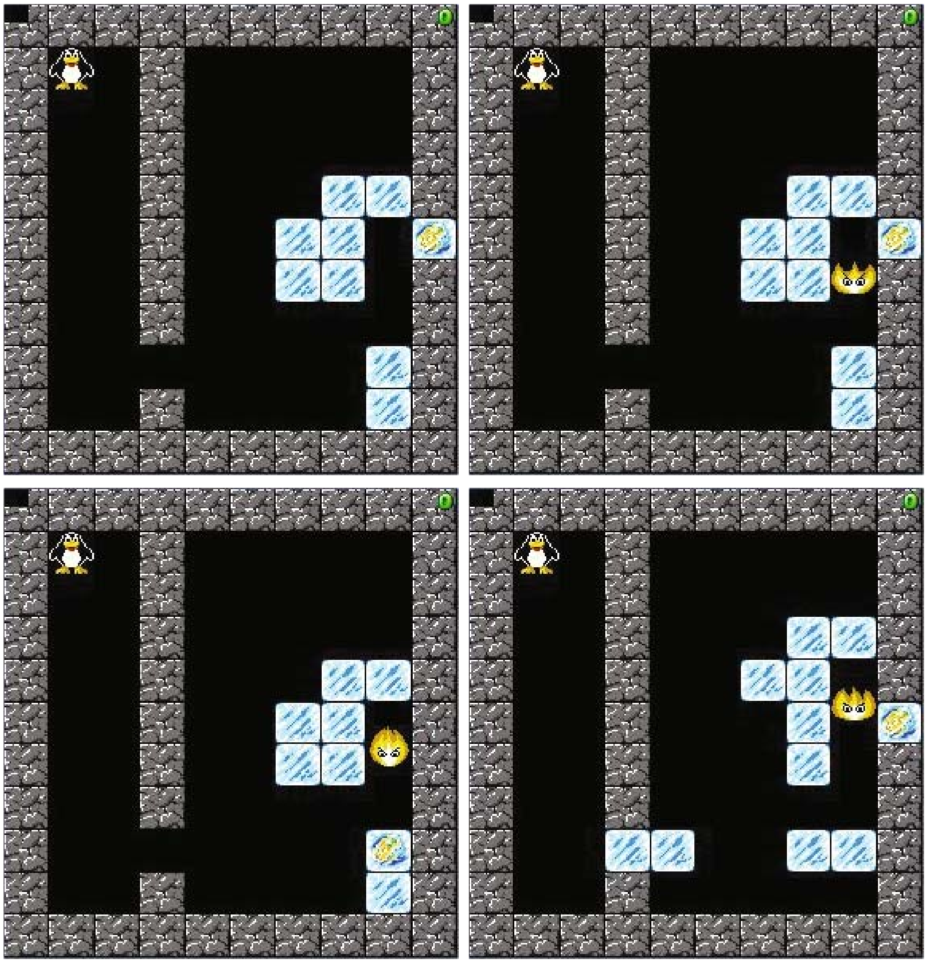


Fig. 7. The Penguin starts in line 1 and column 1 and its objective is to extract the only coin in each problem. When pushed, an ice block can slide only when its other side is free; an ice block can be crushed if its opposite side is not free. A flame dies when it collides with a sliding ice block which is then called a weapon. Because there even is no flame at all, the *top-left* problem is simpler than that of figure 3 and the *top-right* problem is similar to that of figure 4. The *lower-left* problem corresponds to that of figure 5: when pushed to kill the flame, the ice block in line 4 and column 8 shall stop just above the iced coin thus blocking the path to the iced coin; then, the Penguin must either walk round the ice blocks to reach the left side of the iced coin or else destroy the weapon which stopped sliding on the upper side of the iced coin. The *lower-right* problem appears to be more difficult than all other previous problems: first, there are four weapons which, once pushed, could collide with the flame; second, there is an obstacle on the path to these weapons: the entry point between the two rooms is blocked and an ice block must be crushed and another one must be pushed. Now, go to figure 8 to see how difficult these problems really are.

