University of Camerino

School of Science and Technology

Faculty of Computer Science (Class LM-18)

In collaboration with

Reykjavík University

Automatic Abstraction and Refinement for Simulations with Adaptive Level of Detail

Graduand
Michelangelo Diamanti
Number 098858

Supervisor
David James Thue
Assistant Professor
Reykjavík University, Iceland

Co-supervisor
Andrea Polini
Associate Professor
University of Camerino, Italy

A.Y. 2017/2018

Abstract

Optimizing the level of detail of a simulation involves avoiding the computation of unnecessary features, if doing so is transparent to one or more observers. This concept dates back to graphic level of detail which has already been thoroughly studied, and there are various techniques that adjust the rendering quality. Non trivial simulations, in addition to the graphic component, also have a model that is responsible for the behavior of intelligent agents in the environment. Modern computing allows simulations to be always larger and more complex, thus requiring an ever growing amount of resources to function. In this dissertation we propose a system that is able to decompose a simulation in different layers of abstraction, adjust the level of detail of the simulation according to the observer's perception of the world, and present the resulting visual representation to the user. In particular, my contribution to the system revolves around devising ways of understanding when to switch between different levels of detail, and how to perform the adjustment without altering the consistency of the simulation with regards to the user's perception of the environment.

Contents

1. Introduction

The graphic component of simulations has continuously been targeted as a crucial point of optimization. Every modern graphic engine employs a wide set of techniques aimed at automatically adjusting the level of detail of the rendered objects, in order to maximize the performance of the simulation, while providing as much graphic details as possible.

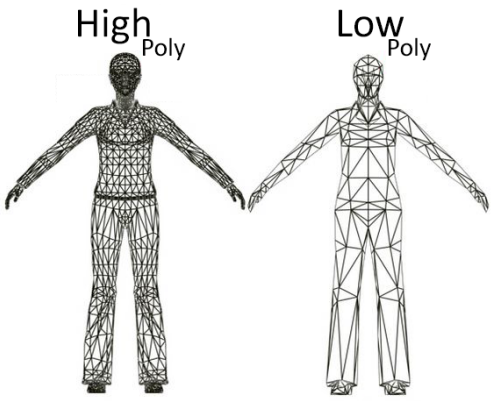


Figure 1.1: High vs Low poly.

For example, the polygons that constitute the model of every object in a graphic simulated environment depend on the distance from where such a model is being rendered. Fewer polygons means less detail, but also easier computation and rendering requirements. If the distance between the player and the model is enough, the difference becomes unnoticeable.

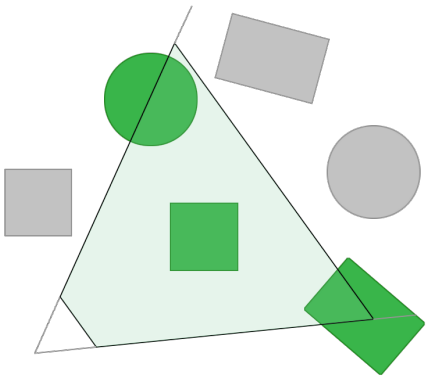


Figure 1.2: Frustum Culling.

Another technique often employed is frustum culling, which uses the camera field of view to check which objects are visible by the player:

- everything inside the frustum gets rendered in the scene
- everything outside of the frustum doesn't get rendered and its model gets deallocated to save resources.

Simulations are not constituted only by their graphic component. In fact, behind almost everything that's happening in the environment, there is an algorithm that decides its behaviour. Although equally important, this aspect of level of detail has not received the same research focus as its graphic counterpart. For this reason, the progress that has been made with regard to simulation level of detail remains limited.

Authors	LOD based on	Number of LODs	LOD applied to	AI behaviors
Chenney et al. [1]	potential visibility	2	updating movement	navigation, collision avoidance
O'Sullivan e al. [2]	distance	not specified	geometry, animations, collision avoidance, gestures and facial expressions, action selection	navigation, collision avoidance, complex dialogs with other agents
Niederberger and Gross [3]	distance and visibility	21	scheduling, collision avoidance, path planning, group decisions	navigation, collision avoidance
Brom et al. [4]	simplified distance	4	action selection, environment simplification	navigation, complex interactions with objects and other agents
Paris et al. [5]	distance	3	navigation, collision avoidance	path planning, navigation, collision avoidance
Lin and Pan [6]	distance	not specified	geometry, animations	locomotion
Osborne and Dickinson [7]	distance	not specified	navigation, flocking, group decisions	navigation
Kistler et al. [8]	distance and visibility	10	updating movement, collision avoidance, navigation, action execution	navigation, collision avoidance, desire-based interactions with agents and smart objects, dialogs

Table 1.1: LoD for simulation in various topics reported by Kistler et al. [8].

Even if there has been some effort to advance simulation LoD adjustment, the results remain limited to their own scopes. Referring to Table 1.1, we notice that the fields are very similar to one another, and their applications are narrow. For example, there are many approaches that deal with agents' navigation, animations of virtual humans and crowd behaviours. We can also notice that, quite often, the abstraction levels are in a fixed and pre-determined number.

In our work, we want to propose a more general approach that can be employed in many different scenarios to adjust the level of detail of the various components of a simulation. To do that, we introduce a system that helps the user design simulations that are capable of self-adjusting their level of detail.

There are a number of key points that make the self adjustment of simulation LoD much more difficult compared to graphic optimization:

- **Different Implementations of Simulations**: There are many different ways of modeling the simulation of an environment. Each one of them is unique, thus requires its own way of adjusting the level of detail. This greatly hinders the reusability of LoD self adjustment techniques among different implementations. For example, if I wanted to simulate a crowd behaviour, I could not reuse the self adjusting LoD crowd simulation system of Assassin's Creed Unity[9], unless the implementation of the crowds is same. This is not true for graphic optimization, where there is a strict standard rendering pipeline common to every engine,

meaning that various techniques can be easily ported between them.

- **Context-Specific Optimization**: since simulation LoD deals with semantics rather than visual props, there is no easy way of finding which parts are best to optimize, nor which metrics can serve as good indicators for switching the level of detail. For example, while dealing with graphic optimization, we can heavily rely on rendering distance, and simply employ one of the techniques seen in Figures 1.1 and 1.2. On the other hand, when dealing with simulation LoD, there is no such clear distinction.

1.1 Background

Complex simulations can have many connected parts that interact with and influence one another with varying degrees of magnitude. Narrative-focused world simulations require that the state of the story, as well as the state of the simulated world, be maintained with consistency so as not to challenge the user's suspension of disbelief. Simulating narrative-focused world spaces thus requires a balance to be maintained between system resources and the consistency of the simulated world.

The work explained here is part of a larger project that focuses on consistent world space Simulation through Level of Detail manipulation for narrative-driven worlds introduced by Ólafsson [10]. The project aims at developing a system composed by many components that interact with one another to create and handle massive simulations. The main components are as follows.

- The **World Generator** receives updates to the narrative from a narrative generation system and expands or modifies the World Data.
- The **World Data** represents the state of the world, its entities, and their connections to each other.
- The **World Manager** manages the Level of Detail of the World Data as the World State is passed to the other systems.
- The **World Simulator**, operating at various levels of detail, manages the behaviour of entities represented by the World Data, making updates to the World Data in response to the connections between entities or to player actions.
- The **World Visualizer** arranges the scene depending on the World Data and the player's position in the world space.
- The **Listener** takes in feedback from player actions and feeds it to both the Narrative Generator and the World Simulator to process.

In particular, my contributions will focus on the world manager and world simulators, as I will propose a method for self-adjusting the level of detail of the simulation, along with an approach for modeling the simulation itself. Both contributions can be integrated into the larger system and interact with the other parts to provide a complete simulation experience.

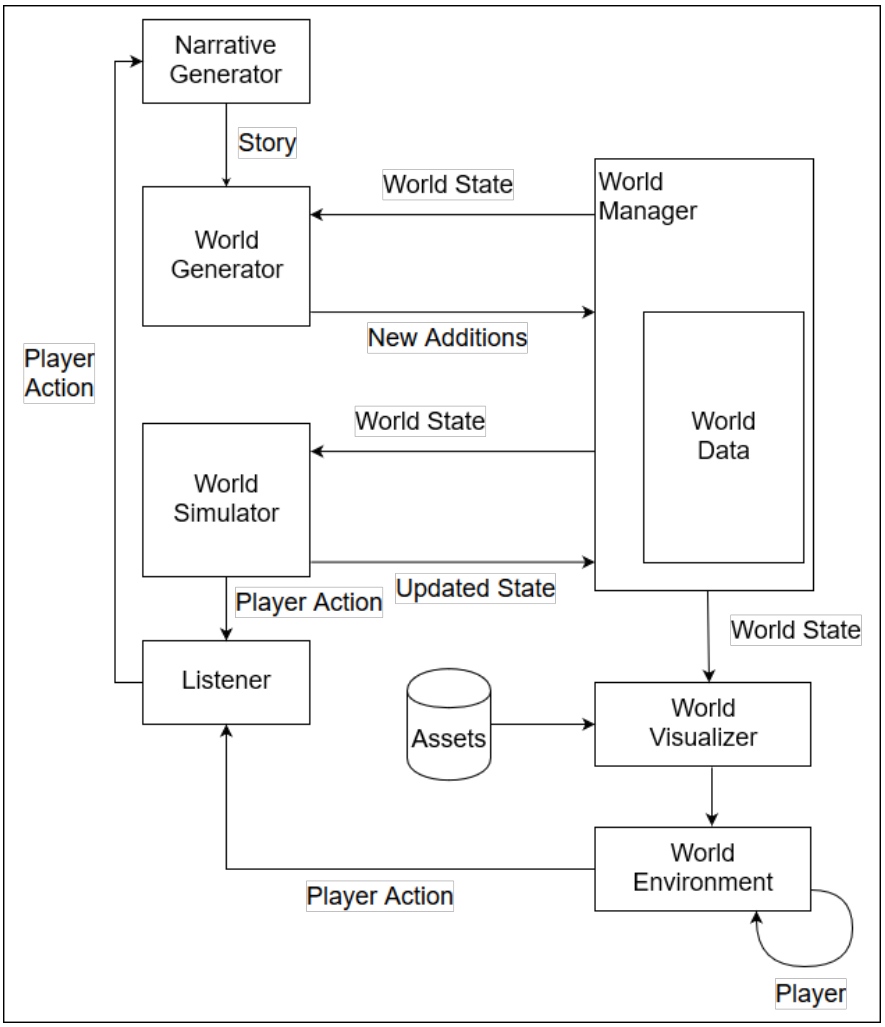


Figure 1.3: General view of the system [10].

My proposed approach for tackling the challenges described in Section 1 is based on two components:

- **A Standard framework** that enables the modeling of general-purpose simulations. The chosen language on which the framework is based on is the Planning Domain Definition Language (PDDL). It is a well-known language considered to be a standard among Artificial Intelligence Planning languages. Having a standard way of modeling simulations enables us to reuse the techniques for self-adjusting level of detail.

- **A LoD algorithm** that, given many models for a simulation of a process representing different level of abstraction, is able to adjust its level of detail. The algorithm should always run the simulation at the level that requires fewest resources, while still providing enough details. When we switch level of detail, the algorithm should make sure that every transition previously computed is compliant with the constraints of the new detail level. When there are some discrepancies, the algorithm should re-write part of the history to fix the inconsistencies.

2. Problem Formulation

While experiencing a large world simulation, there are aspects that cannot be perceived by the user because of many factors such as distance, occlusion, or even lack of interest in that particular component.

Even so, most of the time, if we want to keep the flow of the simulation consistent, we are bound to be accounting for every aspect of the environment at all time. In fact, different events may be tied to one another, and the lack of simulation of one part of the environment may affect some other part that is actually observable by the user.

The problem that I have investigated in this project is: how can we adjust the level of detail of simulations to postpone the computation of unnecessary details if they're not observable by the user, while maintaining a consistent simulated environment? Since the simulation is interactive, its progression may result in a state where the system needs to provide more details than the ones that had been previously computed. Imagine that we are simulating the movement of an agent at abstract level because the user cannot directly observe it. If at a later time the user has access to a recording of the agent, the movements that were previously computed at abstract level need to be refined to account for the current observable details. This means that the system needs to monitor some indicators in order to adjust the level of detail of the simulation. When a change in level of detail is deemed necessary by the system, we need to review part of the simulated past and make up for the lack of details in the previously computed story-flow. The main advantage of delaying the computation of details is twofold:

- **Avoid unnecessary details**: If the simulation never reaches a state that compels us to provide more details, then we completely avoided unnecessary computation.

- **Computation Scaffolding**: When we actually need to generate details, we already have a skeleton that depicts a high level structure upon which we can base our computation, thus greatly reducing the size of the problem.

Definition *Domain Abstraction*: Given two domains, δ_1 and δ_2, we say that δ_1 is an abstraction of δ_2 if every possible outcome reachable by applying the transition model defined by δ_1 is also reachable by applying the transition model of δ_2

Let D be a list of domains $(\delta_1, \delta_2, ..., \delta_n)$ such that for all $< \delta_i, \delta_{i+1} >$ pairs $\in$ D, δ_i is an abstraction of δ_{i+1}. Our goal is to find the $\delta_x \in D$ with lowest x such that using δ_x to run the simulation will preserve perceptual consistency for the user.

2.1 Standard Framework for Simulations

The first component we try to devise in this dissertation is a framework that enables the modeling of large simulations. There are multiple difficulties that arise while devising such a component because it must integrate with the rest of the system, and so it must satisfy multiple criteria:

- It should be flexible enough to allow the differentiation between multiple levels of detail.
- It should account for the presence of multiple intelligent agents, since general purpose simulations may contain more than one entity that is interacting with its surroundings. This problem alone is not trivial: it forces us to account for concurrency challenges during the simulation.
- It should be easily portable to other systems. If we want to reuse this method in other scenarios, it should be possible to incorporate the component in other systems.

2.2 Switch between Levels of Detail

We must devise an algorithm that estimates the observability of each event that is being simulated in the environment, with regard to the player's perception of the world. This measure will then be used to adjust the level of detail of the simulation, to strike a balance between detail level and required resources.

The main challenge in this regard is to choose good indicators that can be employed to compute the observability estimate. They should be impartial enough to allow for deterministic measurement, while still being relevant measures of observability.

2.3 Increasing Level of Detail

We must devise an algorithm that, when the estimated observability level is above a certain threshold, is able to increase the level of detail of the simulation to match the user's new perception of the environment. It could be the case that new rules and mechanics apply to the increased level of detail, and the resulting constraints could have been neglected while applying the previous transition model. This means that the system might have generated some actions that would not have been possible in the new level of detail. In this case we need to be able to parse each abstract transition into a series of actions whose combined effects lead to an equivalent goal state.

In this regard, we must account for concurrency challenges such as synchronization between different agents. In fact, it could be the case that multiple transitions are dependent on one another and they must happen in the correct sequence.

2.4 Decreasing Level of Detail

Analogously, we have to devise a method for switching from a detailed simulation state to an abstract one. During this process we must drop all the aspects of the simulation that are not observable by the user, while still maintaining the relevant ones.

2.5 Criteria for Success

While adjusting the level of detail of a simulation, we should keep in mind the following criteria

- **Consistency**: the simulated story flow should be consistent at all times so as not to challenge the user's suspension of disbelief.
- **Level of Detail**: the simulation should provide enough details to account for every aspect of the world that the user can perceive.
- **Observability Switch**: the observability measure employed to estimate the user's world perception must be adequate. The switch in level of detail should correctly reflect this estimate.
- **Performance**: the system should have adequate performance. Adaptive level of detail should result in saving resources compared to always simulating at the highest possible level of detail.

A good solution should satisfy all the above requirements. In particular, consistency means that if the user perceives a particular aspect of the simulation, and then puts them in a scenario in which they can observe more details of the same aspect, the system must never output details that contradict previous perceptions.

For example, if the user sees an agent that is moving from a location to another at level of detail 1, then triggers the switch to the second level of detail where the same movement is not possible (e.g. due to an obstacle that was not simulated), the system should refine that action with a sequence of equivalent transitions that are also compliant to the previously observed details.

3. Related Work

3.1 Proxy Simulations for Efficient Dynamics

Chenney, Arikan, and Forsyth [1] present an approach for decreasing the motion level of detail for objects that are not perceivable by the user. The authors define the term "proxy simulation" as the procedure responsible for managing the objects that are out of scope, with a lower level of detail. The paper compares two slightly different designs for proxy simulations: both save resources employing discrete event computation, but the second also integrates a statistical model to account for interactions that are otherwise expensive to simulate.

The example scenario is a simulation of the cars in traffic: all the objects that might be observable by the player (street level) are simulated at full LoD, the others with a proxy simulation.

3.2 Geometry Motion and Behaviour Level of Detail

The paper "Crowd and Group Simulation with Levels of Detail for Geometry Motion and Behaviour"[2] written by O'Sullivan et al. focuses on modeling the level of detail for groups of virtual humans. It introduces a way of adjusting the LoD of simulations with regards to three aspects: geometry, motion and behaviour. The geometry is the complexity of the models employed to represents the agents; motion is the level of detail of the algorithms used to control the agents' locomotion or grasp movements; and behaviour can involve both spoken cues, such as language accent, or non spoken cues such as gaze animations that express emotions.

Geometry complexity is handled by reducing the count of polygons. Motion is adjusted by switching from pre-determined animation to full featured inverse kinematics movements. Behaviour is changed by modifying the number of possible cues that express speech and emotions.

3.3 LoD for Cognitive Real-time Characters

Niederberger and Gross [3] present a scheduling algorithm that divides the time allocated for computing the behaviour of each agent depending on its observability. There are two types of behaviour: proactive, which grants the agents full awareness of their surroundings and lets them choose their own actions accordingly, and reactive, which is a cheap behaviour that just tries to avoid inconsistencies.

The objects are collected into a quadtree which is updated using the camera's field of view. They're divided in three levels: visible; near-visible if they have at least one

edge attached to a visible one; non-visible. The scheduling algorithm makes sure that the reactive model is applied to every object, and reserves all the remaining time to run the proactive model on the more important agents.

3.4 Collision Avoidance LoD for Large crowds

Paris, Gerdelan, and O'Sullivan [5] discuss the simulation of large crowd of virtual humans. It focuses on the issues that arise in path finding and locomotion algorithms, such as obstacle avoidance, when many agents have to be simulated at once.

The authors devised three different levels of detail associated with as many path finding algorithms. At the most detailed level, the path is computed with full obstacle avoidance, taking into account the entire entity's perception of its immediate surrounding. The second level only takes into account one potential obstacle and quickly retrieves a solution using fuzzy logic. The third level doesn't have obstacle avoidance at all. The level of detail of each agent is adjusted based on the distance from the player.

3.5 Grouped Hierarchical Flocks of Agents

The paper from "Improving Games AI Performance Using Grouped Hierarchical Level of Detail"[7] by Osborne, Dickinson, et al. focuses on simplifying the LoD of simulations that are composed by hierarchical structured groups of intelligent agents. The authors have developed a simulation of many separate groups of similar entities, comparable to a flock, where each group may contain an arbitrary number of agents.

The LoD is adjusted based on the distance from the player. When the detail level is decreased, an arbitrary portion of the elements of the furthest groups is merged into a single entity. This may save a lot of resources because the functions that regulates the behaviour of the agents needs to be applied to fewer units. When the LoD is increased, the single entities get expanded becoming many agents.

3.6 Switching Level of Detail

The paper "Level of Detail Based Behavior Control for Virtual Characters"[8] by Kistler, Wißner, and André focuses on providing a general way for switching the level of detail of simulations based on a formula that takes into account multiple parameters such as distance from the camera and occlusion.

The solution allows for an arbitrary number of different LoDs. Starting from the most basic one, each one of them adds some complexity, until the full detailed model.

3.7 Planning in a Hierarchy of Abstraction Spaces

The paper Planning in a Hierarchy of Abstraction Spaces [11], published by Sacerdoti in 1973 dives into the difficulties of devising efficient plans for complex environments. The work explained in the paper also serves as introduction for ABSTRIPS (Abstraction-Based Stanford Research Institute Problem Solver) which, along with classic STRIPS, is one of the predecessors of the PDDL syntax.

The core concept presented in the paper is that one can greatly increase the performance of classic planners by reasoning about the problem at hand on different levels of abstraction. The planner starts by differentiating between important parts of the problem and the ones that can be regarded as (trivial) details. In this regard the levels of abstraction are represented by domains, and the author assumes that each domain can differ from the others only for the number of literals (predicates). This restriction is justified as an effort to strike a good balance between abstraction power and similarity between the resulting domains, so that the plans devised at high level are still relevant once they need to account for details at ground level, and don't need to be revised too much.

Computing abstraction spaces requires an initial assignment of predicate importance by the author. Then, a static analysis is performed to differentiate between predicates that can be changed by the effects of actions which are regarded as details, and the ones that are immutable and thus fundamental to the problem.

