

Università degli Studi di Camerino

---

SCHOOL OF SCIENCE AND TECHNOLOGY

Master Degree in Computer Science (Classe LM-18)



# Interactive Visualization for Hierarchical Models of Simulation

Student

**Matteo Altobelli**

Number **096777**

Relator

**Prof. David Thue**

Co-relator

**Prof. Michele Loreti**



# Abstract

One of the main problems when dealing with simulations is the correct visualization of what has been calculated has the next world state. A complex system manages the simulation process, and an equally complex state is produced. The objective of this project is to create a system able to correctly visualize the succession of the different states produced by the simulation. Since moving from one state to another is caused by actions, our problem can be reduced to the correct visualization of the actions and to the control of their success/failure. The entire project has been developed in Unity, the well-known game development platform.



# Contents

<b>Abstract</b>	<b>3</b>
<b>1 Introduction</b>	<b>7</b>
1.1 Background Notions . . . . .	8
1.2 Problem Formulation . . . . .	9
1.3 Criteria for Success . . . . .	10
<b>2 Related Work</b>	<b>11</b>
<b>3 Proposed Approach</b>	<b>15</b>
3.1 Planning Framework . . . . .	15
3.2 Simulation . . . . .	19
3.3 Visualization . . . . .	19
3.3.1 The GUI . . . . .	20
3.3.2 The "Gameplay" . . . . .	21
3.3.3 Errors Detection and Handling . . . . .	32
3.4 Summary of the Chapter . . . . .	33
<b>4 Discussion</b>	<b>35</b>
4.1 Future Work . . . . .	36
<b>5 Conclusion</b>	<b>37</b>



# 1. Introduction

As time goes by, video games are becoming increasingly complex systems with many interacting variables. To make sure that a game provides a satisfying experience, a consistent data visualization is required, particularly because the quality of a game directly relates to the experience a user gains from playing it.

Rendering takes care of this. With the term rendering we define the process that allows one to obtain, starting from a three-dimensional (3D) model elaborated on the computer, an artificial photorealistic image. More in detail, it is an image elaborated on the computer following a three-dimensional modelling based on project data; the geometric model created is covered with colours that are the same as real materials (textures) and illuminated using light sources that reproduce natural or artificial ones. It is obvious that the more complex the model is, the more the work necessary to create a realistic image is greater. Furthermore, the number of models that make up a scene contributes to make the rendering expensive.

But what happens when the amount of data to be displayed becomes increasingly high? Since in a video game very rarely we talk about single images, but rather of sequences of images (and therefore video), one of the main repercussions is the decrease of the frame-rate, which has the effect of making the video less fluid. Subsequently, to avoid an excessive loss of fluency, the quality of the images that make up the video sequence is reduced: an image that has a lower graphic resolution, requires less rendering work. The resolution, in computer science and graphic design, is the quantity that indicates the degree of sharpness or clarity of an image. [4] Therefore, a lower graphic resolution results in a loss of detail.

Many studies have addressed the topic of level of detail (LOD) generation. Efficient LOD generation techniques for point-based surface representations have been presented in the University of California Irvine, but they were not optimized on the bit-level for storage cost. [12]

Other studies have described techniques for improving the performance of image rendering by using CPU cycles going idle while the user is examining a static image on the screen. They proposed to convey the most information to the user as early as possible, with image quality constantly improving with time [1]. But it is evident that in a sequence of images, such a technique cannot be applied.

Even if solutions for using level of details for time dependent meshes have been given, [15] the problem of LOD generation when dealing with a large number of high-defined models has not been publicly addressed yet.

Our project was born with the purpose of giving a way to manage this problem. It aims to create an application that simulates the behaviour of a video game, which

then visualizes the results, and that is able to reduce the cost of the rendering process, making a selection of the objects to be rendered.

To do this, I and two other students used a hierarchical model to create a system that could be able to generate sets of game actions, given specific conditions. Since a video game can be considered as a succession of states, through these actions the system is able to recreate the progress of a game. The sets of actions are then "sent" to the class assigned to their visualization. Based on how the visualization ends, the current state of the game is updated and new sets of actions are produced.

The most important part of the whole project is represented by the actions. Through the **P**lanning **D**omain **D**efinition **L**anguage (**PDDL**), we have created the base structure of the actions.

The other students, then, developed a simulation system that takes into account the current state of the entire game world and the player's status, to produce new actions.

Instead, I focused on how to visualize all these actions, trying to make sense of their behaviour, and how to detect their result as new game states.

## 1.1 Background Notions

Before going further, it is good to explain what a hierarchical model is and why we chose it.

*"A hierarchical database model is a data model in which the data are organized into a tree-like structure."* [19]

A hierarchical model is a type of database; it determines the logical structure of a database and how data are stored, organized and manipulated.

There are four structural types of database management systems:

- **Hierarchical Model.** This design uses a one-to-many relationship for data elements. Its tree structure links a number of disparate elements to one "owner," or "parent", primary record. The speed of data search is high but not very flexible.
- **Network Model.** This model maintains the speed of search of the hierarchy model and increases its flexibility at the price of greater structural complexity. Its structure is, in fact, represented by an undirected graph. In this model there is no primary record, all the elements are equally important, and the links between tables are passable in both directions, without having to follow a particular search order as in the previous model.
- **Relational Model.** This model is based on set theory and first-order logic, and is structured around the mathematical concept of relationship. The relational model (less efficient in terms of search-speed with respect to hierarchical and network databases) was affirmed for its greater flexibility in data search. But it has some drawbacks. For example, Suri and Sharma (2011) [16] found that, though the relational database has an easy to design and use system, as the size of the database increases, it will slow the system down, and will result in performance degradation and data corruption.

- **Object-oriented Model.** In this model the information is represented in the form of objects as in object-oriented programming languages.