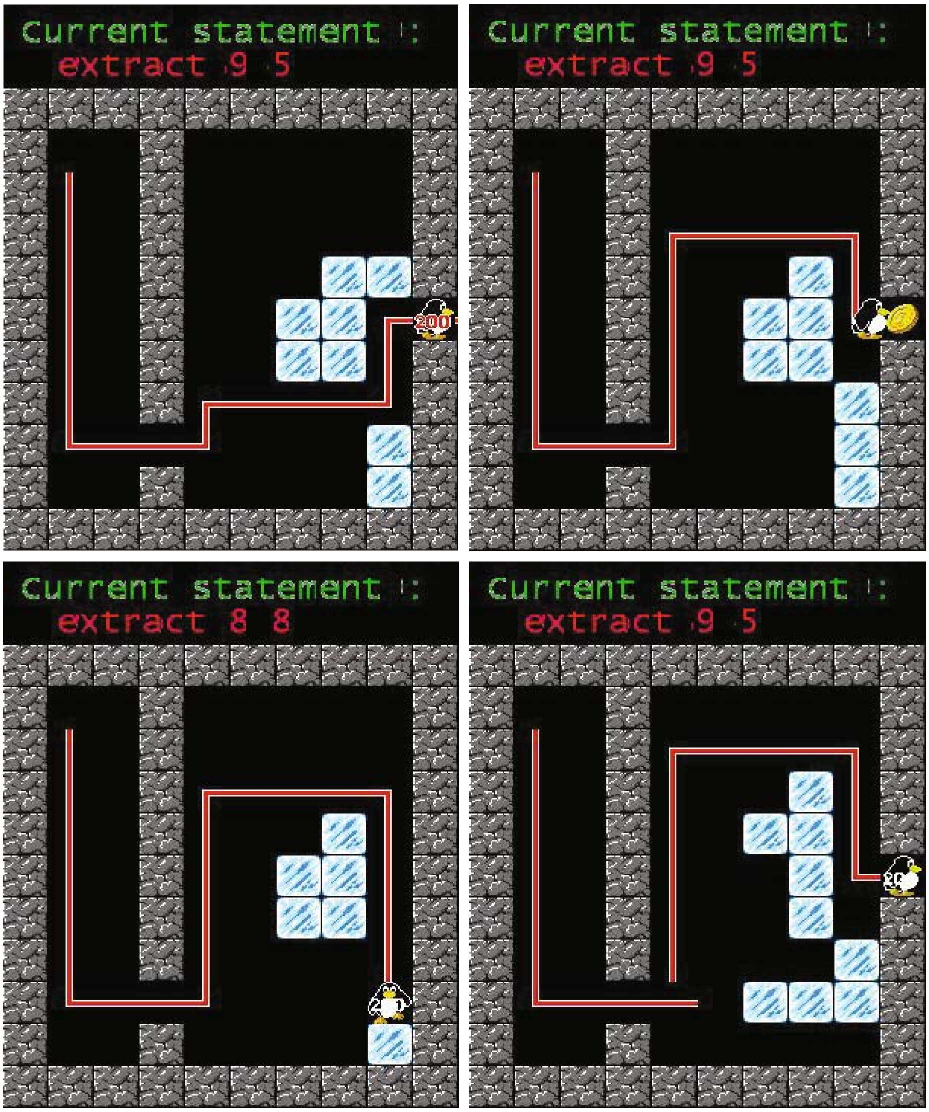


Fig. 8. The moment of truth for the planning problems described in figure 7. The four problems must be solved so we feed the planner with actions to push and crush ice blocks: 7 actions and 12 predicates in the domain file are now necessary to solve these problems whereas 4 actions and 10 predicates were needed for the problem of figures 3, 4 and 5. Current IPC planners can solve the two problems of the *top* line on today's machines but only when objects are typed and the `:typing` requirement appears in the PDDL problem file (several milliseconds for FF [9] on the *top-left* problem and 0.2s for the *top-right* problem). However, the two problems on the *bottom* line are too difficult: all planners but Qweak [10] fails to find a solution within 120 seconds: the 11 parameters of the kick-to-kill-guard action, the 6 parameters of the push action and the number of weapons in the *lower-right* problem create too many search alternatives for the planners. The red and white track marks the path followed by the penguin during the plan execution.

course, but also the generation of PDDL problems and the execution of plans and probably above all, the coupling of all these procedures to the game must not slow down the game. The next section digs our game playing architecture and, in particular, the on line generation of PDDL problems.

6 Game Playing Architecture

This section outlines the plan execution procedure and the generation of PDDL planning problem from Iceblox game situations. Finally we discuss path planning in the Iceblox domain.

6.1 The Plan Execution Process

Table 2 outlines the plan execution process; it is implemented as a thread and can give orders to the game through a global variable which usually contains the key just pressed by the user.

The tricky part is the compiling of the PDDL actions of the plan returned by the PDDL-based planner into basic instructions (the keys pressed by the user) for the game. This allows a regular sensing of danger and gives regular opportunities to response to unexpected events.

6.2 The Generation of PDDL Planning Problems

Table 3 presents how the collection-of-a-single-coin is generated as a PDDL planning problem during game playing. This very fast procedure, also implemented as a thread,

Table 2. The plan execution thread

```
Repeat
  If the first operator of the plan is Move-To-Crossroad($i$,$j$) then