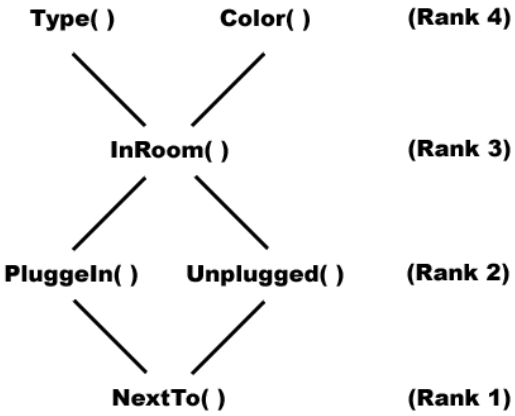


Figure 3.1: Abstraction Spaces [11].

The scenario in Figure 3.1 represents a planning problem of turning off the lamp in a room. The predicates in the environment are used to identify the object's physical properties (type, color etc.), and the spacial characteristics of the agent that must execute the plan (position, proximity etc.). A possible predicate ranking is shown, which is computed based on the effects of the actions of the domain.

Since the agent has many ways of going next to the lamp, this is regarded as a detail (rank 1). On the other hand it cannot alter the physical properties of the lamp thus the type and color are considered high priority.

With the ranking at hand, it is possible to compute many levels of abstraction: the more general ones will only account for important aspects of the problems, while the others will account for progressive amount of details. While devising a plan with this approach it is important to commit as late as possible to decisions about details so that it is possible to refine the plan at a later time while maintaining many different possible solutions, thus avoiding dead ends, which are plans possible at high level but not at ground level.

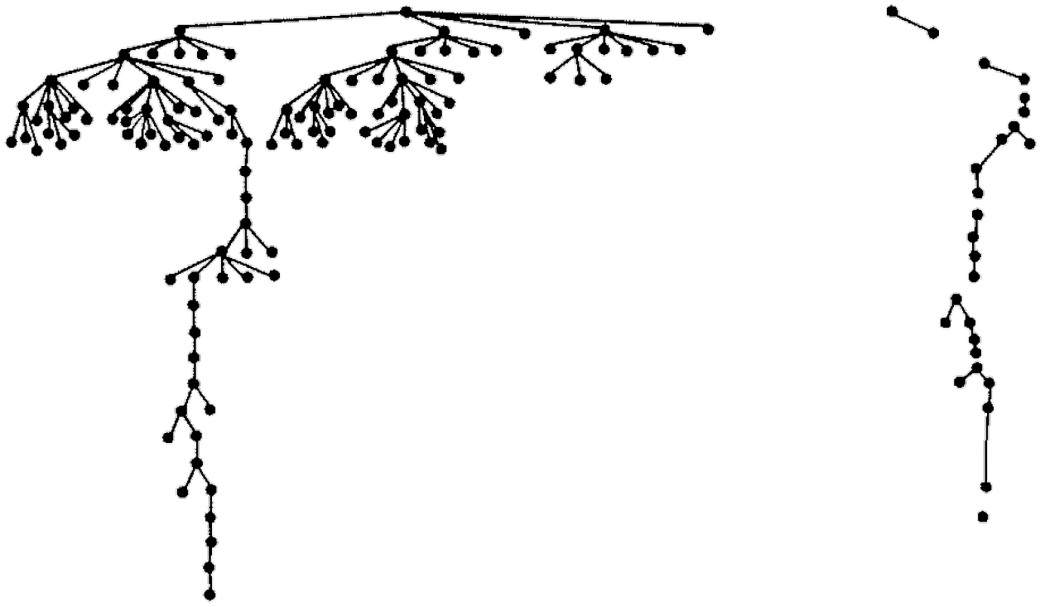


Figure 3.2: Left search tree with classic STRIPS, right ABSTRIPS [11].

We can see that formulating plans at an abstract state and then refining the details afterwards greatly reduces the size of the state space and the size of the search tree. Once the high-level plan is returned it is simply necessary to fill the gaps between the various parts.

The drawback with this approach is given by the strict constraints put on the abstraction domain: the abstraction spaces differ from the ground spaces only in the level of detail used to specify the preconditions of operators. This constraint helps a lot when taking into account effective heuristics because they can prune most branches of the search tree if we don't have to attend many details. On the other hand it is very constrictive, as it forbids us from devising new actions in the domain that would make traversing the search tree much faster.

There is also a fundamental difference that does not allow us to employ this approach to our use case: while planning an activity, one has a clearly set goal, but in our case, we are simulating a process and the end goal is simply to strike a good balance between performance and attention to details. Thus, we can allow more freedom in our abstraction spaces rather then focusing too much on effective heuristics.

3.8 Simulation Level of Detail for Virtual Humans

The paper Simulation Level of Detail for Virtual Humans[4] by Brom, Šerý, and Poch, presents a way of adjusting the level of detail of both spacial components and behavioural components of simulations.

The approach is based on the concept that, if spaces are organized hierarchically, they can be divided in multiple layers of abstraction. The example scenario in the paper simulates miners, sitting at a pub's table, enjoying some drinks. The hierarchy is structured as follows: the pub is located in a suburb, that is in a city residing in a kingdom of the world $pub \longrightarrow suburb \longrightarrow city \longrightarrow kingdom$.

The hierarchy is represented with a tree-structure: the leaves are the most detailed places, which are also called "way-places", for example tables at the pub, all the nodes that are not leaves are referred to as "Areas". Thanks to this hierarchy the authors devised a way of adjusting the LoD of the objects in the space with something that they called the "Membrane method".

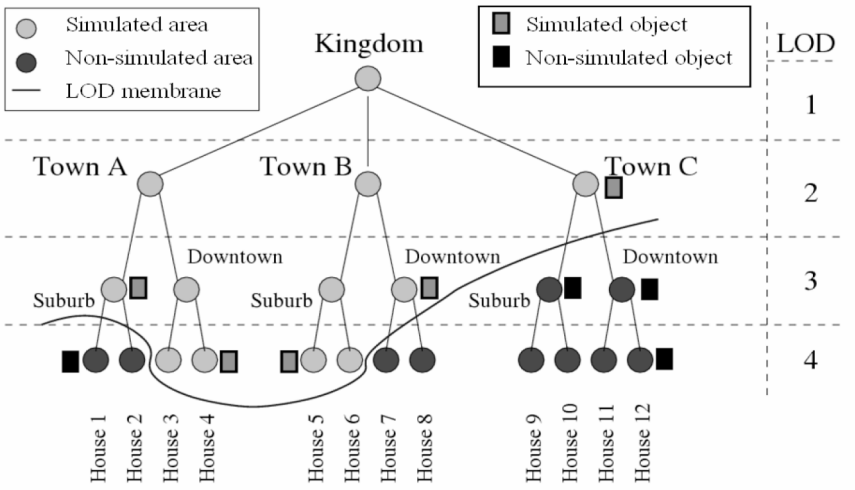


Figure 3.3: The Spacial Membrane cutting trough the layers of the tree [4].

The membrane is a virtual graphical representation of the level of detail. It can be visualized as a line that cuts trough various levels of the tree structure, and defines the boundary of what is observable in the world. Everything which is below the membrane is deemed not perceivable, while the objects that are over the membrane are observable.

The membrane is used to adjust the spatial LoD. It is employed to decide what happens for things that are below it: they can either cease to exist, or can be assigned to some other entity that is over the membrane.

All the objects in the world have a manually assigned "existence-level", which regulates the minimum required LoD for their existence, and a "view-level" that indicates their starting LoD. Every increase in level of detail requires to rearrange the space: objects that remember their position from before the transition are placed in the same spots, the others are assigned by room-specific placing algorithms.

When an object moves to a new area, one of three things can happen: if the object is going to an area that is below its existence level, it disappears; if the area LoD is above the existence and view level of the object, it is just placed there; finally if it is in between the two, the LoD of the area is increased to match the view level of the object.

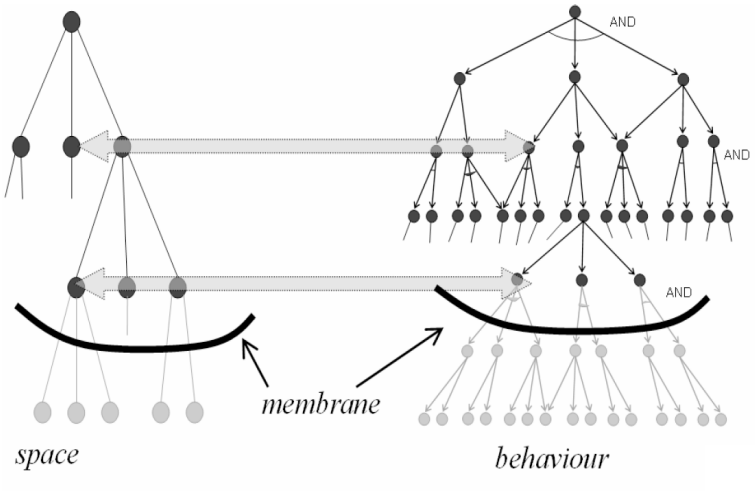


Figure 3.4: The behavior tree compared with the space hierarchy [4].

To describe the behaviour of the agents they employ AND-OR trees where goals are OR nodes and tasks are AND nodes. Every goal can be achieved by several tasks, while every task can be achieved by adopting sub-goals. The AND-OR tree is superimposed over the spatial hierarchy tree to understand which tasks are affordable at every layer. For example, if we're simulating the room of the pub, a possible action could be "enjoying at a table", while if we're only simulating the pub as a leaf node, the closest performable action would be "enjoying at a pub".

The membrane is re-shaped in a bubble around the user, or also by a drama manager that may decide to focus on a particular location. This does not immediately affect the objects that are below the membrane; instead, they employ a "garbage collector" that deallocates them only if needed (low resources).

This approach presents some limitations:

- A hierarchical space division is needed to benefit from this approach. The membrane would be flat if we were to employ it on an environment characterized by a non hierarchical structure, for example a racing game where all the different locations might be at the same level.
- The membrane method requires a lot of human annotation. We need to assign "view-level" and "existence level" to manage how the membrane affects the entities of the world.
- The "Room specific placing algorithms", used to reshape the space when refining an area, greatly hinder the reusability of the solution, as they have to be developed ad hoc for each area.

As we hinted in the introduction, the main limitation with the aforementioned approaches is that they're limited to their own scopes. Many of them concentrate on simulating virtual humans, especially crowds, or other sorts of groups that can be managed with flocks behaviours. This limits each solution's reusability, as it would be hard to employ it for different scenarios.

Another important difference between these approaches and our own is in the number of levels of detail: the majority of these methods provide a fixed and predetermined number of layers. Moreover, they often require the abstractions to be designed ad-hoc.

With our approach we try to mitigate these limitations by providing a system which is capable of: (*i*) creating subsequent abstraction models for an input detailed problem; (*ii*) providing a method that estimates the best time for switching among the various LoD; (*iii*) and designing a way of accounting for the challenges that arise when refining and abstracting the simulation data among different levels of detail.

4. Proposed Approach

In this section we will explain the work that has been done to answer our scientific questions presented in Section 2. We first introduce and describe the language that we chose to model the simulation and the resulting framework that we implemented. We then talk about how we can use the framework to model self-adjusting level of detail simulations. We then articulate the algorithm for increasing or decreasing the level of detail. Finally, we explain how we estimated observability and how we employed that measurement to switch between the various levels of detail.

4.1 Planning Domain Definition Language

The Planning Domain Definition Language (PDDL) is an attempt to standardize Artificial Intelligence (AI) planning languages. A classic problem expressed in PDDL has two main components:

- **A Domain Description**: contains the elements which are common across all instances of the problem that we are modeling, including the *predicates* that can be used to describe the *relations* of the world and the *actions* that each agent can perform.
- **A Problem Description**: the actual instance of the problem that we are going to plan a solution for. It contains all the *entities* that are part of the problem and the relations that describe their current state. It also contains a description of the starting state and the goal state.

Consider a simplified version of the actual problem that we modeled in our project: we have a rover which has been assigned the task to move from a starting location to its destination. First we define the types of entities which are part of the problem:

	Rover	**Waypoint**
Description	the type of the protagonist of the environment, the one that performs actions	the type of the locations where the rover can go

Table 4.1: Entity Types in the example environment.

Then we define the keywords that we are going to use to describe the world, which are called predicates:

	CAN_MOVE	**BEEN_AT**	**AT**
Description	specifies if the rover can move from a starting waypoint to a destination.	specifies if the rover has been in a certain location	the current location of the rover

Table 4.2: Predicates in the example environment.

Finally we must describe the actions that the entities can perform, in this case the only one is *move*, which moves the rover from a location to another:

	MOVE
Parameters	rover, startingWayPoint, destinationWaypoint
Preconditions	rover AT startingWayPoint startingWayPoint CAN_MOVE destinationWayPoint
Postconditions	**not** rover AT startingWayPoint rover AT destinationWayPoint rover BEEN_AT destinationWayPoint

Table 4.3: Move Action Example.

- **The Parameters**: are the rover, the starting waypoint and the destination
- **The preconditions**: are relations that must be true in the state to which the action is applied in order to perform it. In this case the rover current location must be equal to the starting point of the action, and it must be possible to move from start to destination.
- **The postconditions**: is a set of relations that define how the resulting state looks like after applying the action. After the move action the rover location is changed from start to destination and we know that it has been at this new location.

Once the domain has been modeled we have to create an instance of the actual problem. In order to do that we need to populate the world with objects of the types specified in the domain, describe how the initial state looks like using relations composed by the aforementioned predicates and specify what is our goal state.

Referring to Figure 4.1, we can see on the left the domain of the problem with the predicates that can be used to describe the world and the possible actions. On the right we see the instance of the current problem, which describes the entities, the relations, the starting state and the goal state.

```
(define (domain rover-world)

(:predicates
  (can-move ?from-waypoint
      ?to-waypoint)
  (been-at ?rover ?waypoint)
  (at ?rover ?waypoint)
  (waypoint ?waypoint)
  (rover ?rover)
)

(:action move
  :parameters
    (?rover
     ?from-waypoint
     ?to-waypoint)

  :precondition
    (and
    (rover ?rover)
    (waypoint ?from-waypoint)
    (waypoint ?to-waypoint)
    (at ?rover ?from-waypoint)
    (can-move ?from-waypoint
        ?to-waypoint))

  :effect
    (and
    (at ?rover ?to-waypoint)
    (been-at ?rover ?to-waypoint)
    (not (at ?rover
        ?from-waypoint)))
)
```

```
(define (problem rover-1)

  (:domain
    rover-domain
  )

  (:objects
    waypoint1 waypoint2 waypoint3
    rover1
  )

  (:init

    (waypoint waypoint1)
    (waypoint waypoint2)
    (waypoint waypoint3)

    (can-move waypoint1 waypoint2)
    (can-move waypoint2 waypoint3)
    (can-move waypoint3 waypoint1)

    (rover rover1)

    (at rover1 waypoint1)
  )

  (:goal (at rover1 waypoint3))
)
```

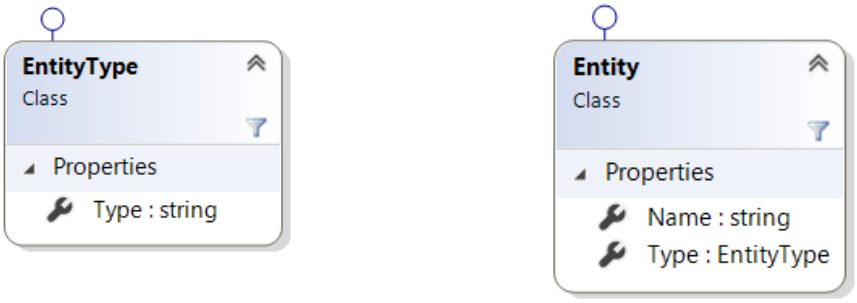
Figure 4.1: PDDL Syntax Example.

4.1.1 PDDL for multi agent environments

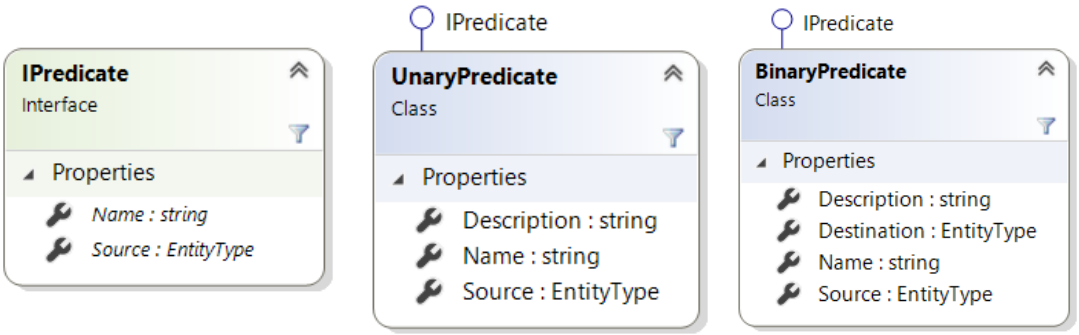
The actions defined in the domain of a PDDL problem do not discriminate which of their parameters is the one that is actually performing the action: for the *MOVE* action in the example in Figure 4.1 we can see that its parameters are *rover*, *from-waypoint* and *to-waypoint*. From this list of parameters a planner has no way of knowing who is performing the move action; it could be the rover as well as the waypoints.

That is because in a simple planning problem we only have one entity performing the actions, which in this case is the rover, and so we can avoid specifying it in the actions. But if we want to model an environment that contains more than one active entity, for example a situation with two rovers instead of one, we need to specify who is the **subject** and who is the **object** or recipient.

4.1.2 Implementation

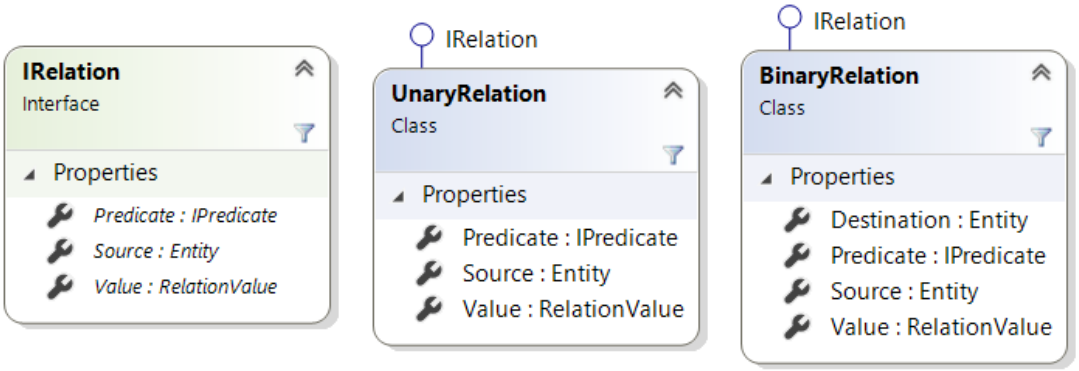


The *EntityType* class is used to express the kind of entities that we can find in our problem. In Figure 4.1 we can see that it is part of the constraints and thus needs to be specified whenever we need to use an entity. Making it a class of the framework resulted in the authoring process being more convenient, since we can then declare typed entities and avoid repetition. The *Entity* class represents the actual entities that are part of our simulation problem; they must be of a specific type and have a name.



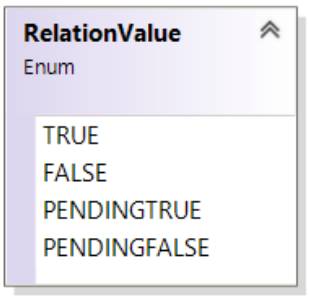
As we already mentioned, a *predicate* is a keyword we use to describe the environment in the PDDL problem. *IPredicate* is an interface that acts as a template showing what a predicate looks like in our framework. The basic elements that a *predicate* must have are a name and an entity type.

Referring to our example in Figure 4.1, we can see that we have a collection of predicates in the domain: the first one, *can-move*, indicates that the rover can go from a starting point to a destination. This means that we can use the keyword can-move to describe a feature in our environment, namely that two waypoints can be traversed.

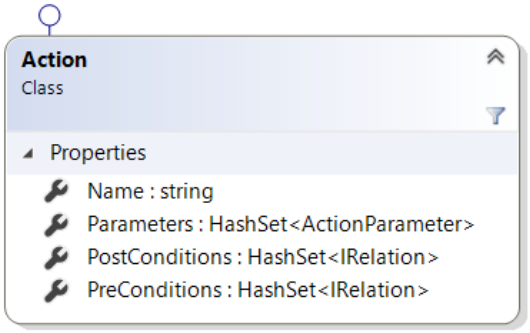


Relations can either be used to describe a property of some entity, mostly in the case of unary relations, or connections in the environment between two entities described by

binary relations. Our example in Figure 4.1 employs relations in the problem description. There we can see the association *can-move wp1 wp2*, indicating the connection between two locations in the environment. Upon instantiation the constructor checks if the entities that have been passed are of the same *EntityType* specified in the predicates: if we are using the keyword *can-move* to specify the connection between two locations, then the entities must be of type waypoint, otherwise we would violate authoring constraints.