The choice of an appropriate database model depends on its use. For example, the structure of the states of a video game is very similar to a hierarchical structure, since from an initial state several new states are reached on the basis of the action performed. From these new states it will be possible to carry out actions that will determine other new states, and so on. Moreover, in a real-time system that needs a quick search among the various possibilities, speed is a key element. For this reason, among all the models, the hierarchical one is more suited to our needs.

## 1.2 Problem Formulation

How can we realize an “Interactive Visualization for Hierarchical Models of Simulation”?

The first topic to be addressed, when trying to answer this question, concerns the creation of a framework on which to run the simulation. The development of such a framework falls within the planning problem of Artificial Intelligence, as the simulation of a video game is about searching for actions that make the states of the world progress.

The planning problem, in Artificial Intelligence, is about the decision making performed by intelligent creatures when trying to achieve some goal. It involves choosing a sequence of actions that will (with a high likelihood) transform the state of the world, step by step, so that it will satisfy the goal.

Subsequently, the focus moves on the simulation of the game progress. This part not only depends on how the planning framework has been constructed, but also determines which actions need to be performed. In this phase, a tricky aspect concerns the construction of an algorithm that expands the hierarchical tree structure of the simulation in the most efficient way possible. Particular importance is given to the speed of expansion, as in a video game a fast response of the system is required.

Finally, there is the visualization of the actions generated by the simulation. The first step to solve this problem is the definition of a communication protocol between this part and the simulation. Such a protocol should be simple and "clean", i.e. without useless information.

The communication must be bidirectional: initially the simulation will communicate its results to the visualization, which will have the task of showing them, and then return the results of the actions to the simulation.

The most delicate part in this process is certainly the detection of the actions' results, in particular when an action is not successful. In order to check that an action has been carried out without any problem it is sufficient, in fact, only to check that the post-conditions of each of them (defined during the creation of the actions) are satisfied. In the opposite case, instead, waiting for the post-conditions to become true could require waiting for a very long (if not infinite) time, blocking the simulation that is waiting for an answer.

## 1.3 Criteria for Success

Based on the goals we set for ourselves, to be able to state that the project is appropriate, we have taken into consideration the following criteria:

- **Suitable protocol:** A protocol able to transmit only the necessary information not only allows a simpler implementation of the visualization, but makes the communication process less heavy, improving the reaction time between simulation and visualization.
- **Correct and efficient visualization:** To ensure a good gaming experience, we have identified two characteristics that the action visualization should satisfy: **correctness**, which guarantees that the visualization of the action is consistent with its meaning and that is displayed entirely; **efficiency**, which ensures that the generation of actions and their visualization is synchronized and that it takes place within an acceptable time.
- **Reliable action result detection:** Particular importance in the visualization process must be placed on the detection of each action's result. The main thing to avoid, in this process, is that the simulation is suspended because for some reason the visualization is not able to assign a value to the outcome of one or more actions.

## 2. Related Work

Summarizing Kelly et al's review [9], we can say that early video games used scripts and finite state machines (FSMs) to create and control the behaviour of NPCs. Some well-known titles, as the QUAKE series or the Blizzard Software's WARCRAFT III, were using finite state machines in their algorithms. Some more recent games, like HALO 2, applied the concept of Hierarchical Finite State Machines (also known as Hierarchical State Machines, HSMs), that differ from the classical FSMs for the introduction of hierarchically nested states. [7] The biggest problem when using FSMs is that their complexity grows quickly when the number of NPCs grows, or their behaviour become more complicated. Thus, it is evident that the use of FSMs is not suitable for the last video games developed, in which often there is a huge number of NPCs and in which we also want to achieve a behaviour more similar to the human one. [11] Scripts are programs written in a particular class of programming languages, called scripting languages. They are used in a special run-time environment that automate the execution of tasks. Various games used scripts to model NPCs (Bioware's NEVERWINTER NIGHTS, or Epic Games' UNREAL TOURNAMENT), but they have properties similar to FSMs: their complexity grows quickly when NPCs and the world become more complex.

Planning technology brought new techniques in Game AI for building more intelligent non-player characters, and thus more realistic NPCs, catching the attention of developers and researchers. Even if there are still a lot of problems and questions unanswered, planning technology has made huge steps forward in these last two decades, since being introduced in the games industry.

Alex J. Champandard's article [2] helped us to look back in the past at the most notable techniques used for planning task in video games.

The first game known to make use of planning techniques is F.E.A.R. (First Encounter Assault Recon). The AI that made the enemies relies on a STRIPS-style planner to search through all the possible actions to find a sequence of actions that could generate a state that matches with the goal criteria. STRIPS is a planning algorithm that uses the proposition logic extended with predicates and a search mechanism in the state space, to obtain a possible sequence of actions that, if performed, causes the achievement of a final state of the world starting from an initial one. It is effective in all those cases where the problem and the environment are known a priori with certainty, but has some limitations: the impossibility of managing differences between model and environment; changes to the system must always and only occur as a result of the intelligent agent, so it is unsuitable for cooperation or interaction in general; increasing the complexity of the model, by introducing new variables and operations, the amount of calculation time needed to plan an operation grows exponentially.

---

In the last ten years, there has been a strong spread of more hierarchical techniques over STRIP-style planning: **Hierarchical Task Networks (HTN)** planners and behavior trees.

The HTN planner is a method of planning based on the hierarchical decomposition of actions on different levels of abstraction. Higher Level Actions (HLAs) allow you to analyze planning at a higher level of abstraction. The Lower Level Actions (LLAs), on the other hand, make it possible to analyze in detail all the single steps of a particular plan. The lower level actions are called primitive actions and are characterized by less abstraction. In an HTN, the solution of a problem is sought by first analyzing the high-level actions, without having to analyze the details of each action and each task. Once the high-level action sequence has been identified to solve the problem, it is possible to analyze the lower level actions during the execution of the planning.