    Compute a path from current location to crossroad$_{i,j}$
    Repeat
      Execute one basic move following that path
    Until crossroad$_{i,j}$ is reached
  Else
    If the ice block at crossroad$_{i,j}$ is not destroyed
      Execute a push action in direction of crossroad$_{i,j}$
    Else
      Execute a basic move towards crossroad$_{i,j}$
    End if
  End if
  Delete the first operator of the plan

  When the flame is no longer dangerous
    Ignore it
  When the flame is no longer aligned with the weapon or
      an unexpected flame becomes dangerous
    Find a weapon and use it
  When the penguin must leave the computed path
    Remember the current crossroad$_{i,j}$
    Get the penguin back to crossroad$_{i,j}$ as soon as possible
Until the plan is empty
```

Table 3. The PDDL Planning problem generation thread for the collection of a single coin

```
Document-the-path-to(object)
  Compute a path for this object
  Generate (reachable-cell i-object j-object)
  If this path is clear then
    Generate (protected-cell i-object j-object)
  Else
    For each ice block on this path do
      Generate (blocked-by-cell i-object j-object i-block j-block)
      If this ice block can be moved with a simple push then
        Generate (pushable-cell i-block j-block i-stop j-stop)
      Else   /* this ice block cannot move and must be crushed */
      Generate (crushable-cell i-block j-block i-crush j-crush)
      End if
    End for each
  End if
End Document-the-path-to

Generate PDDL planning problem header
Generate PDDL initial state header
Generate (position i-penguin j-penguin)
Find the (nearest or possibly flame-less) iced coin
Generate (iced-coin i-coin j-coin)
Document-the-path-to(iced-coin)
For each flame do
  If this flame is less than four crossroads away from the (iced-coin i-coin j-coin) then
    Generate (guard flame i-flame j-flame i-coin j-coin)
    For each nearest weapon in each of the four direction do
      Generate (weapon i-push j-push i-weapon j-weapon i-stop j-stop)
      Document-the-path-to(weapon)
    End for each
  End if
End for
Generate PDDL final state header and (extracted i-coin j-coin)
```

first computes what to do (either push or else crush) with ice blocks on its path to an iced coin. Then it finds weapons by looking in the four directions around a dangerous flame while managing the ice blocks it encounters on its path to these weapons.