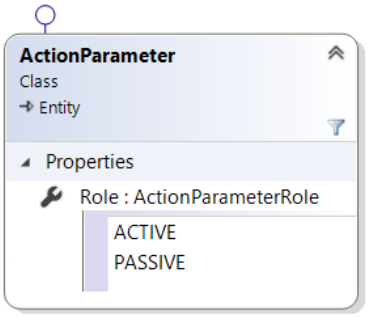
The relation value is an enumeration used to limit the values that each relation can hold. We didn't want to limit them only to true and false so that we could maintain some flexibility between simulation and visualization of the environment. The actions will have a certain duration, which is the period of time needed for them to be visualized. During this time, the relations modified by the action are neither true nor false because their value is being changed; thus we can say that they're pending.



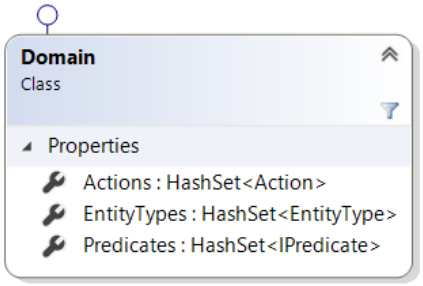
Actions define how the entities can interact with their surrounding environment as well as with other entities. They are defined by a name, parameters, preconditions and postconditions. The parameter list contains the entities that are manipulated by the action. The preconditions are the relations that must be respected in order for the action to be applicable, and the postconditions are the relations that describe how the action affects the world state.

When a new action is defined, the constructor checks if the relations specified in its conditions are compliant with the entities specified in the parameters: this is to ensure self consistency of the action class. Imagine if, in the move action, we stated that it must be possible to go from waypoint1 to waypoint2 but then in the parameters we refer to them with the names waypointx and waypointy. We could no longer discriminate between the two, and the behaviour of the action would be unpredictable: we could request to move from X to Y and end up actually going from Y to X.

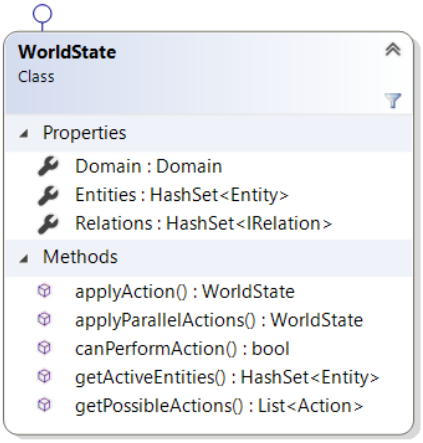
We use an *explicit* collection of parameters (instead of just computing one from the entities contained in the conditions) because this way we give more control to the authors: if they are forced to explicitly declare each parameter of the actions, then we can better inform them about inconsistencies among relations. For example, if they mistakenly give different names to the same entity in two separate conditions, but only one of them appears in the parameters, then we can detect the inconsistency and raise an exception that prevents unintended system behaviours.



As mentioned in Section 4.1.1, we had to slightly modify the standard PDDL framework in order to account for multiple intelligent agents in the environment. This resulted in the creation of the class *ActionParameter* which is an extension of the *Entity* class. This class adds a property called *Role*, an enumeration whose value can be either *active* or *passive*. This lets us distinguish between the subject of the action and their objects.



The *Domain* contains all the *types*, *predicates* and *actions* of our problem. When a new domain is instantiated it first inserts all the input entity-types into a collection. Then it iterates over the input predicates and adds them only if they're compliant with the previously declared types. If they're not consistent we raise an exception. The same constraint is put on actions. They can be added to the domain only as long as they refer to existent types and predicates.



The *WorldState* represents a possible state of an actual instance of the problem. It refers to a specific *Domain* and has both a set of entities and relations. Upon instantiation, the constructor checks if the given entities are of a type defined in the domain. For checking the relations' consistency, it makes sure that the predicates are defined in the domain, and also that the entities are part of the current instance of the world-state.

In this class we also have some relevant methods that are used during the simulation process:

- *getPossibleActions*(): computes a list of all the actions defined in the domain that can be applied to the current world state.
- *getActiveEntities*(): computes a shallow list of the entities that can perform at least one action in the current world state.
- *applyAction*(*Action a*): applies the action a to the current world state, if it is applicable, and returns the resulting world state with the modified relations. If it is not possible to apply the action to the current world state, it throws an exception.
- *applyParallelActions*($HashSet < Action > actions$): applies a set of actions to the current world state and returns the resulting state.
- *canPerformAction*(*Action a*): checks if the action a can be performed in the current world state. Returns true if positive, false otherwise.

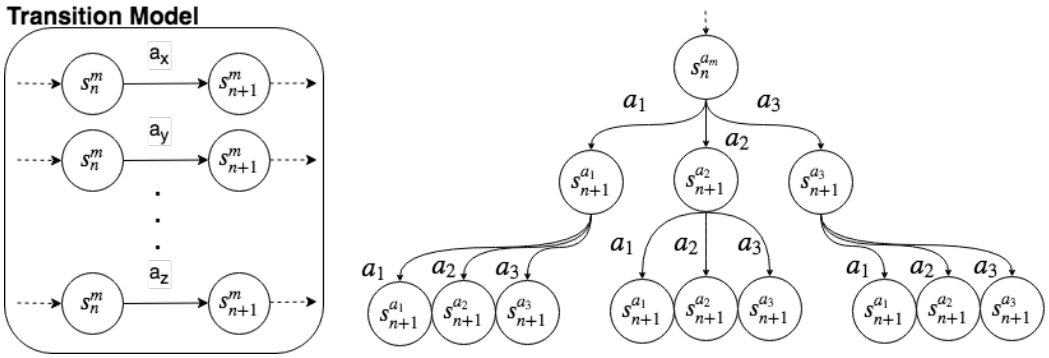
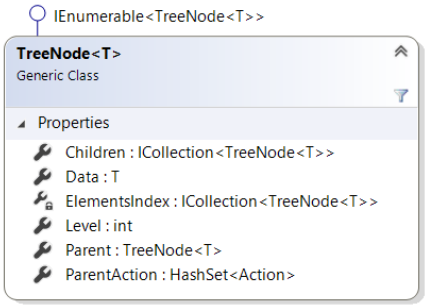


Figure 4.2: Tree structure representing the simulated environment.

We represented the simulation using a tree structure. A tree is an undirected graph in which any two vertices are connected by exactly one edge. vertices nodes contain the world data, while the edges that connect them are the actions from the transition model. This is common practice for AI planning problems as tree structures have thoroughly been studied, and there are many different search algorithms that can be applied to traverse them.



The *TreeNode* is the class that contains the WorldStates and connects them one another. Level is an integer that stores the depth of the current node in the tree structure, Data contains the actual WorldState, Children is a collection of nodes directly connected to this one at *level + 1* and Parent is the node directly connected at *level - 1*. ParentActions is a collections of parallel actions that were performed in order to reach this node from the previous one.

4.1.3 Optimization

HashSet is an unordered collection that contains unique elements, thus implicitly avoiding repetition. It provides all the usual operations offered by collections such as Add, Remove, Contains, but since it uses a hash-based implementation, their time complexity is O(1) as opposed to other memory structures such as List, which is O(n). HashSet also provides standard set operations such as Union, Intersection, and Symmetric difference [12].

In all our use cases, we want to preserve the **uniqueness of the elements**. For example, we can't allow the presence of multiple equal relations in the same world state, because that would create inconsistencies, such as in the case of the rover's position: it would be inadmissible to define it twice or more if they have different values. We also require **fast lookups** of the elements because in some cases, such as when generating the possible actions, we perform heavy computations that access the collections very frequently. Finally, with this kind of memory structure we are able to perform some **set operations** in an efficient and convenient way: if we want to merge two actions into a single one we can just compute the union of the collections without iterating.

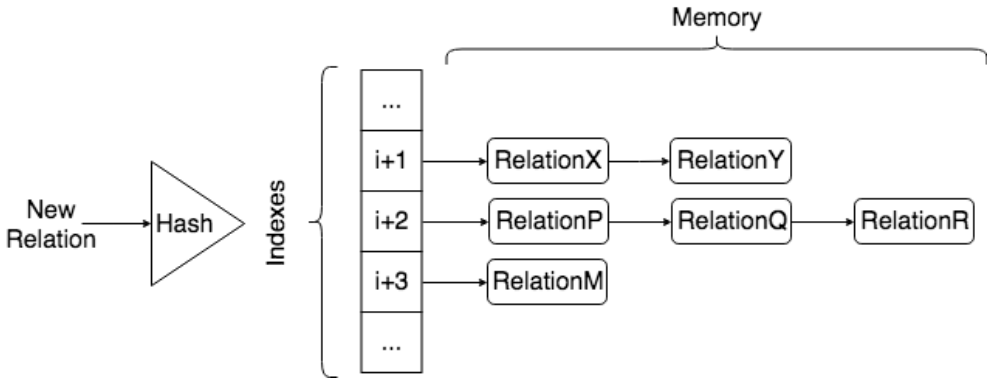


Figure 4.3: An example of how the HashSet is managed internally.

A hash function is any function that can be used to map data of arbitrary size to data of a fixed size. Hash functions are used for storing data in structures such as hash tables [13], to quickly locate a data record given its search key. Specifically, the hash function is used to map the search key to a list; the index gives the place in the hash table where the corresponding record should be stored.

A collision occurs when we want to access an element in the collection which has the same key as another already defined. In this case, we have to iterate over the list of items and look for the exact element. For this reason we had to implement the functions *getHashCode()* and *equals()* for each of our classes in the PDDL framework.

4.1.4 Unit Testing

Since this framework is the core component upon which we based all the modeling and methods for adjusting the level of detail of the simulation, it needs to be reliable and to behave correctly; this is why we unit tested all the classes. In every constructor, we tested the edge cases for the passed parameters. For example, when adding objects to the domain, we check that they're consistent with everything that's already there, and raise an exception if there's a violation of the constraints.

The more complex classes were subject to additional testing. For example, the method that returns the possible actions contained in the class WorldState has specific tests aimed at ensuring that it returns the correct actions.

4.2 Simulation

Once we have the description of the problem in PDDL syntax, we need an algorithm that uses that information to carry out a simulation of the resulting environment.

This process is a constant loop between planning and visualization: the simulator chooses an action from the transition model according to the current world state and then instructs the visualizer to perform it. The visualizer is responsible for showing the action to the user as well as detecting any error and interaction that may occur during the process. After the visualization cycle is completed, the simulator will receive an acknowledgement that will inform it about the outcome of the action: if it was successfully visualized we move forward in the simulation; otherwise, we roll back.

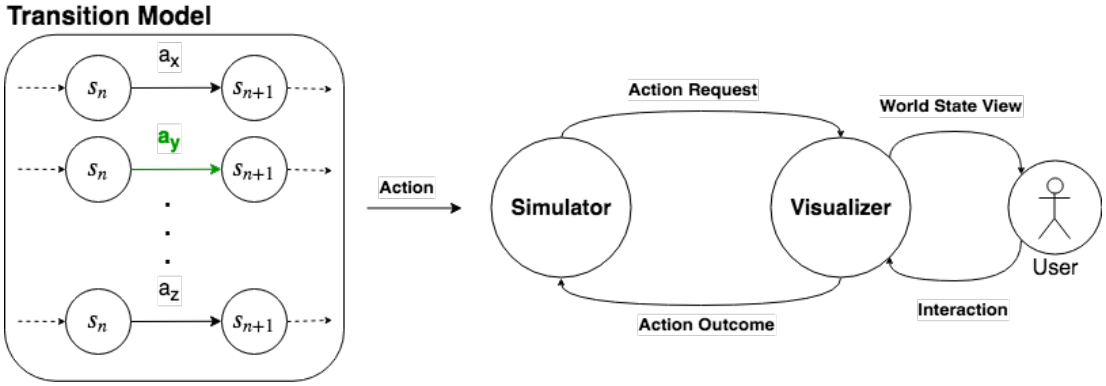


Figure 4.4: A general view of how the simulation-visualization cycle unfolds.

To choose an action from the transition model, we need to first compute a list of actions that are allowed in the current world state: an action is deemed performable if the entities in its parameter are part of the entities of the world state and all its preconditions are satisfied by relations in the current configuration.

In Figure 4.5, we can observe a domain that contains an action: move. This action moves the rover from a starting point to a destination (if they are connected). The world state contains two rovers and three waypoints, so there is a certain number of possible combinations in which this action could be performed in the current state. In order for an action to be performable, all its preconditions must be satisfied by the state to which it is applied. Thus, the allowed combinations are the ones in which

Domain

Action *Move*:
Params:
Rover r
Waypoint start
Waypoint dest
Preconditions:
r AT start
start CONNECTED dest
Postcondition:
r AT dest

World State

Entities:
Rover rov1
Rover rov2
Waypoint way1
Waypoint way2
Waypoint way3

Relations:
rov1 AT way1
rov2 AT way2
way1 CONNECTED way2
way3 CONNECTED way3

Admissible Actions:

Move(rov1, way1, way2) Move(rov2, way2, way3)

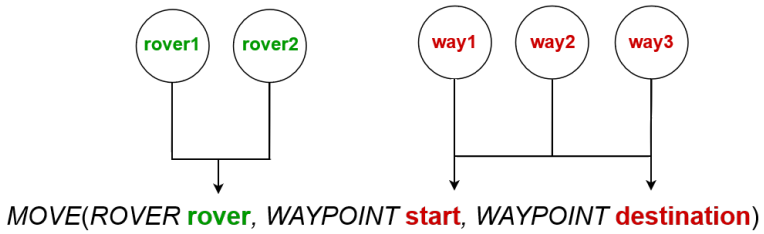
Inadmissible Actions:

Move(rov1, way1, way1) Move(rov2, way1, way1)
Move(rov1, way1, way3) Move(rov2, way1, way2)
Move(rov1, way2, way1) Move(rov2, way1, way3)
Move(rov1, way2, way2) Move(rov2, way2, way1)
Move(rov1, way2, way3) Move(rov2, way2, way2)
Move(rov1, way3, way1) Move(rov2, way3, way1)
Move(rov1, way3, way2) Move(rov2, way3, way2)
Move(rov1, way3, way3) Move(rov2, way3, way3)

Figure 4.5: An example of the admissible and inadmissible actions.

the rover is AT the starting location, and the starting point is CONNECTED to the destination. In the example the only combinations that satisfy the preconditions are $MOVE(rov1, way1, way2)$ and $MOVE(rov2, way2, way3)$. On the other hand there is a high number of combinations that do not satisfy the preconditions and are thus inadmissible. For example $MOVE(rov1, way1, way3)$ cannot be performed since *waypoint1* and *waypoint3* are not connected. To compute a list of all the performable actions, we need to compute all the admissible combinations and check which ones are applicable to the current world state. We do this with a combinatorial algorithm:

- For each parameter of the action we compute a list of entities in the world state that can be candidate substitutions. An entity can symbolize a parameter in a certain action if the two are of the same type.



- We compute every candidate combination performing the Cartesian product of the resulting entities.

Move(rov1, way1, way1)	***Move(rov2, way1, way1)***
Move(rov1, way1, way2)	***Move(rov2, way1, way2)***
Move(rov1, way1, way3)	***Move(rov2, way1, way3)***
Move(rov1, way2, way1)	***Move(rov2, way2, way1)***
Move(rov1, way2, way2)	***Move(rov2, way2, way2)***
Move(rov1, way2, way3)	***Move(rov2, way2, way3)***
Move(rov1, way3, way1)	***Move(rov2, way3, way1)***
Move(rov1, way3, way2)	***Move(rov2, way3, way2)***
Move(rov1, way3, way3)	***Move(rov2, way3, way3)***

- Each combination represents an actual action. As opposed to the ones in the domain that are general actions and describe the behaviour of dummy parameters, these real actions have actual entities from the world state as parameters. This means that they can be applied to the world state, if they satisfy the constraints. We check if the preconditions of each action are satisfied by the relations of the world. If they are, the action can be applied; otherwise, it can't.

(rov1 AT way1 **AND** way1 CONNECTED way1) ? **FALSE**
(rov1 AT way1 **AND** way1 CONNECTED way2) ? **TRUE**
(rov1 AT way1 **AND** way1 CONNECTED way3) ? **FALSE**
...
(rov2 AT way1 **AND** way1 CONNECTED way3) ? **FALSE**
(rov2 AT way2 **AND** way2 CONNECTED way1) ? **FALSE**
(rov2 AT way2 **AND** way2 CONNECTED way3) ? **TRUE**
...
(rov2 AT way3 **AND** way3 CONNECTED way3) ? **FALSE**

After this process is complete, we are left with only the combinations that are applicable to the current world state. We then proceed to select one of them and fee it to the visualizer to advance the simulation. For now, the selection process is done assuming that all the actions are equally probable, so we randomly pick one of them. This could be modified to account for some probability distribution over the actions, given certain conditions.

4.3 Level of Detail Adjustment

As we already mentioned in previous chapters, the aim of this project is to maintain a balance between simulation accuracy and computational needs, adjusting the level of detail according to the perception that the user has of the current state of the world.

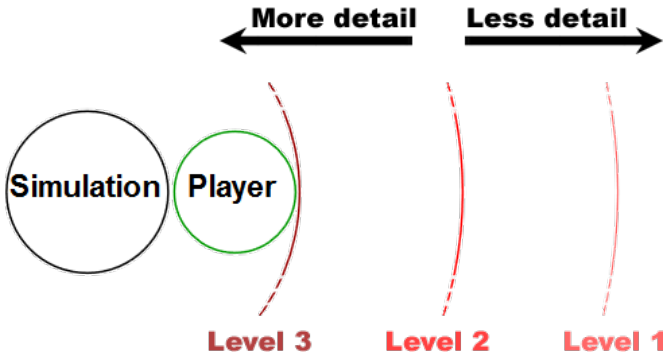


Figure 4.6: A possible representation of a simulation with three different levels of detail, each identified by a threshold. The player is in the most detailed level.

Referring to Figure 4.6 we should change the level of detail of the simulation whenever the player triggers one of the thresholds. A very important part of this process is to devise an algorithm that enables us to translate information between two models of different levels of detail; this process is called abstraction (decreasing detail) and refinement (increasing detail). Since we are employing a tree structure to store information about our world data, as explained in Section 4.1.2, I decided to implement a data structure that stores the last observed state of the world at each level of detail. This allowed us to have easy access to the complete game three while only maintaining references to the last observed nodes.

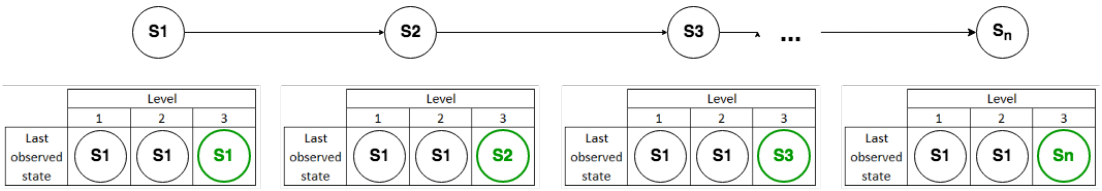


Figure 4.7: the contents of the data structure with references to the nodes at each level of detail if the player is within threshold 3.

In Figure 4.7, we can see that while the environment is being simulated at level of detail 3, only the corresponding reference in the data structure gets updated because we can ignore the advancements in other levels.

4.4 Abstraction

Abstraction is the process of translating information from a detailed model to a more abstract one. Performing abstraction means that we give up on some of the details of the current world state so that we can reduce the size of the state space, thus reducing the resources in terms of memory, computations and time needed to traverse it.

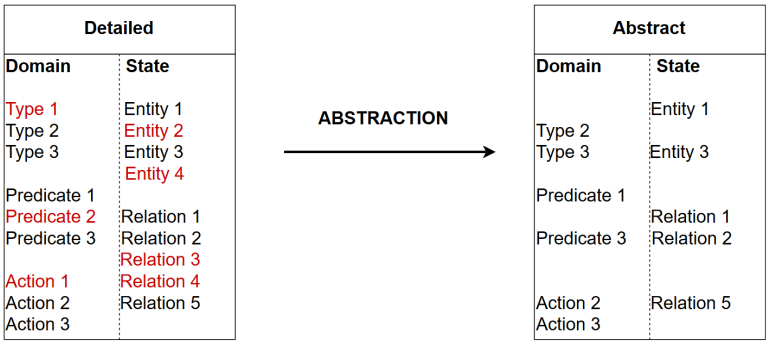


Figure 4.8: The figure shows how we abstract some details when parsing from a state with higher level of detail to one with less details.