Behavior Trees (**BTs**) are "mathematical models of plan execution used in computer science, robotics, control systems and video games". [17] They are trees of hierarchical nodes that control the decision making process of an AI. BTs arrive at their goal by breaking it down into smaller tasks. This programming style facilitates the generation of goal-based behaviors, in contrast to the approach used by the FSMs. BTs are a very easy form of planning, but because they do not consider explicitly the future effects of current actions, relying only on a set of pre-established behaviors, falls within the definition of *Reactive Planner*.

Hierarchical planners have been implemented so that they can include patterns that make them very similar to the industry-standard behaviour trees. [2]

Our system uses the Planning Domain Definition Language (PDDL). It is a recent attempt to standardize planning domain and problem description languages. It was developed mainly to make the International Planning Competition (IPC) series possible, and was inspired by STRIPS and ADL planning systems among others. The PDDL allowed us to model the system such that it can provide the necessary information to execute actions.

In Interactive Storytelling [5], detecting and responding to unanticipated user activity is a central point. Narrative Mediation is used to preserve the coherence of the narration. Riedl, Young and Saretto (from now referred as "RYS"), in their work, [14] defined two types of narrative mediation:

- **Accommodation.** It integrates exceptional user actions into the narration through the re-planning. A narrative plan can be analyzed and preemptively revised in order to anticipate and accommodate exceptions.
- **Intervention.** It is the substitution of a user's exceptional action with an instance of an action called a *failure mode*. The failure mode action is similar to the exception action, but its effects produce results that do not conflict with any of the causal structure in the plan.

Narrative mediation is an effective method for managing the interaction between human and computer-controlled agents in a narrative setting. However, there are several limitations to the approach defined by RYS. Two central limits are:

- **Locality of exceptions.** It occurs when a user executes a sequence of actions in

---

preparation for the exceptional act. A system that made effective predictions about the plan being executed might find opportunities for intervention or accommodation before potentially extreme responses to exceptions are required.

- Locality of *intervention*. Intervention involves the substitution of a single user action in place of the action that a user intends to perform. However, it is possible that a sequence of actions could be executed in order to intervene.

Halldórsson, Gylfason and Gunnlaugsson, [6] tried to expand RYS research, defining more general ways of failure, and proposing a way to handle them. We will explain better their work on the section about handling failures.



## 3. Proposed Approach

The project has been developed through Unity. Unity is a multi-platform game engine that allows the creation of interactive 3D content with ease. It has an excellent functionality, a high-quality content, and the ability to be used for pretty much any type of game. Moreover, it is a well-known engine for us, as we have had the opportunity to study it thoroughly during our double degree program in Iceland, while any other engine, however good it may be, would have required a period of time to understand the functioning and get familiar. With the last updates, Unity started to support only the C# (pronounced "C sharp") language.

C# is an object-oriented programming language developed by Microsoft within the .NET initiative. The syntax and structure of C# are inspired by various languages born previously, in particular Delphi, C++, Java and Visual Basic.

The project can be divided into three sections, connected to each other:

- Planning framework
- Simulation
- Visualization

### 3.1 Planning Framework

A video game can be considered as a succession of discrete states, in which actions define the rules for going from one state to another. In fact, starting from a generic initial state it is possible to reach a new state only if an action (or a series of actions) is performed.

In our project, each state is defined as a set of relationships. Our definition of relationship is quite analogous to the mathematical definition: it binds one or more entities through a predicate.

We defined two types of relations:

- Binary
- Unary

A binary relation is a relationship that connects two entities through a predicate, while a unary relation is a relationship that involves only one entity and a predicate. In the latter case, rather than talking about a relationship, we are talking about a property that is valid for the (only) entity involved. Both types of relations have a parameter

called *RelationValue* that can be TRUE or FALSE, depending on if they are valid or not for a certain state.

For simplicity, we choose to use only these two types of relationship (avoiding those of higher degrees, i.e. ternary, quaternary, etc.). Here is an example for each one of them:

- *Handempty(rover1)*: this unary relation tells that rover1 has its robotic hand empty, which means that it is not holding anything.
- *At(rover2, waypoint7)*: this binary relation says that rover2 is *at* waypoint7.

After having defined states, we needed a way to determine the transitions from one state to another. We decided to use the **Planning Domain Definition Language (PDDL)** for this task.

A PDDL planning task is made of:

- **Objects**: Things in the world that interest us. We called these elements *Entities*.
- **Predicates**: Properties of objects that we are interested in; can be true or false.
- **Initial state**: The state of the world that we start in.
- **Goal specification**: Things that we want to be true. In our project, a goal state is not defined, because we want to consider performing every possible action, among the ones defined.
- **Actions**: Ways of changing the state of the world.

The term *Entity*, already introduced, takes on a meaning similar to the concept of entity in the Entity-Relationship model (E-R model). It represents classes of objects (things, people, etc.) that have common properties and autonomous existence for the purpose of the application of interest. Thus, entities are grouped by *Entity-Type*. An occurrence of an entity is an object or instance of the class that the entity represents.

In our relations, *Predicates* are a link between entities; they represent the "verb" or the "adjective" of every relationship. As for the relations, We defined two types of predicates: binary and unary. In predicates, the equivalent of entities in relationships are the Entity-Types.

One thing that is important to understand is that predicates in a domain definition have no intrinsic meaning. Within the domain definition is specified only what the predicate names are, their number of arguments, and the argument types. The "meaning" of a predicate, in the sense of for what combinations of arguments it can be true and its relationship to other predicates, is determined by the effects that actions in the domain can have on the predicate, and by what instances of the predicate are listed as true in the initial state of the problem definition.