6.3 Path Planning

Path planning is obviously used to compute paths from a location to another. This path planning procedure is first called during the generation of PDDL planning problem (see table 3) and then during the plan execution process (see table 2): path planning is an essential activity.

We implemented the A* [20] algorithm with ideas from [21]: for instance, we added a penalty for each direction change to the cost of a path in order to get more rectilinear paths, following our requirements on actions of kind 3 (see section 3 above). However, Iceblox is not a free path world and destructive operations must happen to find paths: searching on empty crossroads is not sufficient to find paths. Pushing and crushing ice blocks often are necessary actions to find paths, but both modify the game level in an irreversible manner: for the sake of completeness of the A* algorithm, modified game levels must be saved as new search spaces for path planning, which obviously costs memory and therefore time.

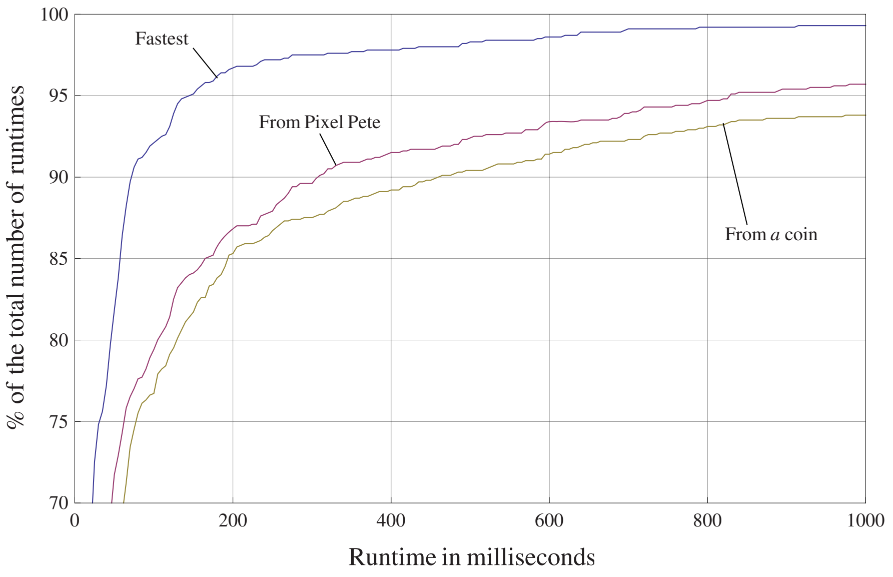


Fig. 9. On the horizontal axis are runtimes (milliseconds) from our implementation of A* which computes paths in Iceblox randomly generated levels (10 lines by 12 columns); Pixel Pete appears in the top left corner, and 9 rocks, 5 coins and 35 pushable ice blocks are randomly placed in this order on the remaining cells. This makes an amazing 10 342 621 660 587 151 106 372 657 654 037 758 770 942 528 117 680 different game levels (see section 6.3). The cost of a path is the cost of actions (this cost is 1 for a basic move in each of the four directions, 1 for pushing an ice block and 7 for crushing an ice block) plus the number of corners (i.e. direction changes) plus the manhattan distance to the goal position. 200 levels were randomly generated; we recorded both the runtime to compute a path from one coin to Pixel Pete (*the lowest curve*) and from Pixel Pete to the same coin (*the second curve*); the five coins of each level were tested. Consequently, 1000 paths have been computed in each of the two directions. How fast (or slow?) are these computations? As we can observe in the above figure, 85% of the computations from a coin took less than 2OO milli-seconds whereas 87% of the computations from Pixel Pete took less than 200ms; so should we prefer to compute a path always from Pixel Pete? The answer is no: the computations from Pixel Pete produced 513 of the fastest runtimes while the computations from a coin produced 547 of the fastest runtimes (60 tests produced the same runtime). But despite these 547 fastest runtimes, *the lowest curve* in the above figure shows that it generally takes a little more time to compute a path from a coin than from Pixel Pete. However, if we choose the fastest runtime of the two available runtimes for each coin, it appears that 97% of the computations took less than 200ms which is just fine (*the highest curve*).