To perform abstraction we rely on the domains: two states at different levels of detail could potentially hold the same entities and relations, but they must have a different domain. To translate between two states of different levels of detail, we begin by abstracting the domain. In this regard, we remove all the *types*, *predicates* and *actions* which appear in the detailed state's domain but not in the abstract one. Referring to Figure 4.8 we can see that *Type 1*, *Predicate 2* and *Action 1* are considered details while performing abstraction and so the resulting state will ignore them. This produces a cascade effect in the world state data, as all the entities of the types that are no longer in the domain cannot be part of the resulting state, and this effect propagates to relations as well.

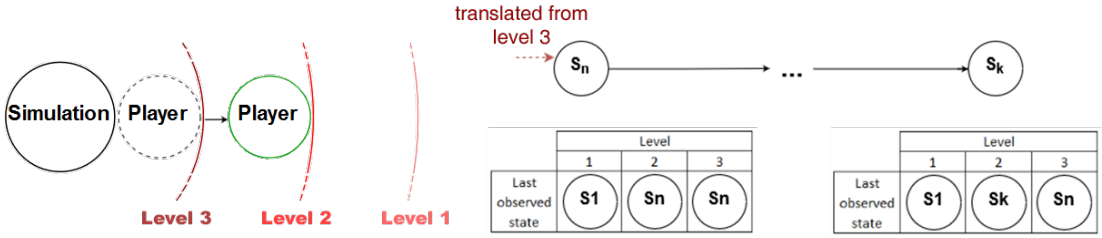


Figure 4.9: Update process of the of the last observed state at each level.

As we can observe in Figure 4.9, when the abstraction process is completed, we store the newly created world state in the corresponding position of the last observed state table. After that, we continue with the simulation at the new level of detail with the usual process of applying the transition model to the world state. After every iteration, we only update the table position corresponding with the detail level.

It is important to notice that maintaining this memory structure updated enables us to discard every reference to the simulated tree branches, because when we need them, we can get them back by traversing in reverse from the last node that we stored.

4.5 Refinement

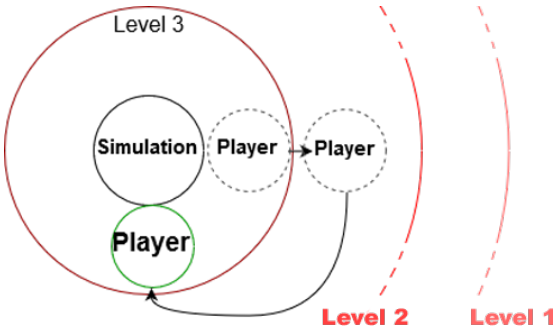


Figure 4.10: Player Triggers Refinement.

Refinement is the process of translating information from an abstract model to a more detailed model. This process is employed when the player triggers the switch from a general simulation to a more detailed one. Figure 4.10 shows the player going towards the simulation, thus triggering refinement.

If we reach a point in the simulation that requires us to provide more details than we had previously computed, we need to roll back history and devise a plausible explanation for the events that unfolded.

Algorithm 1: The subgoal search algorithm employed in the refinement process.

```
   Input  : A list of nodes at abstract level of detail
   Output: A list of detailed nodes equivalent to the given one
 1 currentNode: The node from the abstract simulation;
 2 lastNodeAtCurrentLevel: The last observed node at detailed level;
 3 while not at root node do
 4 |   currentNode ← Parent(currentNode);
 5 end
 6 while not at leaf node do
 7 |   currentNode ← Child(currentNode);
 8 |   solution ← Search(lastNodeAtCurrentLevel, currentNode);
 9 |   while solution is not root do
10 |   |   Add(result, solution);
11 |   |   solution ← Parent(solution);
12 |   end
13 end
14 Reverse(result);
```

We first roll back the simulation until we hit the root node: referring to algorithm 1 lines 3-5, we see that we do so by climbing back to the parent node. Since every time we perform abstraction we discard the current branch and only store the last node in the table, the root node we find will correspond to the last observed state at the previous level. After that, we start traversing the simulation tree (lines 6-7) and pairing each abstract action with a list of detailed ones, whose cumulative effect produce an equal resulting node. To do that, we employ a search algorithm (line 8) that takes in input the starting node at detailed level, and the desired node at abstract level (the input of the algorithm). The algorithm applies the detailed transition model to traverse the tree, and tries to reach a node, at detailed level, which is equal to the abstract one. An equal resulting node is one that contains all the effects of the abstract actions, while still also satisfying the constraints for the additional details.

For example, the refinement of a *move(entity,start,dest)* action should return a state

in which: the entity is in *dest*, and the chosen path for reaching the destination complies with the constraints of the new LoD.

Since the returned solution is a leaf node, we roll back and return the refined actions in the correct order.

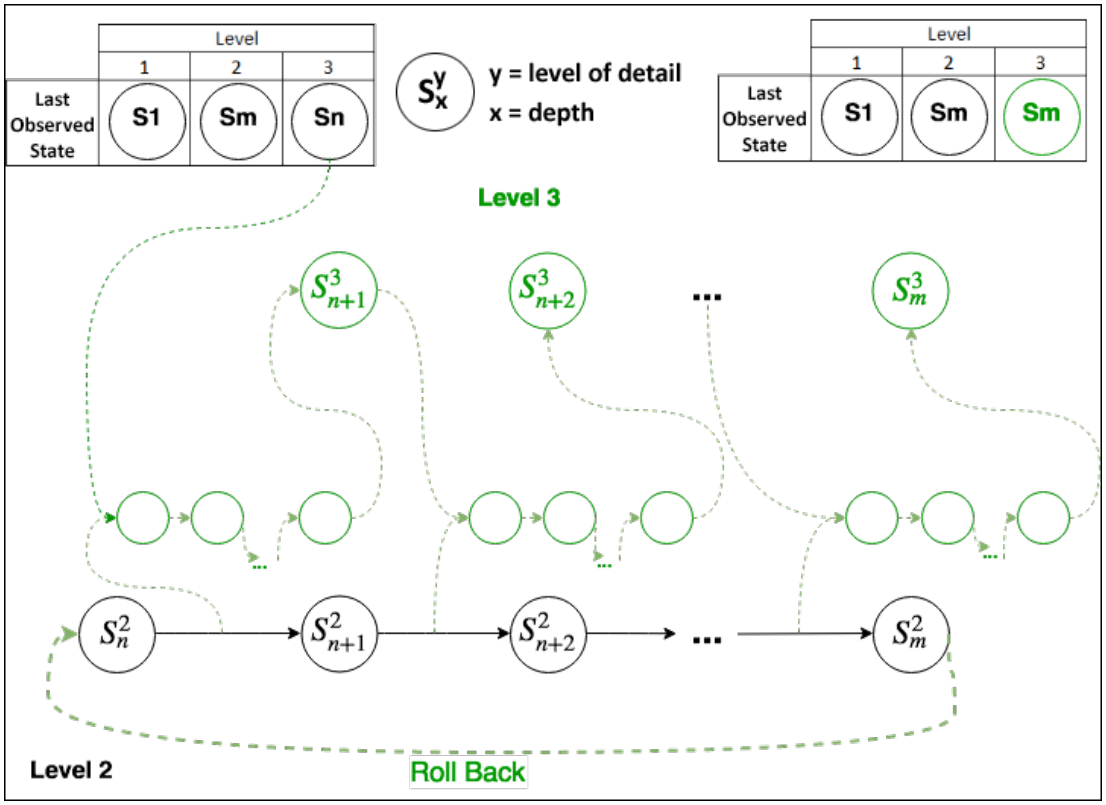


Figure 4.11: A visual representation of the refinement process from level 2 to level 3.

The roll back dotted line shows how we traverse the simulated branch at level 2 until we hit the root node S_n, which is the abstract version of the node contained in the table for level 3. After the roll back, we start pairing each node at level 2 with a list of nodes of level 3. We do so by feeding S_i^3 and S_k^2 into a search algorithm that parses the abstract action into a detailed plan. The most basic version of the search algorithm is a slight modification of the classic Breadth First Search.

The method is shown in algorithm 2, and it differs from the classic breadth first search for two parameters: *desiredAccuracy* represents how much the node should resemble the goal to be considered a solution to the search problem (line 4 and 18), and *cutoff*, which is the maximum depth that the algorithm can reach before returning a failure state (line 9); this is needed because under certain circumstances it is not possible to find a solution and the algorithm would loop forever.

Referring to algorithm 2, we begin by checking whether the input node is the goal state (lines 4-6). *Equal relations* is a function that computes an indicator of similarity between the input state and the goal state. It loops through the relations of the goal state and checks if they're contained in the input state as well. The result is a value $\in \mathbb{R}$ that is computed dividing the number of equal relations by the amount of relations of the input state, thus we always get a value between 0 (completely different) and 1 (equal).

After that, we start exploring the game tree by applying the transition model (actions) to the current node (lines 15-22). Each time we put all the nodes in the *frontier*, thus exploring it "horizontally". At the same time, we add them to an *explored* set (line 14), to avoid checking the same node twice. If we find a node that satisfies the goal test (line 18) we return it, otherwise, if the frontier is empty, we return a failure because we can't apply any more actions.

Algorithm 2: Search algorithm for refining abstract actions.

```
   Inputs : Initial state and goal state, desired accuracy, cutoff
   Output: A node holding the path to the refined solution plan
 1 function breadthFirstSearch(initialState, goalState, desiredAccuracy, cutoff):
 2 begin
 3     node ← a node with STATE = initialState;
 4     if equalRelations (goalState,node.STATE) ≥ desiredAccuracy then
 5         return node
 6     end
 7     frontier ← a queue with node as the only element
 8     explored ← an empty set
 9     while level (node ) < cutoff do
10         if empty (frontier ) then
11             return failure
12         end
13         node ← POP (frontier )
14         add (node.STATE, explored )
15         foreach action in actions (node.STATE) do
16             child ← child (node,action )
17             if child.STATE is not in explored or frontier then
18                 if equalRelations (goalState,node.STATE) ≥ desiredAccuracy
                    then
19                     return node
20                 end
21                 frontier ← insert (child, frontier )
22             end
23         end
24     end
25 end
```

Since the *equal relation* function always returns a real number $0 \leq x \leq 1$, we can easily use it to sort the frontier, thus turning the breadth-first search into a best-first search algorithm.

While breadth-first search explores the nodes in the frontier always following the order of discovery, best-first search inserts them into a queue sorted with a function that estimates how close they are to a solution.

We can see in Figure 4.12 that, in the case of breadth-first search, node B is discovered and also expanded first, while in best-first search, even if node B is discovered before node D, it is expanded last because it is very different from the solution to the problem.

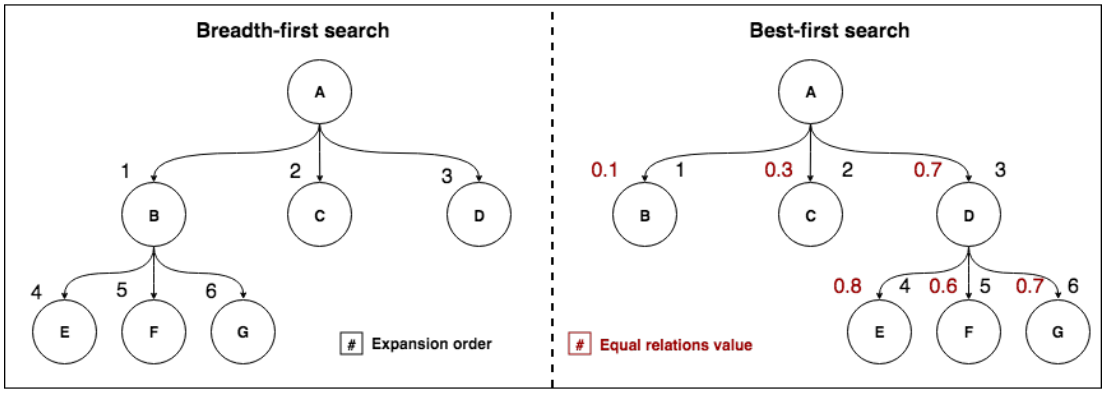


Figure 4.12: Comparison between breadth-first search and best-first search.

4.6 Multi Agents planning synchronization

Until now, we talked about the simulation process and the related abstraction and refinement assuming that there is only one entity which can perform actions in the world, but in reality that is not true. In fact, one could easily model a scenario in which multiple entities perform actions at once. In Section 4.1.1, we explained how we modified the PDDL framework to allow the modeling of multiple active entities. Now we will discuss how we accounted for it with regard to the simulation.

4.6.1 Concurrency

If we allow multiple agents to perform actions at the same time, we will face issues related to concurrency. Actions depend on preconditions, and their effects modify the current state of the world. Allowing multiple actions to be executed in parallel could result in inconsistent states, if they have conflicting effects.

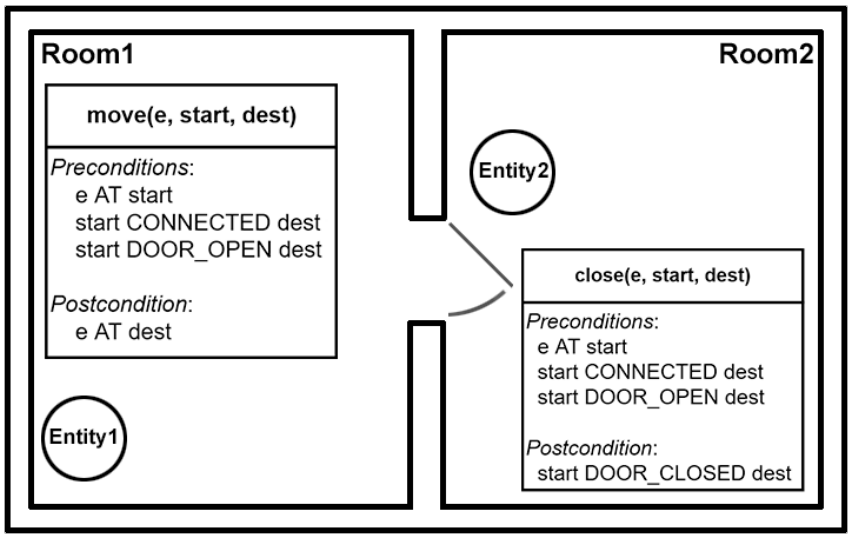


Figure 4.13: An example of two actions that can't happen safely at the same time.

We can see in Figure 4.13 that the two actions can't possibly happen at the same time. In fact, the effects of the action *close* alter the world state in a way that prevents the *move* action's preconditions to be satisfied. On the other hand, there are other combinations of actions that could be executed at the same time. For example, both entities could perform a move operation without conflicting with one another.

Definition 4.6.1
A set of actions can be executed in parallel if every sequence resulting from their permutation leads to a consistent world state.

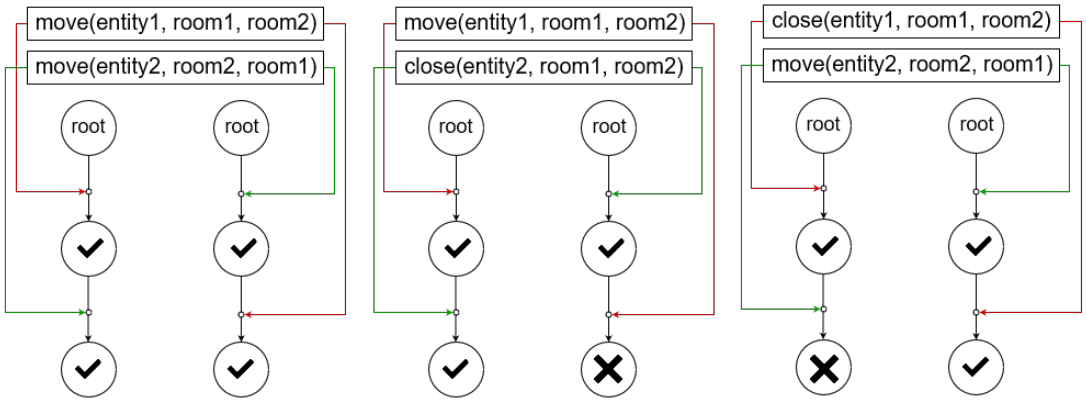


Figure 4.14: Permutations of action sequences.

The Figure 4.14 represents every action that is applicable to the world state of Figure 4.13. We can see that in this scenario every entity can perform two different actions: move from one room to the other, or close the door between the two. This means that there are three different possible combinations of the actions:

- **moving** *entity1* to *room2* and *entity2* to *room1*
- **moving** *entity1* to *room2* and letting *entity2* **close** the door
- letting *entity1* **close** the door and **moving** *entity2* from *room2* to *room1*

Now we need to check which ones of these combinations can be applied at the same time. To do that, we compute every possible permutation of actions for each set. Then we try to apply the resulting sequences to the current world state. If every action in each sequence is applicable and results in a consistent state, it means that they can be applied in parallel.

Figure 4.14 shows that we can have both entities change rooms in parallel, because the possible sequences of actions given by the permutations are:

- **move** *entity1* **to** *room2* **then move** *entity2* **to** *room1*
- **move** *entity2* **to** *room1* **then move** *entity1* **to** *room2*

Both sequences of actions are applicable to the starting state, never violate any condition of a subsequent action, and result in a consistent state.

On the other hand, if we look at any combination that contains a close action, we can see that its actions cannot be performed in parallel. That is because there is always at least one sequence that is not applicable to the starting state. For example, if we

wanted to **move** *entity1* to *room2* and **close** the door with *entity2*, we can observe that, if we first close the door and then try to move we cannot do so, because one precondition of the action move requires the door to be open.

The process for selecting one action, described in Figure 4.5, is modified to make use of the additional information provided by the extension of the PDDL framework in Section 4.1.1. In fact, we need to be able to discriminate between the action's subject and its objects in the parameters list, in order to assign one action to each active entity.

Algorithm 3: Selection algorithm for parallel actions.

Input : A set of applicable actions
Output: A tuple with one action for each active entity

```
1  possibleActions: list of actions applicable to the current worldstate
2  actionsForEachEntity: dictionary mapping each entity with its list of actions
3  parallelActions: list with possible actions for each entity
4  randomIndexes: indexes of parallel actions list in random order

5  possibleActions ← possibleActions (currentWorldState)
6  actionsForEachEntity ← explodeActionList (possibleActions)
7  parallelActions ← cartesianProduct (actionsForEachEntity)
8  randomIndexes ← intRange (0, size (parallelActions))
9  randomIndexes ← shuffle (randomIndexes)
10 foreach i in randomIndexes do
11     randomParallelActions ← parallelActions [i]
12     permutations ← permutations (varRandomParallelActions)
13     canPerformActions: boolean indicating if actions are applicable
14     canPerformActions ← True
15     foreach p in permutations do
16         foreach a in p do
17             if canPerformAction (currentWorldState, a) then
18                 currentWorldState ← applyAction (currentWorldState, a)
19             else
20                 canPerformActions ← False
21                 exit loop
22             end
23         end
24         if canPerformActions is False then
25             exit loop
26         end
27     end
28     if canPerformActions is True then
29         return randomParallelActions
30     end
31 end
```

Referring to algorithm 3, we start by computing all the actions that are applicable to the current world state (line 5). That returns us a list which doesn't distinguish between actions for each entity. Since our end goal is to assign one action to each entity that is deemed active in the current world state, we need to parse this list into a

memory structure that differentiates between them. To do that we use a function that iterates over the list, and fills a dictionary with actions for each active entity (line 6).

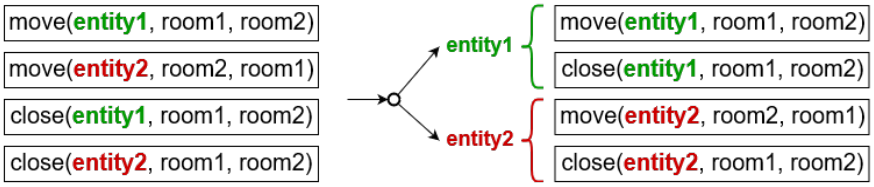


Figure 4.15: Actions Divided by Active Entity.

Then, as shown in figure 4.16, we compute the Cartesian product of the various actions (line 7), and obtain all the possible tuples, each containing one action for every active entity.

move(entity1, room1, room2)	move(entity2, room2, room1)
move(entity1, room1, room2)	close(entity2, room1, room2)
close(entity1, room1, room2)	move(entity2, room2, room1)
close(entity1, room1, room2)	close(entity2, room1, room2)

Figure 4.16: Cartesian product of the actions

At this point, we have a structure that contains sequences of actions, that may be applicable in parallel, divided by the entity that is performing them. Our goal is to select one sequence that is performable in parallel. According to the definition 4.6.1, we need to check, for each sequence, if every permutation is fully applicable to the current world state. The mere computation required to obtain all the permutations for every sequence is very costly. For this reason, we iterate over the list of sequences (lines 15-30) and each time, after computing its permutations, we check if every action of each permutation is applicable to the starting state (lines 16-26).