Both Relations and Predicates are defined through an interface. In the C# language, an interface contains only the declaration of methods, events, and properties, but not their implementation. The class that implements the interface has the task of implementing all the members of the interface. The main contribution of an interface is that it makes it easy to maintain a program.

An interface can be defined using the **interface** keyword. Following are the two definitions of our interfaces:

---

```

1 public interface IRelation
2 {
3     Entity Source { get; }
4     IPredicate Predicate { get; }
5     RelationValue Value { get; set; }
6     bool EqualsThroughPredicate(IRelation other);
7     bool EqualsWithoutValue(IRelation other);
8     bool Equals(object obj);
9     IRelation Clone();
10    int GetHashCode();
11    string ToString();
12 }
```

---

```

1 public interface IPredicate
2 {
3     string Name { get; }
4     EntityType Source { get; }
5     IPredicate Clone();
6     bool Equals(object obj);
7     string ToString();
8 }
```

---

Different classes can implement the interfaces using <Class Name> : <Interface Name> syntax. These classes need thus to provide an implementation of the methods and properties of the interfaces they are implementing. An example is now given:

---

```

1 public class BinaryPredicate : IPredicate, System.IEquatable<IPredicate>
2 {
3     private string _name;
4     private EntityType _source;
5
6     ...
7
8     public string Name
9     {
10         get { return _name; }
11     }
12
13     public EntityType Source
14     {
15         get { return _source; }
16     }
17
18     ...
19
20     public IPredicate Clone()
21     {
22         return new BinaryPredicate(_source.Clone(), _name,
23                                     _destination.Clone());
24     }
25
26     public override bool Equals(object obj)
27     {
28         if (obj == null)
29             return false;
```

---

```

29
30         if (obj.GetType() != typeof(BinaryPredicate))
31             return false;
32
33         BinaryPredicate otherPredicate = obj as BinaryPredicate;
34         if (_source.Equals(otherPredicate.Source) == false)
35             return false;
36         if (_name.Equals(otherPredicate.Name) == false)
37             return false;
38         if (_destination.Equals(otherPredicate.Destination) == false)
39             return false;
40         return true;
41     }
42
43     public override string ToString()
44     {
45         return _source + " " + _name + " " + _destination;
46     }
47     ...
48 }

```

---

Following the PDDL structure, the *initial state* is defined inside the Domain. For our specific case, the Initial state is formed by the following relations:

- IS\_CONNECTED\_TO(wayPoint1, wayPoint5)
- IS\_CONNECTED\_TO(wayPoint2, wayPoint5)
- IS\_CONNECTED\_TO(wayPoint3, wayPoint6)
- IS\_CONNECTED\_TO(wayPoint4, wayPoint8)
- IS\_CONNECTED\_TO(wayPoint5, wayPoint1)
- IS\_CONNECTED\_TO(wayPoint6, wayPoint3)
- IS\_CONNECTED\_TO(wayPoint6, wayPoint8)
- IS\_CONNECTED\_TO(wayPoint8, wayPoint4)
- IS\_CONNECTED\_TO(wayPoint9, wayPoint1)
- IS\_CONNECTED\_TO(wayPoint1, wayPoint9)
- IS\_CONNECTED\_TO(wayPoint3, wayPoint4)
- IS\_CONNECTED\_TO(wayPoint4, wayPoint3)
- IS\_CONNECTED\_TO(wayPoint4, wayPoint9)
- IS\_CONNECTED\_TO(wayPoint5, wayPoint2)
- IS\_CONNECTED\_TO(wayPoint6, wayPoint7)
- IS\_CONNECTED\_TO(wayPoint7, wayPoint6)
- IS\_CONNECTED\_TO(wayPoint8, wayPoint6)
- IS\_CONNECTED\_TO(wayPoint9, wayPoint4)

- AT(rover1, wayPoint6)
- AT(rover2, wayPoint6)
- IS\_IN(sample1, wayPoint2)
- IS\_IN(sample2, wayPoint9)
- IS\_IN(sample5, wayPoint3)
- IS\_IN(sample2, wayPoint3)
- IS\_IN(sample4, wayPoint8)
- IS\_IN(sample6, wayPoint3)
- IS\_EMPTY(rover1)
- IS\_EMPTY(rover1)

*Actions* define how to progress in the game. An action has:

- A **Name**, used as identifier.
- **Pre-Conditions**, that need to be satisfied in order for the action to be performed.
- **Post-Conditions**, that affect the world state once the action is finished.
- **Parameters**, which are the entities involved in the action.

Pre-conditions and post-conditions are lists of relations, binary and/or unary.

## 3.2 Simulation

The simulation has the task of exploring the actions that can be performed for any given state. It checks which relationships are currently true, compares them with the preconditions of each defined action, and then selects the actions that can be performed (those whose preconditions are met). Among these, one is chosen for each rover and then sent to the visualization. Thus, the simulation allows the progress of the game, generating actions.

## 3.3 Visualization

To test the simulation, a scene was created with Unity. We discussed the choice of a scene that could be adapted to our project. Initially it was thought to recreate the behaviour of two villages with a player who travels from one to another. But, we needed an example that could show our results directly and clearly, so we opted for a simpler scene with fewer elements that could not confuse the observer.

At the end, we came up with a scene in which the player, represented by a satellite, is moving between two planets. Each planet has two rovers that are performing the actions produced by the simulation. Depending on which planet the player moves toward and

his distance from it, they can see a certain amount of details, until they get to influence the actions of the planet's rovers.

Let's begin the description of how the visualization is structured from the graphic interface.