The figure 9 shows the proportions (vertical axis) of runtimes (horizontal axis) in milli-seconds above a given runtime for paths computed from Pixel Pete to a coin (*the middle curve*) and from a coin to Pixel Pete (*the lowest curve*). For instance, 95% of the runtimes are below 800ms, which clearly is unsatisfactory for real-time playability. If we look for path planning runtimes equivalent to PDDL-based plan construction runtimes, we observe that around 86% of the runtimes are below 200ms (which is not very fast yet). 1000 paths (in both ways: 2000 paths in total) have been computed over 200 generated levels where Pixel Pete appears in line 1 and column 1, 9 blocks, 5 coins and 35 ice blocks are placed at random on the remaining cells, in this order. The initial position of the penguin is fixed so we first have to choose 9 cells among the 119 remaining cells to place the rocks, then choose 5 cells among the 110 remainings cells and finally choose 35 cells among the 105 remaining cells; let lc be number of lines times the number of columns, we have:

$$\binom{(lc-1)}{9} \times \binom{(lc-10)}{5} \times \binom{(lc-15)}{35} = \frac{(lc-1)!}{5!\,9!\,35!\,(lc-50)!}$$

This total number of such levels has to do with astronomy and we leave the computations to the numbers lovers; it is, however, not difficult to remark that 200 generated levels is a desperately small number compared to the numbers provided by the above formula. However, it seems that more tests only reinforce the observed results of figure 9.

By taking the minimum value of the two available runtimes for each coin, we get the *highest-curve* in figure 9, where 97% of these minimum runtimes are less than 200 ms. Consequently, we are left with several options: (i) let the two computations (from Pixel Pete and from a coin) run concurrently and take the first returned path, (ii) search in a destructive manner (whatever the path), (iii) investigate a bi-directional search procedure, (iv) spend some time optimizing any step of our A* implementation, in the spirit of [22] and (v) focus on something else. After failing to gain much with the third and fourth options, perhaps strangely, we chose the fifth option: it happens that, finally, the gaming experience is not much affected by a somewhat slow but nicely correctly implemented path planning procedure. Now, we advise the wise programmer to first investigate option (ii) before trying option (i).

7 Conclusions

This paper reports on the design and implementation of a PDDL-based planning system (problem generator, planner and plan execution) which, most of the time, is able to play an arcade game in real-time. We detailed engineering decisions because this research project showed us that an efficient planner itself (PDDL-based or not) certainly is not sufficient to achieve real-time playability. In the case of an arcade game, everything must be fast; or, if preferred, nothing must be slow. To avoid slowing the game, many planning knowledge oriented decisions must be taken: which predicates must be designed to represent a gaming situation? What is the balance between detailed planning operators and a clever plan execution system? What can be put in a thread? What part of planning computations can be distributed over multiple graphic frames? This paper proposes answers to these questions, mainly from an engineering perspective and not from a theoretical perspective: decisions have, most of the time, been made in favour of the game playability.

Due to the good quality of the path planning computation, Pixel Pete looks pretty rational at the speed of the flames. Sincerely, we thought many times we'd never make it and the granularity of our Plans is crucial to achieve our goal: we just produce Planning problems with a small number of predicates. Since then, we have reused our architecture in the commercial serious game domain; we successfully coupled our planning system to Virtual Battle Space 2 [23] [24], in a bottom-up manner. We are actually working on extending this coupling.