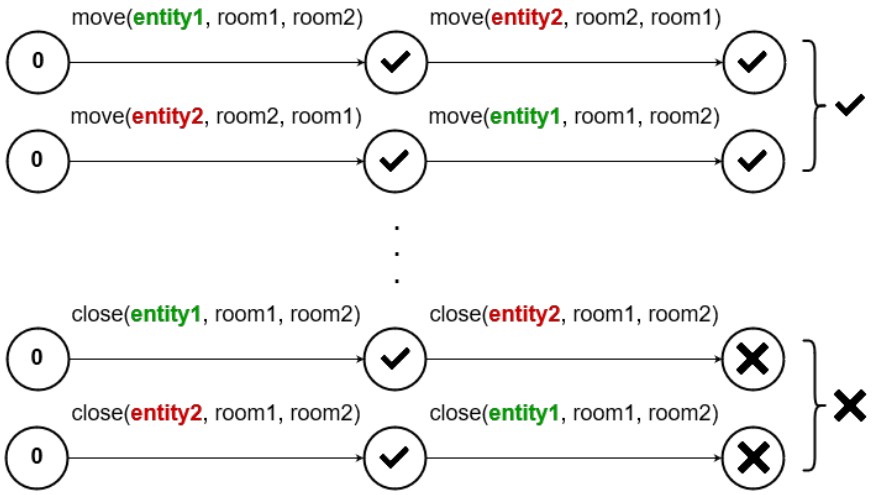


Figure 4.17: Permutations of actions tuples.

As shown in figure 4.17, the actions *move(entity1, room1, room2)* and *move(entity2, room2, room1)* are performable in parallel because every possible permutation is applicable to the current state. On the other hand, the set of actions containing the close action lead to a conflicting state.

4.6.2 Synchronization

Another issue that we have to consider when dealing with parallel actions is their effect during the refinement process. In particular, we need to worry about the synchronization of parallel actions when refining abstract plans.

Let us consider the case in which we have a **give action** that lets an entity give something to another entity. Let us define a precondition of such action so that it requires that both entities are in the same room. If we define two levels of abstraction, one of which omits this precondition's relation, we might need to account for that while refining the plans.

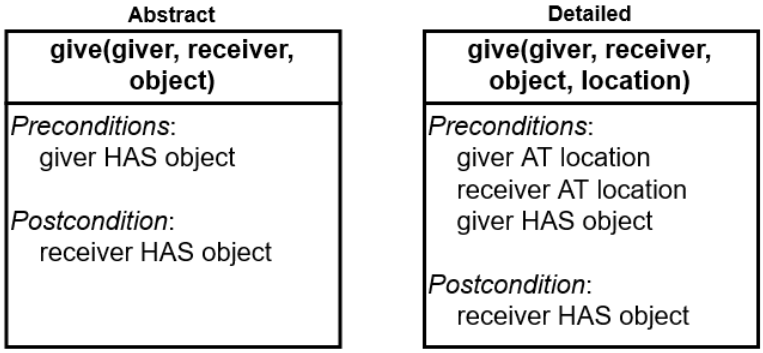


Figure 4.18: Give action at different levels of detail.

If we roll forward the simulation with the abstract level of detail in Figure 4.18, we could generate states in which the give action takes place without the entities being in the same location. When we try to refine the simulation we would need to devise a plan that respects the preconditions: in this case we would need to move each entity to a common location. In this regard, the synchronization is not trivial. In fact, it could be the case that different entities need different numbers of steps to reach the chosen location.

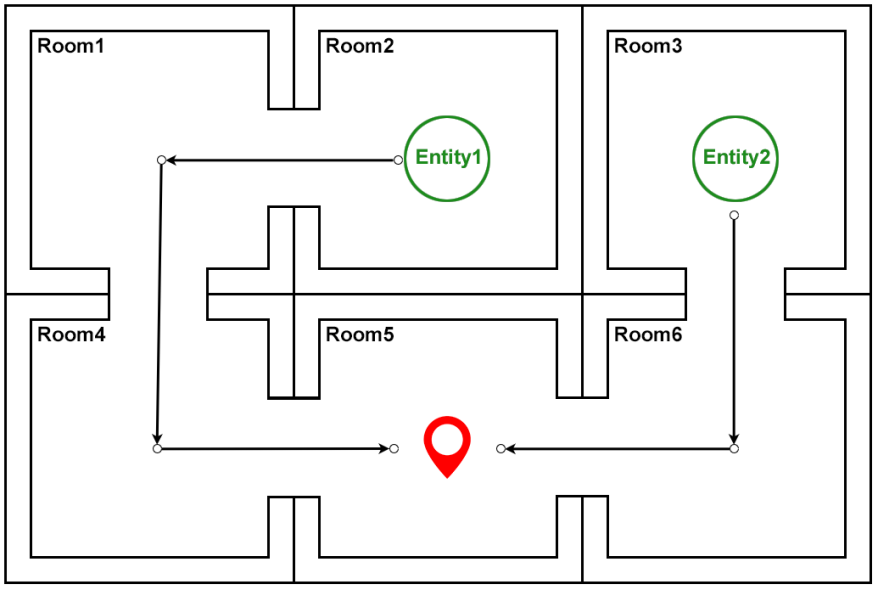


Figure 4.19: Synchronous actions with different numbers of steps.

We can see in Figure 4.19 that both entities need to reach room5 to perform the give action. Entity1 requires three steps to reach the destination, while entity2 only needs two actions. A possible solution to the problem would be for entity2, upon reaching the destination, to wait for entity1. In general terms, we need to find a point of synchronization. To achieve this, at each refinement step, we divide the detailed plan into sequences of synchronized parallel actions.

Algorithm 4: Refinement step for synchronized parallel actions.

```
   Input  : Plan composed of refined actions
   Output: Equivalent plan with sequences of synchronized parallel actions
 1 actionsForEachEntity ← Split the plan into actions for each entity
 2 activeEntities ← List of entities that can perform some action
 3 sequentialActions ← Queue of list of actions
 4 while there are actions to assign do
 5 |   parallelActions ← An empty list of actions, will contain actions for each turn
 6 |   foreach Entity e in activeEntities do
 7 |   |   entityActions ← Queue of entity's actions from the plan still unassigned
 8 |   |   if entityActions contains some action then
 9 |   |   |   parallelActions ← Pop(entityActions)
10 |   |   else
11 |   |   |   actionIdle ← make entity do nothing
12 |   |   |   parallelActions ← actionIdle
13 |   |   end
14 |   end
15 |   if we assigned any non-idle action then
16 |   |   sequentialActions ← add(parallelActions)
17 |   end
18 end
```

The plan returned from the refinement step in Figure 4.19 would look like the image on the right. The order is not important as the search algorithm is just trying to reach a goal state equivalent to the one in the abstract plan. This plan is not synchronized: entity 1 has more actions to perform compared to entity 2.

move(entity1, room2, room1)
move(entity1, room1, room4)
move(entity2, room3, room6)
move(entity2, room6, room5)
move(entity1, room4, room5)

Figure 4.20: Plan not synchronized.

Following the Algorithm 4, we start by splitting the plan into the actions for each entity (line 1). Referring to Figure 4.20, we notice that entity1 has three actions to perform before reaching the goal, while entity2 only has two.

We divide the plan into turns; at each turn we assign one action for each entity (lines 4-17), until we assign every step of the plan (line 4). If one entity reaches its own goal, but we still need to assign actions to other entities, that entity will continuously be assigned idle actions at each turn (lines 10-13), until the goal is reached by everybody.

	Entity1	Entity2
Turn 1	move(room2, room1)	move(room3, room6)
Turn 2	move(room1, room4)	move(room6, room5)
Turn 3	move(room4, room5)	IDLE

Figure 4.21: Plan divided in synchronous turn.

Since entity2 reaches the desired location before entity3, it has to wait for one turn. When entity3 arrives, the plan was successfully synchronized, so the give action can take place.

4.7 Level of Detail Switcher

In previous sections, we explained how we can adjust the level of detail of the simulation by performing abstraction and refinement. Another important function of the system is to constantly monitor the user's knowledge of the environment, and decide when it should perform the switch between the various levels of detail.

To do that, we need to devise a function which outputs an estimate of the observability of the simulation. The estimate should be based on some indicators of the player's ability to perceive the environment. I chose to employ a weighted average function because it is both versatile and easy to tune.

$$\frac{\sum \prod_{i\leftarrow 0}^{n} \phi_i \delta_i}{\sum_{i\leftarrow 0}^{n} \delta_i} \qquad \begin{array}{l} \phi_i : indicator\ measure \\ \delta_i : indicator's\ weight \end{array}$$

The function's inputs are two indicators representing the *proximity* of the player to the simulation and its actual *visibility*.

4.7.1 Proximity

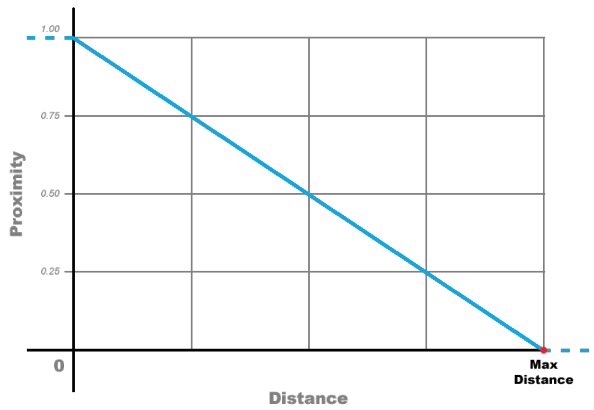


Figure 4.22: Proximity chart.

Proximity measures the inverse of the distance between the player and the simulation.

- If the distance exceeds the maximum possible value, over which the simulation is no longer visible, it returns a value of 0.
- If the distance between the player and the simulation is less or equal to 0 it returns a value of 1.
- Otherwise we return a linear value between 0 and 1.

This measure is a good factor to estimate observability, because the user's perception of the world will be affected by their distance from the simulated component.

4.7.2 Visibility

The other measure is **visibility**, which estimates whether the player's vision of the simulation is occluded by some obstacle.

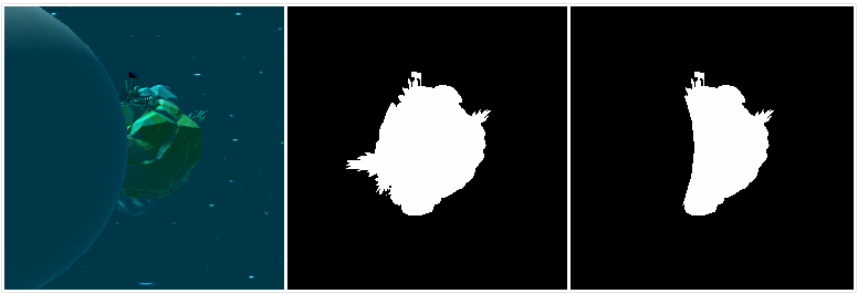


Figure 4.23: Normal image vs Filtered Image vs Occluded Image.

The Figure 4.23 shows an example of occlusion: the player is facing a planet while its moon approaches from the left, partially occluding the view. To estimate how much of the simulation is being occluded by the obstacle, we need to divide the image into layers and filter them, so that we have a clear separation between simulation and obstacles.

To produce the picture in Figure 4.23, we employed a setup with two cameras that render to different textures. In the scene we separated the relevant part of the simulation from the rest using layers.
Thanks to culling masks, we were able to make a camera render only some part of the scene: one render texture displays the occluded simulation, while on the other ignores the obstacles. Figure 4.24 shows the player vision of the location where the simulation is taking place.

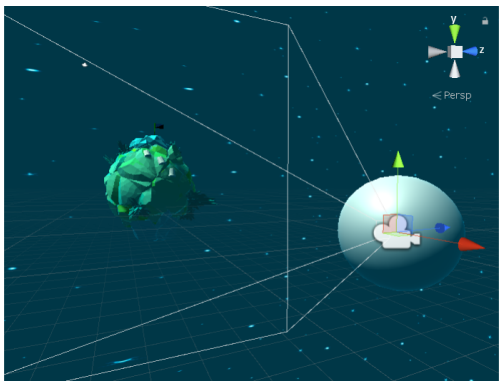


Figure 4.24: Player's field of view.

At this point, we need to be able to compute the difference between the two layers. One way to do that is to count how many pixels are different between the occluded image and the one without obstacles. To do that, we first needed to apply a particular filter to the image, so that it flattens out everything and leaves only two colors: one for the simulation and one for obstacles. The reason behind this is that we are only interested in obstacles that are occluding the player's field of view. If we counted the different pixels without filtering the image beforehand, we would account also for obstacles that are not obstructing the player's vision of the simulation.

To achieve this result, we used the post-processing stack offered by Unity. This tool lets us create post-processing profiles and apply them to a particular camera using a post-processing behaviour script. The end result is that we can apply a white profile to the simulation layer, and a black profile to everything else.

After producing the textures in Figure 4.23, we count the white pixels in each of them and compute the difference between the two, so that we get a measure of the simulation's occlusion.

4.7.3 Parallel Computation of Visibility

All the textures are 256x256 resolution, so each one of them contains 65536 pixels. Since we might need to update the metrics very frequently, we needed to devise a way to count the pixels more efficiently. We did so by employing compute shaders.

Compute shaders are programs that run on the graphics card, outside of the normal rendering pipeline. They can be used for massively parallel GPGPU (General-purpose computing on graphics processing units) algorithms, or to accelerate parts of game rendering [14].

Counting the white pixels of a texture is a task that could greatly benefit from parallelization, because even if usually there is a correlation between adjacent pixels in an image, their values are not causally related. For this reason we can treat them as disjoint units, and split the problem into multiple parallel tasks.

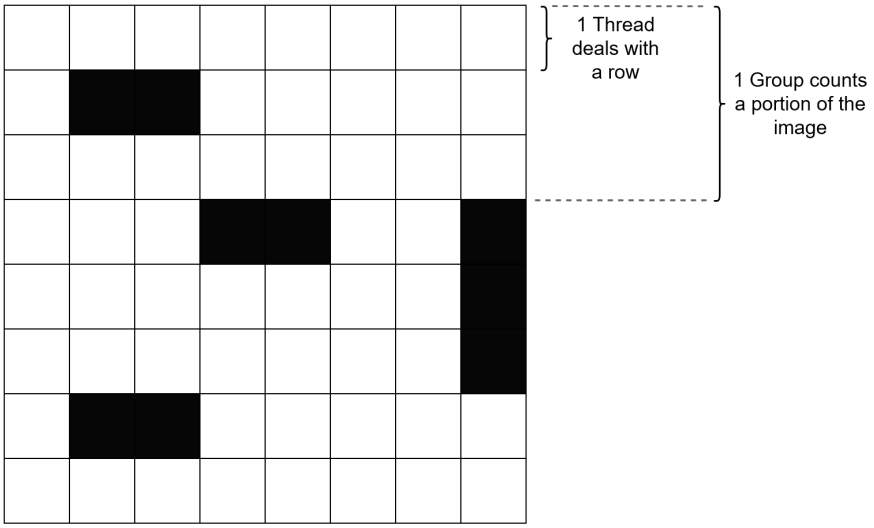


Figure 4.25: White Pixels Parallel Computation.

We dispatch a number of thread groups proportional to the height of the image and the amount of threads in each group.

$$G = \left\lfloor \frac{H + (T - 1)}{T} \right\rfloor$$

G: number of groups
H: height of the image
T: number of threads in each group

For example, if we are dealing with an image which is 300 pixels tall, and we want each group to contain 64 threads, we will dispatch 5 groups. The offset $T - 1$ is needed to have enough threads to cover the whole image. If we computed the necessary groups without the offset, we would have dispatched 4 groups, which are not enough since $64 \times 4 = 256 < 300$.

After dispatching the threads, each one of them iterates over its row of the image, and checks the color of every pixel (lines 4-10). If it is white we increase a counter in a memory location which is shared with the group. Each thread checks if it was assigned a row within the image boundaries (line 5). For example, if we have an image which is 300 pixels tall, processed with 5 groups of 64 threads each, in the fifth group, only the first 44 threads would count the pixels, as the others would count something which is not within the image.

There is a synchronization point after the iteration over the row (line 12), so that each thread has to wait for the others to complete their tasks before proceeding. Once every thread has counted its row, only the first thread of the group sums all the partial counters, computing the sum of white pixels in the portion of the image assigned to the group (lines 13-19).

Once every group has completed dealing with its portion of the image, we return an array of integers that contains the white pixels of each group. Then we can easily sum all the values in the array to obtain the total count of white pixels in the image.

Algorithm 5: Count white pixels in parallel with compute shader.

```
   Input  : The texture with the pixels to count
   Output: The number of white pixels in the texture
 1 groupID: ID of thread in a thread group
 2 groupIndex: index within the group
 3 rowSumData: white pixels for each row

 4 rowSumData [groupIndex ] ← 0
 5 if thread is inside texture then
 6 |   foreach Pixel p in the row do
 7 |   |   if p is white then
 8 |   |   |   rowSumData [groupIndex ]++
 9 |   |   end
10 |   end
11 end
12 SynchronizeThreadsInGroup ( )
13 if groupIndex = 0 then
14 |   sum: sum of the white pixels in the group
15 |   foreach row r in rows of the group do
16 |   |   sum += rowSumData [r]
17 |   end
18 |   GroupSumBuffer [groupID ] = sum
19 end
```

4.8 Summary

In this chapter we discussed:

- **PDDL framework**: we designed and implemented this component to model general purpose simulations. Since it is based on the Planning Domain Definition Language, we can easily apply search algorithms to traverse the simulation tree.
- **Decrease the Level of Detail**: we devised a way of simplifying any given state of the simulation. We apply the abstract domain to the state of the world at current level of detail. All the features that disappear from the world are the details. Running the simulation at abstract LoD should be less computationally intensive.
- **Increase the Level of Detail**: we devised a way of computing the additional features required for running the simulation at a higher level of detail. We refine

every abstract action that already happened. This leads us to a state which is consistent with the additional constraints of the higher level of detail. From there we apply the transition model contained in the new domain to run the simulation.

- **Level of detail switcher**: we employed a function that estimates observability to switch the simulation level of detail. This function is based on the distance between the player and the simulation; and the occlusion of the player's field of view.

5. Evaluation

To test our approach, we devised a scenario that models the behaviour of multiple intelligent agents in a wide environment. This setting is ideal, because the agents that are part of the model do not have a precisely defined goal, and so we can easily simulate their behaviour, applying the transition model. It also contains some constraints that challenge the refinement process, and may force the system to deal with concurrency problems.

We will now introduce the modeled environment and go trough a step by step computation of the adjustment of level of detail. We will then present the obtained results.

5.1 Example Environment

The scenario contains two rovers that can perform multiple actions: moving around the environment, taking samples of the soil, dropping the sample on a specific dock and snapping pictures of objectives. We modeled the resulting environment with our PDDL framework so that we could simulate it using our approach for self adjusting level of detail. The first thing we had to do was split the general problem into multiple levels of abstraction, to have different degrees of complexity to work with. Each level of detail has its own domain which can vary from the others in the conditions that are required for action execution, and can even contain new actions and mechanics.

Figure 5.1: An overview of the relation between the player and the simulation.

Everything that we're simulating happens on the planet; the player is thought of as an external entity far away from the place where the simulation is taking place. The player can move toward the location of the simulation or in the opposite direction. The lines are visual representations of the estimation function: when the observability goes higher than a certain threshold, we switch the simulation to the respective level of detail. As shown in figure 5.1, the environment is divided in areas denoted by the thresholds. *(i)* When the observability value is between 0 and 0.4, the simulation is

run at level of detail 1. *(ii)* Between 0.4 and 0.7, the simulation is run at level of detail 2. *(iii)* Between 0.7 and 1 the simulation is run at level of detail 3.

5.1.1 First Level of Detail

In the first level of detail, the rovers are free to move among the waypoints as long as they're connected to one another. If a rover happens to be in a waypoint that contains a sample, it can collect it. Analogously if an objective is visible from a waypoint, the rover can take a picture of it. One of the waypoints is marked as a dropping dock, where the rover can drop the collected sample. There are two rovers in the environment, both starting in the same location.

Entity Type	Description
ROVER	The protagonist of the world; an intelligent agent that performs actions.
WAYPOINT	A location in the world; it can be connected to other waypoints so the rover can move among them.
SAMPLE	A soil sample of the planet that the rover can pick up and store in a dropping dock.
OBJECTIVE	An interesting feature of the planet; the rover can capture it by snapping a picture.

Table 5.1: Entity Types in the first level of detail of rover's environment.