### 3.3.1 The GUI

A **Graphic User Interface (GUI)** allows users to navigate through the applications of a computer. [20]

There are studies that have identified basic psychological factors to take into consideration when designing a good GUI [8]:

- the physical limits of **visual acuity**: it is one of the main visual abilities of the visual system and is defined as the ability of the eye to solve and perceive fine details of an object and depends directly on the sharpness of the image projected on the retina. [3]
- the limits of **absolute memory**: it refers to the fact that there is a limit to the amount of information that a person can process at one time. Through his work, the psychologist George A. Miller showed that “the span of absolute judgment and the span of immediate memory impose severe limitations on the amount of information that we are able to receive, process, and remember”. [10]
- the principles of grouping (or **Gestalt laws of grouping**): they are a set of principles in psychology that explain that human beings naturally perceive objects as models and organized objects. [13]

Figure 3.1 the first view when entering the “game”.

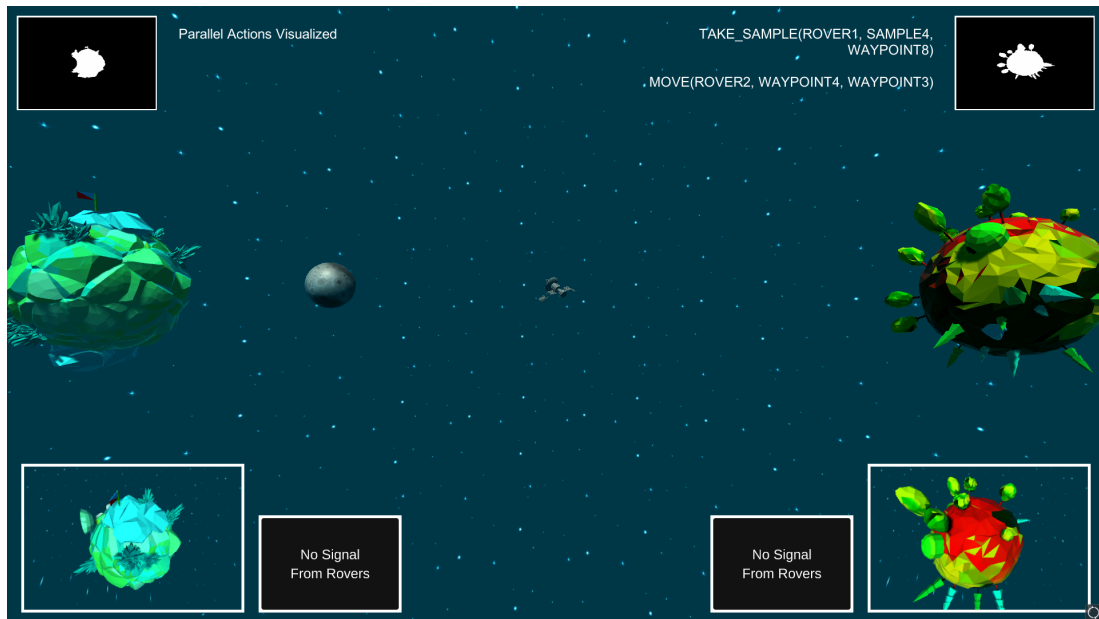


Figure 3.1: Game's First View.

As can be noticed immediately, the scene appears divided into two parts by an imaginary vertical line placed at the centre of the screen and, with respect to this line, the elements appear arranged following a reflection symmetry (or mirror symmetry, or line symmetry). This technique, shown in Figure 3.2, has been used to create a sort of logical division between the two planets and the information concerning them, and exploits the *Proximity* principle belonging to the Gestalt laws. This principle states that within the same scene or image, the elements close to each other are perceived as a unitary element. In this way, users can easily locate the information they are looking for.

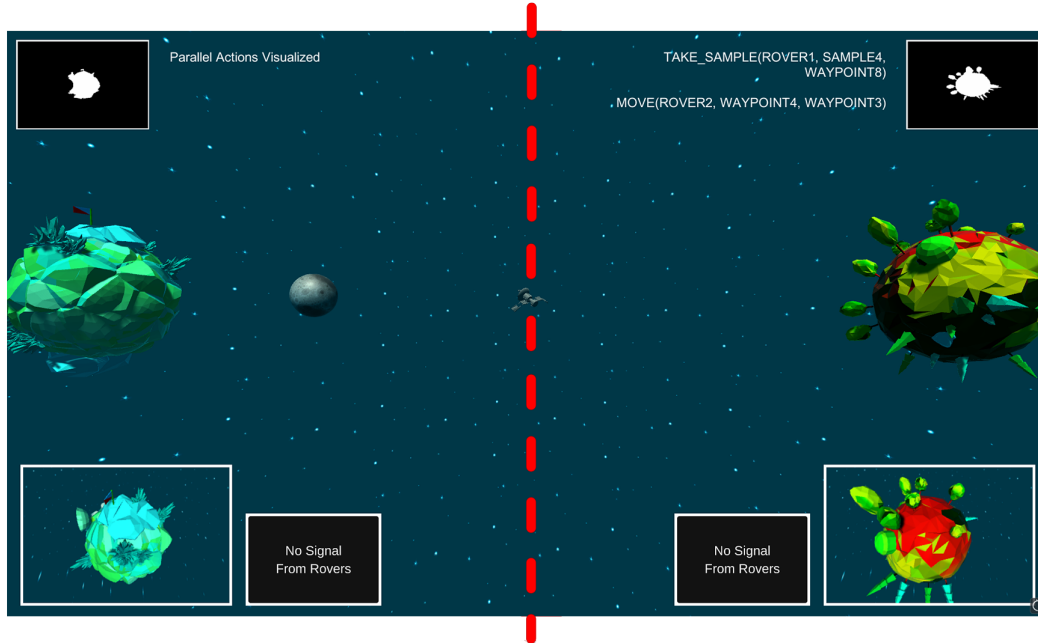


Figure 3.2: Representation of Gestalt laws of grouping.