Acknowledgements. This work is part of a 3 year project founded by the Fondation Saint-Cyr. Thanks to Jeremy Buisson for his help on multithreading. Thanks to Rick Alterman, Marc Cavazza, Jon Gratch, David Kirsh, Carlos Linarès, Jeff Orkin and Adrian Smith for helpful discussions since the beginning of this project.

References

1. O'Brien, J.: A flexible goal-based planning architecture, pp. 375–383. Charles River Media (2002)
2. Orkin, J.: Applying goal-oriented action planning to games. In: Rabin, S. (ed.) AI Game Programming Wisdom 2, pp. 217–227. Charles River Media (2003)
3. International Planning Competition (1998–2009), `http://ipc.icaps-conference.org/`
4. Gerevini, A., Haslum, P., Long, D., Saetti, A., Dimopoulos, Y.: Deterministic planning in the fifth international planning competition: Pddl3 and experimental evaluation of the planners. AI Journal 173, 619–668 (2009)
5. Hornell, K.: Iceblox (1996), `http://www.javaonthebrain.com/java/iceblox/`
6. Bartlett, N., Simkin, S., Stranc, C.: Java Game Programming. Coriolis Group Books (1996)
7. Wikipedia: Pengo, arcade game (2007), `http://www.wikipedia.org/`
8. Orkin, J.: Three States and a Plan: The A.I. of F.E.A.R. In: Proceedings of the Game Developper Conference, 17 pages (2006)
9. Hoffmann, J.: FF: The Fast-Forward planning system. AI Magazine 22(3), 57–62 (2001)
10. Bartheye, O., Jacopin, É.: New results for arithmetic constraints partial order planning. In: 24th Workshop of the UK Planning and Scheduling Special Interest Group, London, UK, December 15-16 (2005)
11. Nareyek, A., Fourer, R., Giunchiglia, E., Goldman, R., Kautz, H., Rintanen, J., Tate, A.: Constraints and AI Planning. IEEE Intelligent Systems, 62–72 (March/April 2005)
12. Bonet, B., Geffner, H.: Heuristic search planner 2.0. AI Magazine 22(3), 77–80 (2001)
13. Hoffman, J.: The metric-ff planning system: Translating "ignoring delete list" to numeric state variables. Journal of AI Research 20, 291–341 (2003)
14. Chen, Y., Wah, B., Hsu, C.W.: Temporal planning using subgoal partioning and resolution in sgplan. Journal of AI Research 26, 323–369 (2006)
15. Long, D., Fox, M.: Efficient implementation of the plan graph in stan. Journal of AI Research 10, 87–115 (1999)
16. Coles, A., Fox, M., Long, D., Smith, A.: Planning with respect to an existing schedule of events. In: Proceedings of the 17th ICAPS. AAAI Press, Menlo Park (2007)
17. Vidal, V.: A lookahead strategy for heuristic search planning. In: Proceedings of the 14th ICAPS, pp. 150–159. AAAI Press, Menlo Park (2004)
18. Bartheye, O., Jacopin, É.: Planning as a software component: a report from the trenches. In: 26th Workshop of the UK Planning and Scheduling Special Interest Group, Prague, CZ, December 17-18 (2007)
19. Microsoft: Directx 9 (2009), `http://msdn.microsoft.com/en-us/directx`

20. Pearl, J.: Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley, Reading (1984)
21. Rabin, S.: A* aesthetic optimizations. In: Deloura, M. (ed.) Game Programming Gems, pp. 264–271. Charles River Media (2000)
22. Rabin, S.: A* speed optimizations. In: Deloura, M. (ed.) Game Programming Gems, pp. 272–287. Charles River Media (2000)
23. Bartheye, O., Jacopin, É.: A planning plug-in for virtual battle space2: A report from the trenches. In: Proceedings of the Spring Simulation Multiconference, 4 pages (2009)
24. Bartheye, O., Jacopin, É.: A real-time pddl-based planning component for video games. In: Proceedings of 5th AIIDE. AAAI Press, Menlo Park (to appear, 2009)