Type	Entities
ROVER	ROVER1, ROVER2
WAYPOINT	WAYPOINT1,WAYPOINT2,WAYPOINT3, WAYPOINT4,WAYPOINT5,WAYPOINT6, WAYPOINT7,WAYPOINT8,WAYPOINT9
SAMPLE	SAMPLE1,SAMPLE2,SAMPLE3, SAMPLE4,SAMPLE5,SAMPLE6
OBJECTIVE	OBJECTIVE1,OBJECTIVE2,OBJECTIVE3, OBJECTIVE4, OBJECTIVE5,OBJECTIVE6, OBJECTIVE7,OBJECTIVE8,

Predicate	Type	Description
IS_CONNECTED_TO	binary	describes if a two waypoints are connected
IS_VISIBLE	binary	describes wheter an objective is visible from a waypoint
IS_IN	binary	describes if a sample is in a waypoint
BEEN_AT	binary	describes wheter the rover has been in a waypoint
CARRY	binary	describes if the rover is carrying a sample
AT	binary	indicates the current location of the rover
IS_DROPPING_DOCK	unary	indicates if the waypoint is a dropping dock for samples
TAKEN_IMAGE	unary	indicates if the rover has taken an image of the objective
STORED_SAMPLE	unary	indicates if the rover has stored a particular sample
IS_EMPTY	unary	indicates if the rover is empty

Table 5.2: Predicates in the first level of detail of rover's environment.

Name	Parameters	Preconditions	Postconditions	Description
MOVE	• rover • start • destination	• rover AT start • start IS_CONNECTED_TO destination	• **not** rover AT start • rover AT destination	moves the rover from a start to a destination
TAKE SAMPLE	• rover • sample • waypoint	• sample IS_IN waypoint • rover IS_AT waypoint • rover IS_EMPTY	• **not** sample IS_IN waypoint • **not** rover IS_EMPTY • rover CARRY sample	take sample from location
DROP SAMPLE	• rover • sample • waypoint	• waypoint IS_DROPPING_DOCK • rover IS_AT waypoint • rover CARRY sample	• sample IS_IN waypoint • rover IS_EMPTY • **not** rover CARRY sample	drop sample to location
TAKE IMAGE	• rover • objective • waypoint	• IS_VISIBLE waypoint • rover IS_AT waypoint	• objective TAKEN_IMAGE	take image of objective

Table 5.3: Actions in the first level of detail of rover's environment.

Source	Predicate	Destination
WAYPOINT1	IS_CONNECTED_TO	WAYPOINT5
WAYPOINT2	IS_CONNECTED_TO	WAYPOINT5
WAYPOINT3	IS_CONNECTED_TO	WAYPOINT6
WAYPOINT4	IS_CONNECTED_TO	WAYPOINT8
WAYPOINT5	IS_CONNECTED_TO	WAYPOINT1
WAYPOINT6	IS_CONNECTED_TO	WAYPOINT3
WAYPOINT6	IS_CONNECTED_TO	WAYPOINT8
WAYPOINT8	IS_CONNECTED_TO	WAYPOINT4
WAYPOINT9	IS_CONNECTED_TO	WAYPOINT1
WAYPOINT1	IS_CONNECTED_TO	WAYPOINT9
WAYPOINT3	IS_CONNECTED_TO	WAYPOINT4
WAYPOINT4	IS_CONNECTED_TO	WAYPOINT3
WAYPOINT4	IS_CONNECTED_TO	WAYPOINT9
WAYPOINT5	IS_CONNECTED_TO	WAYPOINT2
WAYPOINT6	IS_CONNECTED_TO	WAYPOINT7
WAYPOINT7	IS_CONNECTED_TO	WAYPOINT6
WAYPOINT8	IS_CONNECTED_TO	WAYPOINT6
WAYPOINT9	IS_CONNECTED_TO	WAYPOINT4
OBJECTIVE1	IS_VISIBLE	WAYPOINT2
OBJECTIVE1	IS_VISIBLE	WAYPOINT4
OBJECTIVE2	IS_VISIBLE	WAYPOINT7
OBJECTIVE4	IS_VISIBLE	WAYPOINT5
OBJECTIVE1	IS_VISIBLE	WAYPOINT3
OBJECTIVE2	IS_VISIBLE	WAYPOINT5
OBJECTIVE3	IS_VISIBLE	WAYPOINT8
OBJECTIVE4	IS_VISIBLE	WAYPOINT1
SAMPLE1	IS_IN	WAYPOINT2
SAMPLE3	IS_IN	WAYPOINT9
SAMPLE5	IS_IN	WAYPOINT3
SAMPLE2	IS_IN	WAYPOINT3
SAMPLE4	IS_IN	WAYPOINT8

SAMPLE6	IS_IN	WAYPOINT3
WAYPOINT7	IS_DROPPING_DOCK	#
ROVER1	IS_EMPTY	#
ROVER2	IS_EMPTY	#
ROVER1	AT	WAYPOINT6
ROVER2	AT	WAYPOINT6

Table 5.4: Relations in the initial state for the first level of detail.

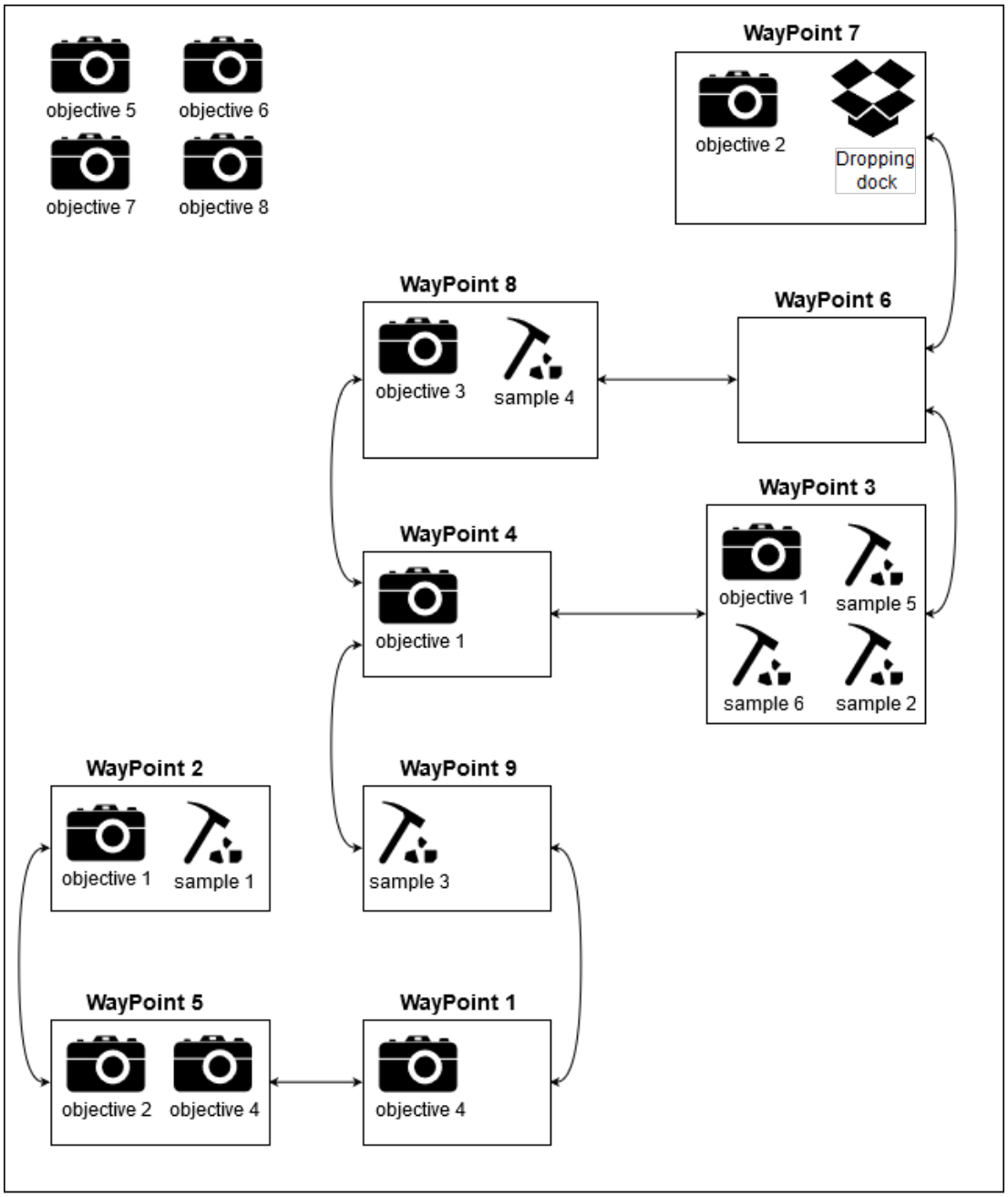


Figure 5.2: Rover Environment Level of Detail 1.

5.1.2 Second Level of Detail

The second Level of Detail in the example simulation differs only slightly from the first one. We added one new predicate, called **OBSTACLE_BETWEEN**, that indicates the presence of obstacles between waypoints. The resulting relation has a directional meaning: if there is an obstacle between waypoint1 and waypoint2, the rover cannot go from waypoint1 to waypoint2, but it can still go from waypoint2 to waypoint1 (e.g. rolling down a hill).

Source	**Predicate**	**Destination**	**Value**
WAYPOINT4	OBSTACLE_BETWEEN	WAYPOINT3	TRUE
WAYPOINT8	OBSTACLE_BETWEEN	WAYPOINT4	TRUE
WAYPOINT6	OBSTACLE_BETWEEN	WAYPOINT8	TRUE
WAYPOINT3	OBSTACLE_BETWEEN	WAYPOINT6	TRUE
WAYPOINT1	OBSTACLE_BETWEEN	WAYPOINT5	FALSE
WAYPOINT2	OBSTACLE_BETWEEN	WAYPOINT5	FALSE
WAYPOINT4	OBSTACLE_BETWEEN	WAYPOINT8	FALSE
WAYPOINT5	OBSTACLE_BETWEEN	WAYPOINT1	FALSE
WAYPOINT6	OBSTACLE_BETWEEN	WAYPOINT3	FALSE
WAYPOINT9	OBSTACLE_BETWEEN	WAYPOINT1	FALSE
WAYPOINT1	OBSTACLE_BETWEEN	WAYPOINT9	FALSE
WAYPOINT3	OBSTACLE_BETWEEN	WAYPOINT4	FALSE
WAYPOINT4	OBSTACLE_BETWEEN	WAYPOINT9	FALSE
WAYPOINT5	OBSTACLE_BETWEEN	WAYPOINT2	FALSE
WAYPOINT6	OBSTACLE_BETWEEN	WAYPOINT7	FALSE
WAYPOINT7	OBSTACLE_BETWEEN	WAYPOINT6	FALSE
WAYPOINT8	OBSTACLE_BETWEEN	WAYPOINT6	FALSE

Table 5.5: Additional Relations in the initial state for the second level of detail.

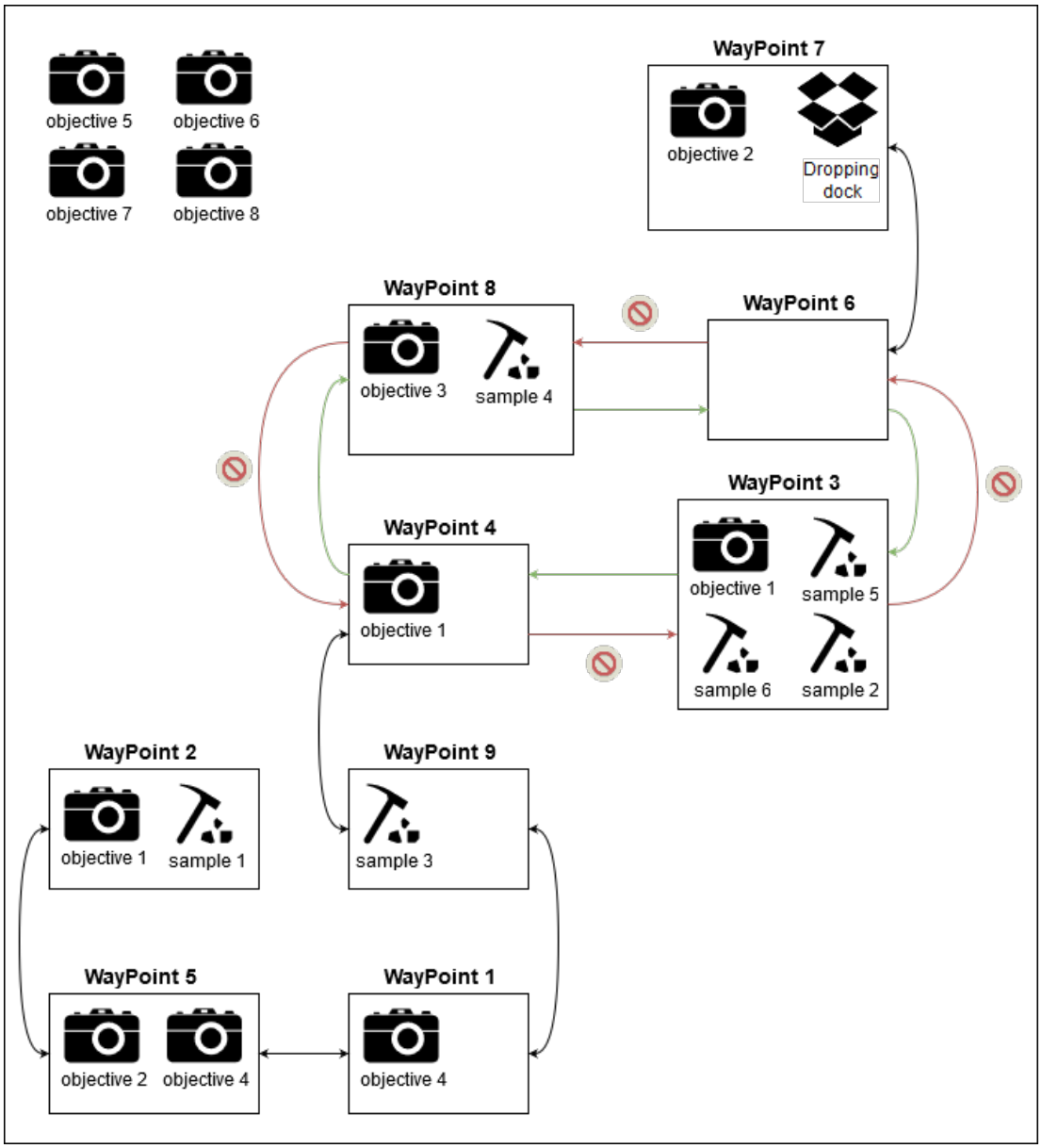


Figure 5.3: Rover Environment Level of Detail 2.

As we can see from Figure 5.3, in the central part, the rover can move only in a particular direction. For example, it cannot move from waypoint3 to waypoint6, instead it has to go the other way around: waypoint3 $\longrightarrow$ waypoint4 $\longrightarrow$ waypoint8 $\longrightarrow$ waypoint6.

5.1.3 Third Level of Detail

In the third level of detail, we added a battery to the rover: we implemented a transition model that regulates the charge level of the battery, tied each rover with its own battery, and modified the actions, so that they make the battery transition from one charge level to a successor. We also implemented a new action that lets the rover charge the battery.

Entity Type	**Description**
BATTERY	Type of a battery that keeps the carge level of the rover
BATTERY_LEVEL	A charge level, can either perform actions or not

Table 5.6: Entity Types in the third level of detail of rover's environment.

Predicate	**Type**	**Description**
HAS	binary	ties each rover to its own battery
HAS_BATTERY_LEVEL	binary	ties each battery to its own battery level
CAN_PERFORM_ACTION	unary	describes if a battery level can perform an action
DISCHARGES_TO	binary	ties each battery level to its predecessor
CHARGES_TO	binary	ties each battery level to its successor

Table 5.7: Predicates in the third level of detail of rover's environment.

We modified all the actions so that they interact with the new system, and discharge the battery of the rover. Here's an example for the move action:

	MOVE
Parameters	rover, from, to, battery, batteryLevelFrom, batteryLevelTo
Preconditions	*move preconditions from LoD2...* • rover HAS battery, • battery HAS_BATTERY_LEVEL batteryLevelFrom, • batteryLevelFrom CAN_PERFORM_ACTION • batteryLevelFrom DISCHARGES_TO batteryLevelTo
Postconditions	*move postconditions from LoD2...* • **not** battery HAS_BATTERY_LEVEL batteryLevelFrom, • battery HAS_BATTERY_LEVEL batteryLevelTo

Table 5.8: The action move discharges the battery of the rover.

Referring to Figure 5.8, we can see that the action now takes into consideration the battery system, and is able to interact with it. For the action to be performable, its parameter must contain: (*i*) the rover; (*ii*) the correct rover's battery; and (*iii*) the battery's charge level. Before execution, we check if the battery level is enough to perform an action. The last precondition is employed so that we only get the correct charge levels. For example MOVE(..., level3, level2) is correct, while MOVE(..., level2, level3) is not correct, because it would mean that we were charging the battery with a move action, instead of discharging it.

	CHARGE
Parameters	rover, battery, batteryLevelFrom, batteryLevelTo
Preconditions	• rover HAS battery, • battery HAS_BATTERY_LEVEL batteryLevelFrom, • batteryLevelFrom CHARGES_TO batteryLevelTo
Postconditions	• **not** battery HAS_BATTERY_LEVEL batteryLevelFrom, • battery HAS_BATTERY_LEVEL batteryLevelTo

Table 5.9: Action charge in the third level of detail of the rover's environment.

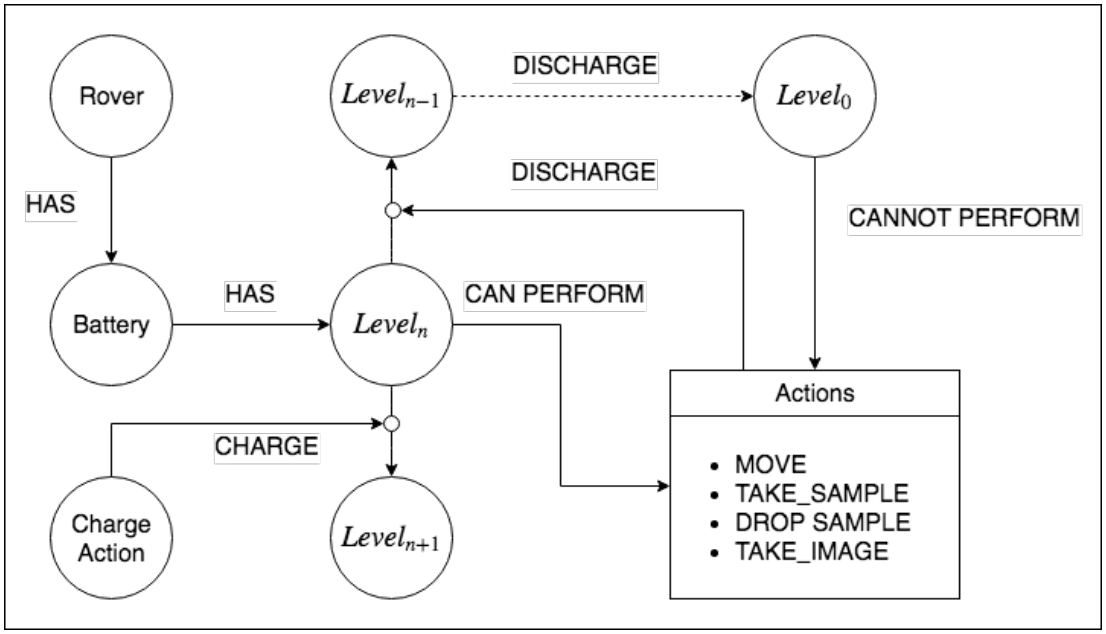


Figure 5.4: Rover Environment Level of Detail 3.

In Figure 5.4, we can see how we implemented the battery system in the example simulation. Each rover is tied to its own battery with a **HAS** relation in the world state, and similarly, each battery has its own charge level. The rover can increase its battery level by performing a charge action, which transitions the battery charge to the adjacent upper level (e.g. $2 \longrightarrow 3$). Each action performed by the rover, excluding charge, decreases the battery level by one, until it reaches a level that is not allowed to perform actions.

5.1.4 Refinement

Let's assume that the player starts in a setting such that the observability estimate makes the system run the simulation at level of detail 1 (e.g. the environment of Figure 5.1). As shown in Figure 5.5, the rovers perform three sets of parallel actions. After all of them are completed, we put the reached state into the slot of LoD 1.