As the player starts moving and, thus, interacting with the space, the GUI changes, showing or hiding elements.

### 3.3.2 The "Gameplay"

Because of the nature of the project, which is not focused on the creation of a video game, but on the correct display of a simulation of states, the level of interaction is limited. Initially, the user can only move the satellite horizontally to the right or left. The closer the satellite gets to one of the planets, the more details will be shown on the screen.

The two planets, the moon, and the artificial satellite are the only actual 3D objects that can be seen when the application is initially run. Their movements (respectively, rotation, revolution, and translation) are made by acting directly on the Transform component attached to these game objects. Figure 3.3 sums up the initial possible interaction. At this point, some explanation is necessary. Every element inside the scene is a game object; `GameObject` is the base class for all entities in Unity scenes. Game objects are made of components; a `Component` is the base class for everything attached to `GameObjects`. Finally, every object in the scene has a `Transform` component. It

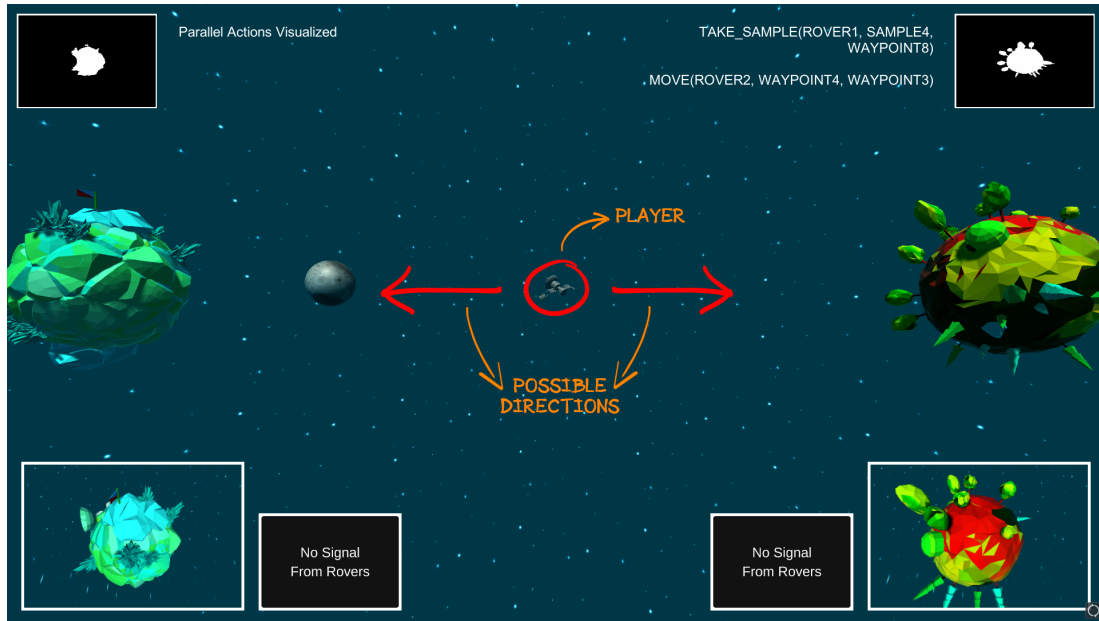


Figure 3.3: The initial possible Interaction.

stores the position, rotation and scale of the object, and is also used to manipulate these parameters. [6] A lot of other components exist in Unity (Image, Text, Audio Source/Listener, Mesh Renderer, and so on...) and are used to define characteristics for game objects. The ones used for this project will be explained when encountered.

Here is a list of all the 3D game objects rendered in our scene:

- Artificial satellite, which represents the player.
- Two planets. We decided to call them Venus (the one to the left) and Mars (the one to the right).
- A moon. This object has the effect of reducing the vision of Venus to the player
- 4 rovers, 2 for each planet.
- 18 waypoints, 9 for each planet.
- 2 terrains that represent the planet's landscape.

The terrains are Navigation Meshes in the Unity Navigation System. This system allows one to create objects which can navigate the game world. The Unity NavMesh system consists of the following pieces:

- **NavMesh** (that stands for *Navigation Mesh*) is a data structure which describes the walkable surfaces of the world and allows to find path from one location to another. The data structure is built, or *baked*, automatically from the level geometry.
- **NavMesh Agent** is a component with several functions that allow the component to which it is attached to navigate the scene using the NavMesh.

- **The Off-Mesh Link** component allows to add navigation shortcuts which cannot be represented using a walkable surface. For example, jumping over a pit or opening a door before walking through it.
- **The NavMesh Obstacle** component allows to describe moving obstacles the agents should avoid while navigating the world.

Each rover has a NavMesh Agent attached to it. The Off-Mesh Link and NavMesh Obstacle components are not used in this project.

We can now go back describing the “game”.

Each planet has different levels of detail, depending on the distance the player is away from them. Entering a higher LOD (which can be translated as “moving closer to one planet”) means having increased and more detailed information.

Figure 3.4 shows the view of LODs of our two planets from the editor.

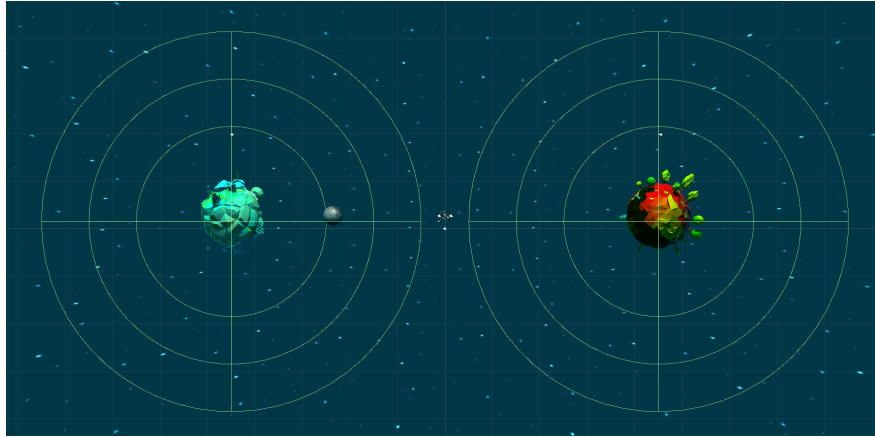


Figure 3.4: Editor’s view of LOD.