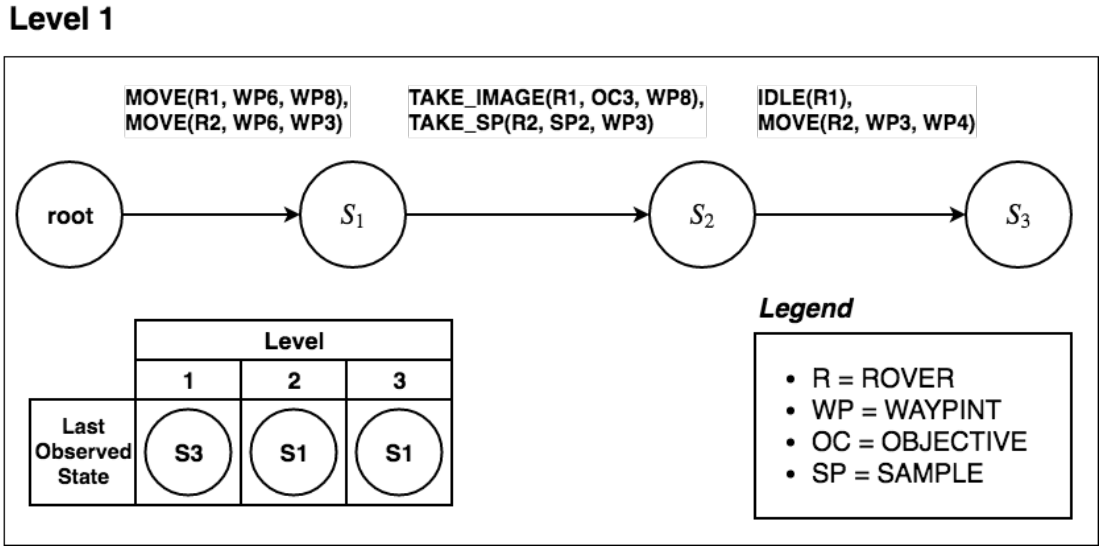
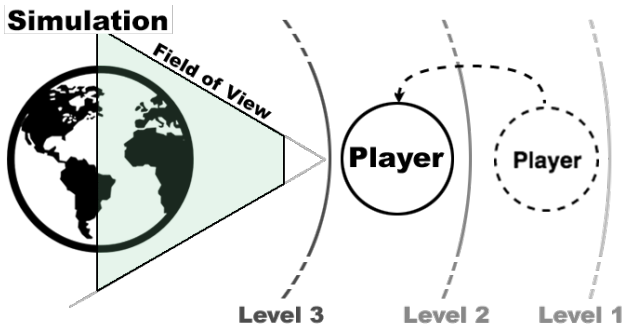


Figure 5.5: three sets of parallel actions performed at LoD 1.



Now let's assume that the player moves towards the simulation. In this case, he would be able to perceive more details of the environment. Eventually, the estimation function will exceed the threshold and trigger refinement.

Once the refinement process is triggered, the system will roll back the simulation until it hits the root node. From there, it will employ a search algorithm to compute, for every set of parallel actions, an equivalent sequence that also accounts for the details in the new level.

As we mentioned in Figure 5.3, the second level of detail accounts for obstacles in the paths that the rover can take to move between locations. That means that, if some actions performed in the previous level of detail, are not performable because of some detail previously ignored, now we have to account for them.

In particular, referring to Figure 5.3, we can see that now the rover cannot go straight from WAYPOINT6 to WAYPOINT8 because there is an obstacle in between. For this reason the search algorithm will return the following detour:

$$waypoint6 \longrightarrow waypoint3 \longrightarrow waypoint4 \longrightarrow waypoint8.$$

In this regard, we can also observe the way that synchronization comes into play. Since

the single action for *rover one* has been refined into three actions; while *rover two*'s action was directly applicable, and didn't need refinement, rover two will wait for two turns before performing its action.

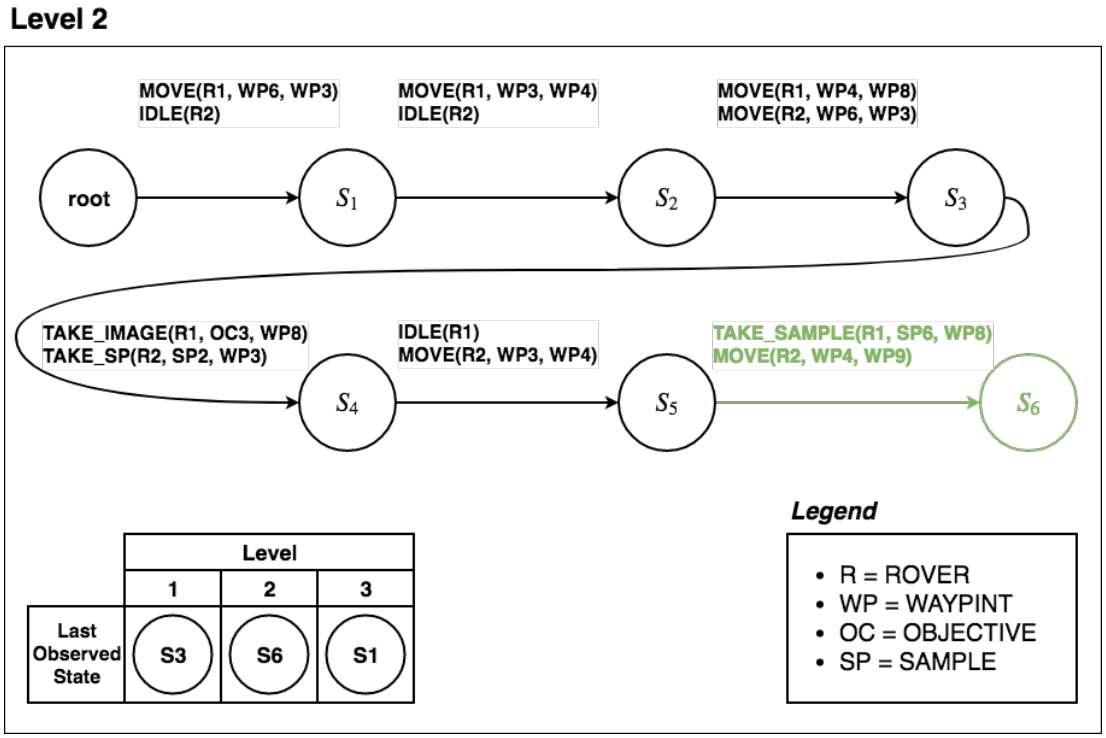
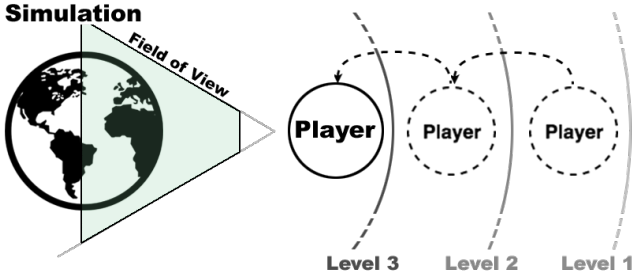


Figure 5.7: Three sets of parallel actions performed at LoD 2.

After the refinement process is completed, the system can choose another action from the transition model and roll forward the simulation. This time though, the simulation will be run according to the rules of the new level of detail. We can see that *rover one* takes a sample, and *rover two* moves to a new location (shown in green in Figure 5.7). Both actions are compliant with the constraints of the second level of detail.



Let's assume that the player moves again towards the simulation. This enables them to perceive more details of the environment. Eventually, the estimation function will exceed the threshold and trigger refinement.

As shown in Figure 5.4, the new level of detail models the battery system. At this level of detail, the actions discharge the battery level of the rover that is executing them. This entails that the rovers can only perform actions as long as their battery has charge left.

In the actual world state we have 4 battery levels, going from 3 (fully charged) to 0 (empty). If we recall the first refinement step of Figure 5.7, we parsed the move actions into a sequence of 3 actions, and then, from S_3 to S_4, we performed another

set of actions. Now, since each action discharges the battery by 1 charge level, rover 1 reaches S_3 with an empty battery. This means that, before performing any other action, it must take a turn to charge its battery. In fact, we can observe that $S_3 \longrightarrow S_4$ is a charge action for rover 1, along with an idle for rover 2.

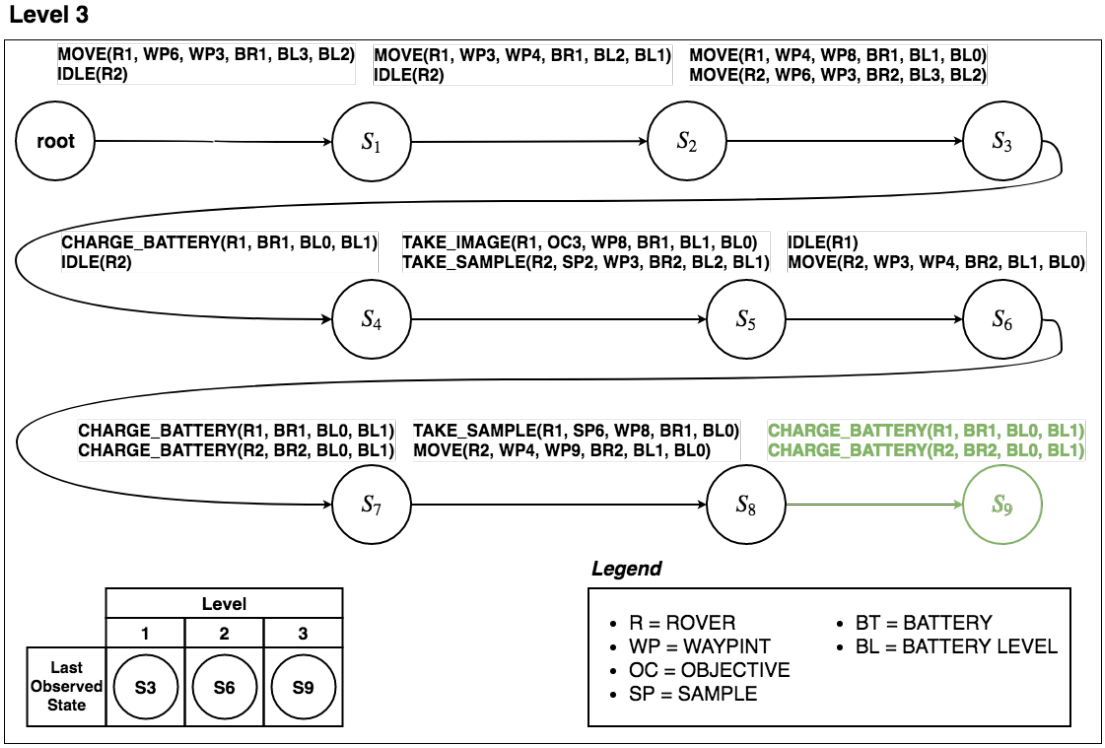


Figure 5.9: three sets of parallel actions performed at LoD 3.

5.1.5 Abstraction

Now let us imagine that an obstacle appears in front of the player while the simulation is running at level of detail 3. This occludes the field of view of the player, causing the observability function to drop below the threshold of level 2.

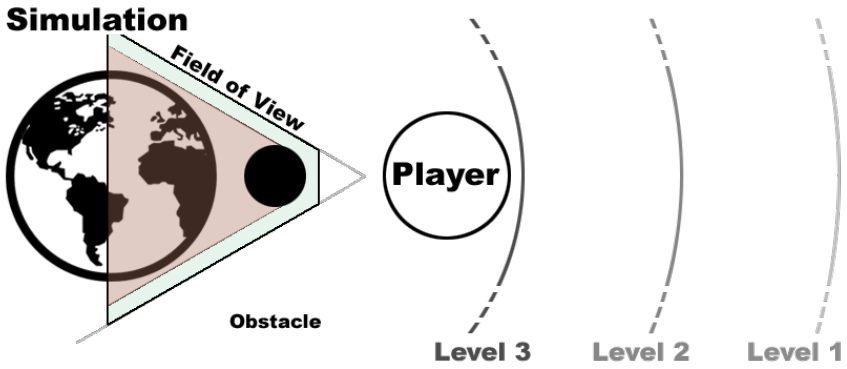


Figure 5.10: Obstacle occluding the field of view of the player.

This means that we can perform abstraction, and start running the simulation at level of detail 2, saving resources without the player noticing the drop in detail level.

If we follow all the actions described in Figure 5.9, we end up with the following world state at the third level of detail.

DOMAIN	WORLD STATE	
Entity Types	**Entities**	**Relations**
ROVER WAYPOINT SAMPLE OBJECTIVE **BATTERY** **BATTERY_LEVEL**	ROVER1 ROVER2 WAYPOINT1 ... WAYPOINT9 SAMPLE1 ... SAMPLE6 OBJECTIVE1 ... OBJECTIVE8 **BATTERY_ROVER1** **BATTERY_ROVER2** **BATTERY_LEVEL0** **BATTERY_LEVEL1** **BATTERY_LEVEL2** **BATTERY_LEVEL3**	WAYPOINT_A IS_CONNECTED_TO WAYPOINT_B ... WAYPOINT_Y IS_CONNECTED_TO WAYPOINT_Z WAYPOINT_4 OBSTACLE_BETWEEN WAYPOINT_3 WAYPOINT_8 OBSTACLE_BETWEEN WAYPOINT_4 WAYPOINT_6 OBSTACLE_BETWEEN WAYPOINT_8 WAYPOINT_3 OBSTACLE_BETWEEN WAYPOINT_6 OBJECTIVE_A IS_VISIBLE WAYPOINT_B ... OBJECTIVE_Y IS_VISIBLE WAYPOINT_Z SAMPLE_A IS_IN WAYPOINT_B ... SAMPLE_Y IS_VISIBLE WAYPOINT_Z WAYPOINT7 IS_DROPPING_DOCK ROVER1 IS_EMPTY ROVER2 IS_EMPTY ROVER1 AT WAYPOINT6 ROVER2 AT WAYPOINT6 **OBJECTIVE3 TAKEN_IMAGE** **SAMPLE2 STORED_SAMPLE** **SAMPLE6 STORED_SAMPLE** **NOT SAMPLE2 IS_IN WP3** **NOT SAMPLE6 IS_IN WP8** **ROVER1 HAS BATTERY_ROVER1** **ROVER2 HAS BATTERY_ROVER2** **BATTERY_ROVER1 HAS BATTERY_LEVEL3** **BATTERY_ROVER2 HAS BATTERY_LEVEL3** **BATTERY_LEVEL0 CHARGES_TO BATTERY_LEVEL1** **BATTERY_LEVEL1 CHARGES_TO BATTERY_LEVEL2** **BATTERY_LEVEL2 CHARGES_TO BATTERY_LEVEL3** **BATTERY_LEVEL3 DISCHARGES_TO BATTERY_LEVEL2** **BATTERY_LEVEL2 DISCHARGES_TO BATTERY_LEVEL1** **BATTERY_LEVEL1 DISCHARGES_TO BATTERY_LEVEL0**
Predicates		
WAYPOINT IS_CONNECTED_TO WAYPOINT OBJECTIVE IS_VISIBLE WAYPOINT SAMPLE IS_IN WAYPOINT ROVER BEEN_AT WAYPOINT ROVER CARRY SAMPLE ROVER AT WAYPOINT IS_DROPPING_DOCK OBJECTIVE TAKEN_IMAGE SAMPLE STORED_SAMPLE ROVER IS_EMPTY WAYPOINT OBSTACLE_BETWEEN WAYPOINT **ROVER HAS BATTERY** **BATTER HAS_BATTERY_LEVEL BATTERY_LEVEL** **BATTERY_LEVEL CAN_PERFORM_ACTION** **BATTERY_LEVEL DISCHARGES_TO BATTERY_LEVEL** **BATTERY_LEVEL CHARGES_TO BATTERY_LEVEL**		
Actions		
IDLE(R) MOVE(R, START, DEST, BR, BLX, BLY) TAKE_SAMPLE(R, SP, WP, BR, BLX, BLY) DROP_SAMPLE(R, SP, WP, BR, BLX, BLY) TAKE_IMAGE(R, OC, WP, BR, BLX, BLY) **CHARGE(R, BR, BLX, BLY)**		

Figure 5.11: Final state of the example simulation at level of detail 3.

The relations in bold are the ones added to the world state during the course of the simulation, as effects of the applied actions. For example, the action "take image" is responsible for the relation "objective taken".

All the components in red are the ones specific to this level of detail. We can see that they describe everything related to the simulation of the battery system. As soon as we perform abstraction, the domain of the problem is reverted back to level 2. This means that we will lose every component related to the battery system. We should pay particular attention to the actions. From LoD3 to LoD2 many of them get modified, losing the parameters related to the battery system.

Since the domain is modified, the world state is affected as well. Everything related to the missing components of the domain can no longer be part of the world state. We see that we lose the "battery" and "battery level" entities, and all the related relations.

DOMAIN	WORLD STATE	
Entity Types	**Entities**	**Relations**
ROVER WAYPOINT SAMPLE OBJECTIVE	ROVER1 ROVER2 WAYPOINT1 ... WAYPOINT9 SAMPLE1 ... SAMPLE6 OBJECTIVE1 ... OBJECTIVE8	WAYPOINT_A IS_CONNECTED_TO WAYPOINT_B ... WAYPOINT_Y IS_CONNECTED_TO WAYPOINT_Z **WAYPOINT_4 OBSTACLE_BETWEEN WAYPOINT_3** **WAYPOINT_8 OBSTACLE_BETWEEN WAYPOINT_4** **WAYPOINT_6 OBSTACLE_BETWEEN WAYPOINT_8** **WAYPOINT_3 OBSTACLE_BETWEEN WAYPOINT_6** OBJECTIVE_A IS_VISIBLE WAYPOINT_B ... OBJECTIVE_Y IS_VISIBLE WAYPOINT_Z SAMPLE_A IS_IN WAYPOINT_B ... SAMPLE_Y IS_VISIBLE WAYPOINT_Z WAYPOINT7 IS_DROPPING_DOCK ROVER1 IS_EMPTY ROVER2 IS_EMPTY ROVER1 AT WAYPOINT6 ROVER2 AT WAYPOINT6 **OBJECTIVE3 TAKEN_IMAGE** **SAMPLE2 STORED_SAMPLE** **SAMPLE6 STORED_SAMPLE** **NOT SAMPLE2 IS_IN WP3** **NOT SAMPLE6 IS_IN WP8**
Predicates		
WAYPOINT IS_CONNECTED_TO WAYPOINT OBJECTIVE IS_VISIBLE WAYPOINT SAMPLE IS_IN WAYPOINT ROVER BEEN_AT WAYPOINT ROVER CARRY SAMPLE ROVER AT WAYPOINT IS_DROPPING_DOCK OBJECTIVE TAKEN_IMAGE SAMPLE STORED_SAMPLE ROVER IS_EMPTY **WAYPOINT OBSTACLE_BETWEEN WAYPOINT**		
Actions		
IDLE(R) MOVE(R, START, DEST) TAKE_SAMPLE(R, SP, WP) DROP_SAMPLE(R, SP, WP,) TAKE_IMAGE(R, OC, WP)		

Figure 5.12: Final state of the example simulation at level of detail 2.

In Figure 5.12, the components in red are the ones specific to level of detail 2, namely the ones used to describe the presence of obstacles between waypoints. If we were to perform another abstraction step, we would end up with this basic world state:

DOMAIN	WORLD STATE	
Entity Types	**Entities**	**Relations**
ROVER WAYPOINT SAMPLE OBJECTIVE	ROVER1 ROVER2 WAYPOINT1 ... WAYPOINT9 SAMPLE1 ... SAMPLE6 OBJECTIVE1 ... OBJECTIVE8	WAYPOINT_A IS_CONNECTED_TO WAYPOINT_B ... WAYPOINT_Y IS_CONNECTED_TO WAYPOINT_Z OBJECTIVE_A IS_VISIBLE WAYPOINT_B ... OBJECTIVE_Y IS_VISIBLE WAYPOINT_Z SAMPLE_A IS_IN WAYPOINT_B ... SAMPLE_Y IS_VISIBLE WAYPOINT_Z WAYPOINT7 IS_DROPPING_DOCK ROVER1 IS_EMPTY ROVER2 IS_EMPTY ROVER1 AT WAYPOINT6 ROVER2 AT WAYPOINT6 **OBJECTIVE3 TAKEN_IMAGE** **SAMPLE2 STORED_SAMPLE** **SAMPLE6 STORED_SAMPLE** **NOT SAMPLE2 IS_IN WP3** **NOT SAMPLE6 IS_IN WP8**
Predicates		
WAYPOINT IS_CONNECTED_TO WAYPOINT OBJECTIVE IS_VISIBLE WAYPOINT SAMPLE IS_IN WAYPOINT ROVER BEEN_AT WAYPOINT ROVER CARRY SAMPLE ROVER AT WAYPOINT IS_DROPPING_DOCK OBJECTIVE TAKEN_IMAGE SAMPLE STORED_SAMPLE ROVER IS_EMPTY		
Actions		
IDLE(R) MOVE(R, START, DEST) TAKE_SAMPLE(R, SP, WP) DROP_SAMPLE(R, SP, WP,) TAKE_IMAGE(R, OC, WP)		

Figure 5.13: Final state of the example simulation at level of detail 1.