As soon as the player (as the satellite) enters the first LoD, the connection is made with the rovers of the planet to which it is moving. Following the logical division of the screen, the two rectangles placed below the chosen planet are activated. These game objects simulate the screens that transmit the view of the two rovers. They are shown in Figure 3.5.

Henceforth, the explanation begins to go down into detail and the terminology will become more and more specific. I will try to make it clear for everybody.

What we need to come up with is a rectangle that transmits what is happening on the planet, live. As it occurs in the real world when live images of a faraway place are required, a camera that captures them is necessary. Thus, each rover has a Camera object that follows it, as shown in Figure 3.6.

The images collected from the camera are then recorded in the form of a Render Texture. A Render Texture is a special type of Texture that is updated at runtime; instead of being a fixed texture, its content is modified every frame with what is captured by the camera. To do this, a camera is designated to render into it.

Now that there is a way to record what happens on the planets, we need to create a system that can visualize these images, a sort of screen. Our screen will be a simple rectangular game object with a RawImage component attached. We used this component, instead of a normal Image component, because the latter requires the texture to be a



Figure 3.5: Cameras' screens

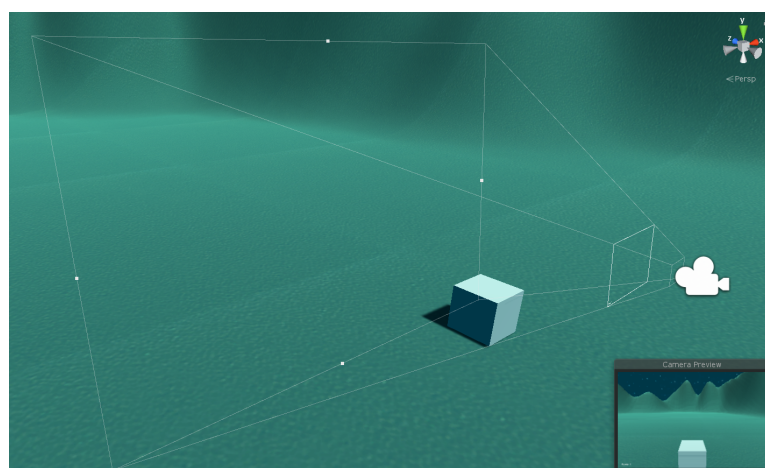


Figure 3.6: Editor's view of a rover's camera.

Sprite, while the RawImage can accept any kind of texture, including RenderTexture, which is exactly what we were looking for.

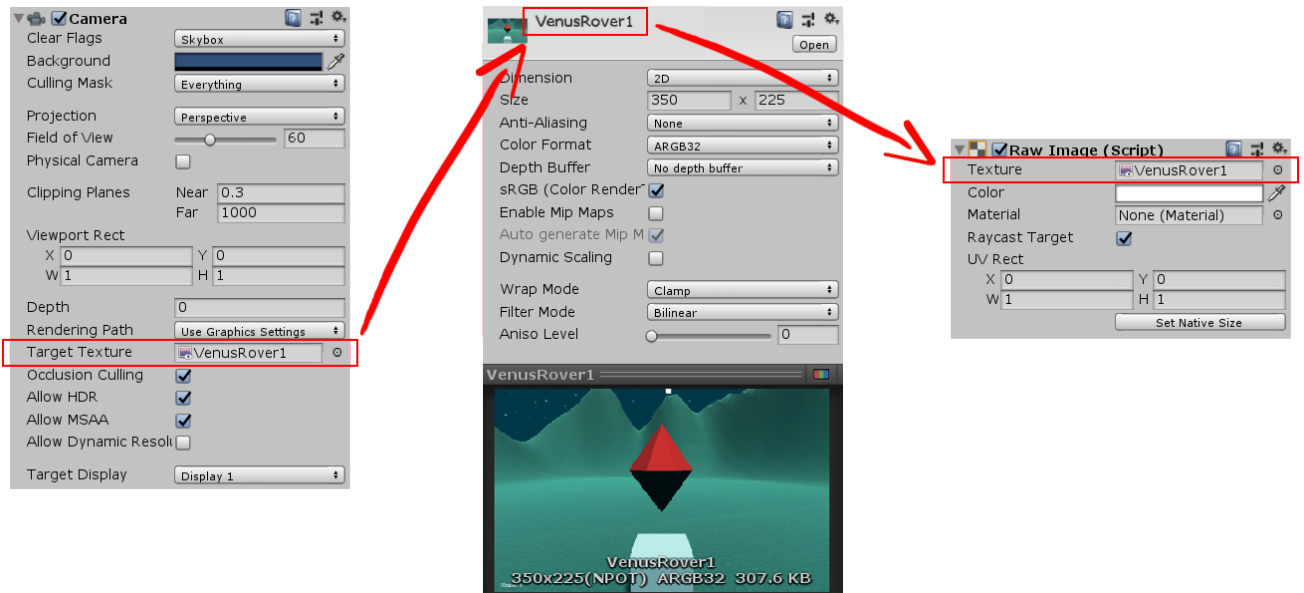


Figure 3.7: Cameras Rendering Scheme

Figure 3.7 summarizes what has been explained previously. It shows that the Camera (a) is rendering into the RenderTexture (b), which is then used as texture for the RawImage component (c) attached to the rectangle that simulates the screen.