5.2 Results

The goal of the process explained in previous chapters is for the simulation to be less computationally intensive and require less resources.

To test the gains, we first computed the theoretical size of the problem we're dealing with. After running the simulation on the example environment, we collected some data to show the impact of adaptive level of detail on system resources, and compared the practical results with the theoretical ones.

To describe a state of the simulation we use relations. A relation can either be unary or binary, depending on how many entities are involved. A relation has a truth value that can be either true or false. Thus, we can compute the number of different relations obtainable from a predicate by multiplying the number of entities in the world state that have the same type as the left side of the predicate by the number of entities in the world state that have the same type has the right side of the predicate.

LoD	Entity Types	Entities
1	ROVER	2
	WAYPOINT	9
	SAMPLE	6
	OBJECTIVE	8
3	BATTERY	2
	BATTERY_LEVEL	4

Table 5.10: Entities involved in each level of detail.

LoD	Predicates	Relations	Combinations
1	WP IS_CONNECTED_O WP	81	2,41785E+24
	OC IS_VISIBLE WP	72	4,72237E+21
	SP IS_IN WP	54	1,80144E+16
	ROV BEEN_AT WP	18	262144
	ROV CARRY SP	12	4096
	ROV AT WP	18	262144
	WP IS_DROPPING_DOCK	9	512
	OC TAKEN_IMAGE	8	256
	SP STORED_SAMPLE	6	64
	ROV IS_EMPTY	2	4
	Size of the state space		**1,94267E+84**
2	WP OBSTACLE_BETWEEN WP	81	2,41785E+24
	Size of the state space		**4,69709E+108**
3	ROV HAS BT	4	16
	BT HAS_BATTERY_LEVEL BL	8	256
	BL CAN_PERFORM_ACTION	4	16
	BL DISCHARGES_TO BL	16	65536
	BL CHARGES_TO BL	16	65536
	Size of the state space		**1,32211E+123**

Table 5.11: Size of the state space for each Level of Detail.

For example, referring to Table 5.11, the predicate "IS_VISIBLE" is used to indicate that an OBJECTIVE is visible from a certain WAYPOINT. Table 5.10 tells us that our world state contains 8 different objectives and 9 different waypoints, so we can have $8 \times 9 = 72$ different relations.

Since each relation can either be true or false, and the states are composed by collections of relations, we can encode a state of the environment with a bit string. So, the number of possible states in the environment is exponential with respect to the relations. Referring to the previous example, we would have 2^{72} possible states just for the "IS_VISIBLE" predicate.

To get the number of all the possible states, we have to compute the product of all the combinations. We can see from Table 5.11, that reducing the level of detail greatly shrinks the size of the state space.

In reality however, we're not directly instantiating states. We are rather traversing the state space by applying actions from the transition model. So, even if the size of the state space is important, we want to know how many actions we are exploring at each level of detail:

Level of Detail	**Expanded Actions**
1	524
2	522
3	16770

Table 5.12: Expanded Actions for each LoD.

Interestingly, we can see that we expand more actions in the first level of detail compared to the second, even if it should have more detail, thus more complexity. The reason behind this is that the complexity is represented by just more constraints: we don't add new types or entities; we just modify the preconditions of the "MOVE" action to account for obstacles. But we can clearly see that the number increases greatly at the third level of detail, by a factor of 32.

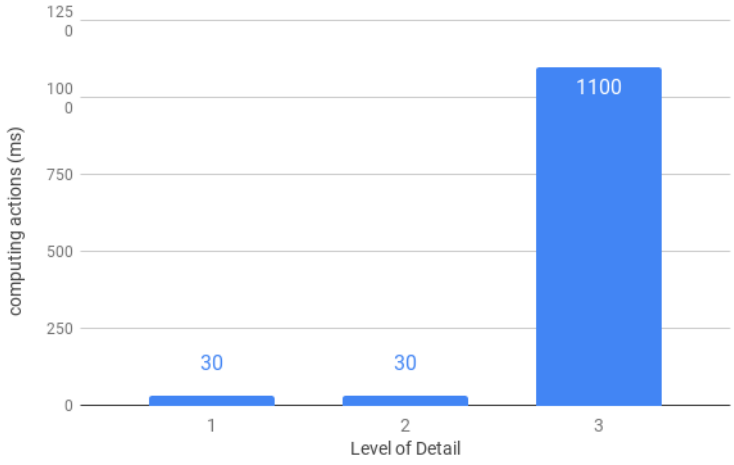


Figure 5.14: Time for computing actions at each LoD.

Hardware:

- CPU: Intel core i7-4700mq cpu @ 2.40ghz
- GPU: NVIDIA GeForce GT 750M with GDDR5 2GB VRAM
- RAM: 8GB DDR3
- HDD: 256GB SSD
- OS: Windows 10

As shown in Figure 5.14, the number of actions computed at each level of detail is strictly tied to the performances of the system. In fact, in the first two LoDs we only require around 30 ms to compute the possible actions. This lets us run the simulation consistently at 60 FPS. When we switch to the third level of detail, the system requires 1100ms to compute the actions, during that time the simulation is run at around 14FPS.

6. Discussion

The model that we presented in this dissertation focuses on the level of detail self-adjustment for simulations. The key aspects that we discussed where: (*i*) the design of a framework for modeling general purpose simulations; (*ii*) the design of algorithms for maintaining simulation consistency while adjusting its level of detail; (*iii*) and the implementation of an algorithm for switching the level of detail based on observability estimates.

The first goal was met by choosing PDDL as the underlying formalism around which we implemented our framework. This choice was made keeping in mind that PDDL is a well known standard for AI planning, thus making it a robust formalism to model simulations. With this formalism, we were able to model an arguably complex scenario, that presents challenges such as concurrency and synchronization. This required us to slightly modify a the standard framework 4.1.1.

The resulting model of the example simulation is able to express every detail that we wanted to be part of the simulation. This includes also cause-effect relations between various actions: all the actions at level of detail 3 have the side effect of discharging the battery of the agent that executes them.

The second goal, namely switching the level of detail of the simulation, was reached by devising an estimate of observability based on two parameters. Proximity is the distance between the player and the simulation. Visibility is the actual non-occluded portion of the user's field of view, with respect to the simulation. These two measures, even if they may not account for a complete estimate of observability, have the great advantage of being impartial. Even if a more complex model could be applied to estimate the observability of the simulation, they're often based on some estimate of attention, which could be subjective and biased by one's personal attitude.

Maintaining simulation consistency while adjusting its level of detail led us to implement algorithms for abstraction and refinement. As we can see in Section 5.1.4, the results of the refinement process applied to the example are rather convincing. All the actions that were selected from the transition model for simulating the environment at the first level of detail were then correctly refined at higher levels. Referring to our criteria for success in Section 2.5, we can see that consistency is maintained among all the levels of detail. The plans that were returned from the refinement process, to increase the level of detail, account for good alibis for each intelligent agent. The additional generated details also never conflict with previously observed features of the world. For example, while refining the first MOVE action from LoD 1 to LoD 2, the planner finds an alternative path, since the direct one cannot be traversed. All the resulting actions are compliant with the constraints of the world, and they do not contradict any previously observed fact.

Analogously, the abstraction process, as explained in the example in Section 5.1.5,

produces states that only keep the relevant part of the simulation at that particular level of detail. Everything else is deemed unnecessary, since it is not observable by the user, and gets dropped from the world state.

Finally, the gains in resources are appealing: from Table 5.11, we can see that we greatly reduce the size of the state space, if we keep a low level of detail. Table 5.12 confirms that a reduction of the size of the state space entails an overall simplification of the simulation. At LoD 1 we only need to compute $\frac{1}{32}$ of the actions compared to LoD 3, which is a decrease of 3200%.

Analogously, from Figure 5.14, we can see that the average time for computing the possible actions at level of detail 3 is 1100ms, which is almost 37 times higher than level of detail 2 (30ms). This is due to the fact that we have to check many more combinations compared to other levels, because the state space is larger.

6.1 Limitations

Part of the limitations of the system are related to the modeling capabilities offered by the current state of the framework. The Planning Domain Definition lLanguage is a really powerful tool for modeling planning problems: in a classic planning problem it is enough to design the transition model of the environment, enabling traversals of the state space. The lack of intent of the agents is offsetted by the clearly stated goal condition. The job of the planner is to find a path that ties the starting state to the goal, thus finding a solution to the problem. This characteristic of the planner compels the agents to chose actions that always maximize the performance measure towards the goal, thus giving a sense of rationality that makes up for the lack of explicitly stated intents.

This is not the case for simulators. In fact, the key difference between the two is that in the latter we don't have a clearly stated goal, because we are just simulating a process. Since we don't have a goal, we don't have a way to make agents behave rationally. For example, in the case of planners, we could be looking for a way of moving a rover between two distant waypoints. In that case, the initial state would be the one in which the rover is at the starting position, and the goal state the one in which it is at the destination. This means that we don't need to clearly state in PDDL that the intent of the rover is to reach the destination. It will be the planner's job to make it behave rationally, as a side effect of finding a solution to the problem. In similar scenario viewed from a simulation point of view, we would not have a clearly defined goal state, and so we would be bound to blindly apply the transition model to roll forward the simulation. Any other solution would require some other means of expressing intent.

Another limitation of the system is related to performance: even if we're saving a lot of resources performing the simulation at the first level of detail, we still need to deal with the extremely large state space of the third level of detail, when we perform refinement. The resources required for the refinement method are proportional to the increased size of the state space, thus the overall method could be very costly. The frequency at which the system needs to perform refinement is inversely proportional to the frequency at which the player changes their state. For this reason, as the size of the simulation grows, it might be more difficult for the player to change their state very often (e.g. the player has to travel large distances to change state). Thus the cost of refinement may not affect performances too much, because we might not need to

perform refinement very often.

Finally, there is a limitation regarding the interactiveness of the simulation. For now, the system does not support complex interactions between the user and the simulation. The only way that the player has, to control the flow of the simulation, is for him to allow or deny actions that get selected for execution.

6.2 Future Work

There are many ways in which we can improve the system. To improve the expressiveness of the PDDL framework, with regard to giving intelligent agents intent, we could integrate some probability model into it. For example, we could employ Bayesian networks to express a probabilistic transition model. This would represent the way agents behave inside the environment in a more realistic way. This would result in a perceived behaviour that would seem much more rational, compared to the random transition model. For example, the current state of the system could pick the IDLE action ten times in a row, because all actions are equiprobable. If we modeled some probability into the system, the agents would act accordingly, and it would be less likely for them to pick actions that are unrealistic.

We could also improve the switch between various levels of detail, by employing more complex estimates of observability. We could integrate algorithms that account for more subjective metrics, such as salience and the player's attention like discussed by Flores and Thue [15] or Sunshine-Hill [16]. This is because, sometimes, basing our estimates solely on objective metrics (such as distance and occlusion) may not be enough to have a complete grasp the player's perception of the world.

We can improve the overall performance of the refinement algorithm by employing heuristics that guide the search algorithm. This would make it expand as few states as possible, thus reducing the time and memory needed to find a solution. We could also implement a caching system that, each time a new state is encountered, stores all the possible actions that are applicable to it in a memory location. If we ever happen to re-visit the same state twice, the second time we would get the possible actions straight from the memory, without needing to compute them.

Another improvement to the refinement process could be introduced by estimating the memory of the user. If one particular aspect of the world has been simulated for a long time at a shallow detail level, because the user could not perceive it, it may not be essential to refine every single action. In fact, the user may not remember every detail from their last observation. If we manage to estimate the user's memory, we could limit our refinement step to the last k actions, requiring even less resources to keep the simulation believable.

7. Conclusion

Simulation level of detail self-adjustment hasn't been studied as much as its graphical counterpart. As simulations become continuously more complex and expressive, the computational needs required to model the behaviour of the intelligent agents that populate them rapidly increase. This dissertation proposes a possible approach aimed at decreasing the amount of resources required to conduct a simulation.

There are multiple challenges to tackle when devising a way to self-adjust the level of detail of a simulation. The first is that, as of now, there is no unified framework for modeling them. For this reason, the developed approaches are limited to their own scopes and implementations. It is also difficult to devise a method that is general enough to be suitable for different applications, while still presenting considerable advantages that are not based on context-specific optimizations.

We propose a system that could be a starting point for general level of detail self adjustment. The system is based on a method for automatically computing different levels of abstractions given an input problem. The underlying formalism of the system is a framework based on PDDL, which is a well-known standard for AI planning. The approach is based on the idea that we can save resources by avoiding the computation of details that the user cannot perceive. The system is also responsible for interacting with the user, displaying the resulting simulation and dealing with the received inputs.

In this dissertation I focused on my part of the project, which was to: (*i*) devise a way to switch the level of detail of the simulation; (*ii*) deal with the related abstraction and refinement issues; (*iii*) and manage the issues related to multi-agent environments. My contributions to these goals can be summarized as follows:

(*i*) We compute an estimate of the player's perception of the simulation, which is based on impartial metrics such as proximity and visibility, and then we adjust the level of detail accordingly. Our process for adapting the level of detail is based on the work of Sacerdoti which introduces the concept of planning in a hierarchy of abstraction spaces [11]. Similarly, in this dissertation, we use multiple abstraction spaces to simulate the environment at different levels of detail.

(*ii*) When we need to generate more details than the ones that had previously been computed, we **refine** the less detailed actions performed by the agents, up to the current point in time. This process is taken care of by a search algorithm, which traverses the state space to find a path leading from starting state to goal state. This is done for each node of the abstract solution. The result is a sequence of refined actions, which are equivalent to the abstract ones. Equivalent means that the detailed actions lead the simulation to a world state that has every relation from the abstract one, but also accounts for the additional constraints and features of the higher level of detail.

The opposite process is called **abstraction**, and is responsible for decreasing the level of detail of the simulation when the user is not able to perceive the additional

information. This process translates a detailed state of the system into an abstract one that requires less resources to be simulated.

(*iii*) Since we are dealing with multiple intelligent agents, we had to tackle issues of concurrency and synchronization. For the first one we devised a definition of parallelizability, seen in Section 4.6.1. For the second, we devised a system that synchronizes the refined plans, as explained in Section 4.6.2.

We tested our system on a non-trivial example, and the results are in line with our criteria for success. The framework was capable of modelling every aspect of the test scenario. The estimator switches the level of detail when the user's perception is within certain thresholds. The refined solutions comply with the constraints of the world at every level of detail. They never contradict any previously observed fact, thus maintaining consistency. The abstraction algorithm is capable of correctly dropping unnecessary components from a detailed state, thus translating it to an abstract level.

The results show that by abstracting the level of detail of the simulation, we are greatly shrinking the size of the state space. At level of detail 1 the size of the state space is $\sim 1.94 \times 10^{84}$, at second level $\sim 4.70 \times 10^{108}$, and at third level $\sim 1.32 \times 10^{123}$.

These estimates are reflected in the amount of possible actions computed by the system, at each level of detail. At level of detail 3 we are computing 16770 actions compared to the 524 actions of level 1. This means that we can reduce computation by a factor of 32.

In the future, we aim to improve the performance of the system. We would like to do so by introducing more efficient heuristic search algorithms that minimize the amount of expanded states. We also want to increase the interactiveness of the system, by allowing the users to perform more actions that affect the course of the simulation.

List of Figures

Bibliography

[1] Stephen Chenney, Okan Arikan, and David A. Forsyth. "Proxy Simulations For Efficient Dynamics". In: *Proceedings of Eurographics 2001*. Blackwell Publishers Ltd and the Eurographics Association, 2001, 10 pages.

[2] C. O'Sullivan et al. "Levels of Detail for Crowds and Groups". In: *Computer Graphics Forum* 21.4 (), pp. 733–741. DOI: 10.1111/1467-8659.00631. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/1467-8659.00631. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/1467-8659.00631.

[3] Christoph Niederberger and Markus Gross. "Level-of-detail for cognitive real-time characters". In: *The Visual Computer* 21.3 (Apr. 2005), pp. 188–202. ISSN: 1432-2315. DOI: 10.1007/s00371-005-0279-1. URL: https://doi.org/10.1007/s00371-005-0279-1.

[4] Cyril Brom, Ondřej Šerý, and Tomáš Poch. "Simulation Level of Detail for Virtual Humans". In: *Intelligent Virtual Agents*. Ed. by Catherine Pelachaud et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 1–14. ISBN: 978-3-540-74997-4.

[5] Sébastien Paris, Anton Gerdelan, and Carol O'Sullivan. "CA-LOD: Collision Avoidance Level of Detail for Scalable, Controllable Crowds". In: *Motion in Games*. Ed. by Arjan Egges, Roland Geraerts, and Mark Overmars. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 13–28. ISBN: 978-3-642-10347-6.

[6] Zuoyan Lin and Zhigeng Pan. "LoD-Based Locomotion Engine for Game Characters". In: *Technologies for E-Learning and Digital Entertainment*. Ed. by Kinchuen Hui et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 214–224. ISBN: 978-3-540-73011-8.

[7] David Osborne, Patrick Dickinson, et al. "Improving games AI performance using grouped hierarchical level of detail". In: *Proceedings of the Third International Symposium on AI & Games, Daniela M. Romano and David C. Moffat (Eds.), at the AISB 2010 convention, 29 March - 1 April 2010*. SSAISB, 2010, pp. 19–24.

[8] Felix Kistler, Michael Wißner, and Elisabeth André. "Level of Detail Based Behavior Control for Virtual Characters". In: *Intelligent Virtual Agents*. Ed. by Jan Allbeck et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 118–124. ISBN: 978-3-642-15892-6.

[9] Francois Cournoyer. *Massive Crowd on Assassin's Creed Unity: AI Recycling*. 2015. URL: https://www.gdcvault.com/play/1022411/Massive-Crowd-on-Assassin-s (visited on 10/07/2018).

[10] Sigurgrímur Unnar Ólafsson. "Computationally Generated Settlement Layouts". MA thesis. Reykjavik, Iceland: School of Computer Science, Reykjavik University, 2018.

[11] Earl D. Sacerdoti. "Planning in a hierarchy of abstraction spaces". In: *Artificial Intelligence* 5.2 (1974), pp. 115–135. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(74)90026-5. URL: http://www.sciencedirect.com/science/article/pii/0004370274900265.

[12] Kim Hamilton. *Introducing Generic HashSet*. 2006. URL: https://blogs.msdn.microsoft.com/bclteam/2006/11/09/introducing-hashsett-kim-hamilton/ (visited on 09/10/2018).

[13] A.G. Konheim. *Hashing in Computer Science: Fifty Years of Slicing and Dicing*. Wiley, 2010. ISBN: 9781118031834. URL: https://books.google.it/books?id=mU6fpT1sXCoC.

[14] Unity. *Compute shaders*. 2018. URL: https://docs.unity3d.com/Manual/ComputeShaders.html (visited on 10/06/2018).

[15] Luis Flores and David Thue. "Level of Detail Event Generation". In: *Interactive Storytelling*. Ed. by Nuno Nunes, Ian Oakley, and Valentina Nisi. Cham: Springer International Publishing, 2017, pp. 75–86. ISBN: 978-3-319-71027-3.

[16] Ben Sunshine-Hill. "Managing Simulation Level-of-Detail with the LOD Trader". In: *Proceedings of Motion on Games*. MIG '13. Dublin 2, Ireland: ACM, 2013, 13:13–13:18. ISBN: 978-1-4503-2546-2. DOI: 10.1145/2522628.2541250. URL: http://doi.acm.org/10.1145/2522628.2541250.

Acknowledgements

Contributors

Like every achievement that requires time and effort, the realization of this work could not have been possible without the help of many people who i wish to thank. My work is just a contribution to a larger project led by David James Thue, who deserves my many thanks for being an exceptional supervisor, always willing to invest time helping me overcome all the challenges that arose during the project, with his useful insights. I want to thank professor Andrea Polini for co-supervising the thesis, giving me valuable feedback that greatly helped shaping this dissertation. Finally, i want to thank both UNICAM and Reykjavík University for giving me the wonderful opportunity of working on a field i am passionate about.

Ringraziamenti

Ad Aurora, per aver accompagnato le mie risate nei momenti più felici ed aver asciugato le mie lacrime in quelli più tristi; ogni attimo è stato inestimabile.

A mia madre, per il costante supporto che dimostra verso tutto ciò che faccio, e per avermi sempre spronato ad inseguire i miei sogni.

A Gianmarco, per la sua preziosissima amicizia, e per essermi stato vicino nonostante la grande distanza.

Alla mia Famiglia, per l'incessante e caloroso affetto che mi è servito a raggiungere questo traguardo.

A tutti gli Amici che mi hanno affiancato durante questo percorso, per avermi aiutato a rimanere sano di mente.