Another change that can be seen when entering the first LoD is the appearance of two icons, one for each rover live screen. These icons, representing a container, belong to the inventory game object and are accompanied by other icons that appear when the LoD increases.

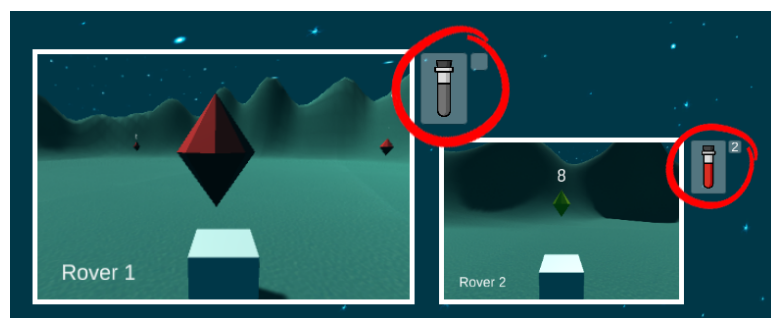


Figure 3.8: Samples Icon.

As shown in Figure 3.8, there are two possible states for the sample icon:

- **Disabled**, in which the icon is in a scale of grey. It means that the rover does not hold any sample.
- **Enabled**, in which the icon is coloured. It means that the rover is holding a sample.

The number that appears on top-right of the icon is the unique number of the sample held (it is empty when no sample is held). Other icons related to the samples will be introduced and explained when talking about the action representation.

When the player's satellite enters the second LoD, another icon appears inside the inventory game object. This icon represents the battery level of the rover.



Figure 3.9: Battery icons.

Figure 3.9 shows the battery icons in the inventory, while Figure 3.10 shows their possible different levels: Full, Medium, and Low.



Figure 3.10: Battery levels.

Entering the third LoD, in which there is the maximum level of information, allows the player to interact directly with the rovers' actions, deciding if they should be performed or not. For this purpose, the system will show a panel that asks about allowing (or not) the actions chosen by the simulation, as can be seen in Figure 3.11. If the user does not make a choice, the system will choose automatically, after a fixed period of time, to NOT perform the actions.

Now that the explanation of the interface part of the project (i.e. how it appears to the user and the possible interactions that may exist) is complete, we can move on to the description of the design and development of the source code behind the visualization.

Most of the logic of the visualization is included in a class that, for this reason, is called *Visualization*. It derives from *MonoBehaviour*. *MonoBehaviour* is the class from which all the components of a game inherit. It is essential to talk about this class, since it allows one to associate to each object in the scene some particular behaviours when certain conditions occur. Conditions are represented by the methods of the *MonoBehaviour* class in the form of events, to which each object can react. Whenever a specific event occurs, Unity automatically calls the method that handles it. At this point, for an object to react to an event, there must be a function assigned to it, in which will be written the behaviour of our object. This behaviour is a set of instructions necessary to react to the event. At each frame, Unity runs all the scripts active on the scene,



Figure 3.11: Interaction panel.

and if it finds one of these predefined functions, it calls it (passing the control to the function). At the end of the execution, the control is returned to Unity.

The functions that are used to manage events are typically called event handlers, but in Unity they take the name of event functions, for simplicity. Among these functions, two assume a relevant role, so as to be automatically created (even if empty, i.e., without any instruction) on the creation of each MonoBehaviour script. These functions are:

- **START:** it is called only once at the beginning of the life cycle of an object. it is useful to perform operations of preparation for the game, such as creating arrays that will be filled later, or searching for elements in the scene whose reference we want to save and then work on it later during execution.  
In any case, Unity will NEVER call Start more than once in the entire life cycle of a script.
- **UPDATE:** it is called every frame just before it is rendered, which makes it dependent on the frame-rate. It is often used to update the position of moving objects (through operations on their Transform and/or Rigidbody components), so that they are rendered in the new position giving the player the illusion of movement. In addition, checks on player input are usually performed in the Update. For example, many features of the Input class are designed to read the player's inputs in that given frame. It is important to understand that everything contained in the Update is executed completely in the space of a frame, before the player can see anything.

The Visualization class contains two main methods, *visualize* and *interact*, which are **Coroutines**. Let's see what the difference is between a normal function and a coroutine. The normal behaviour when a function is called is that it runs to completion before returning. This means that every action taking place in a function must be done within a single frame. Thus, a function call cannot contain an animation or a sequence of

events over time, because they take more than one frame to be performed. As an example (which is not used in our project, but is very explanatory), consider the task of gradually reducing the opacity of an object until it becomes completely invisible.

---

```

1 void Fade() {
2     for (float f = 1f; f >= 0; f -= 0.1f)
3     {
4         Color color = renderer.material.color;
5         color.a = f; //Color.a returns the alpha component of the color
                      //(0 is transparent, 1 is opaque).
6         renderer.material.color = c;
7     }
8 }

```

---

As it stands, the Fade function will not have the expected result. In order for the fading effect to be visible, the alpha must be reduced over a sequence of frames to show the intermediate values being rendered. However, the function will execute entirely within a single frame update. The intermediate values will never be seen and the object will disappear instantly.

It is possible to handle situations like this by adding code to the Update function that executes the fade on a frame-by-frame basis. However, it is awkward to manage more complex situations with the Update function, in which more than one function that require several frames to be completed, need to be executed at the same time.

A coroutine is like a function that has the ability to pause execution and return control to Unity but then to continue where it left off on the following frame. In C#, a coroutine has to be declared with a return type of IEnumerator and with a yield return statement included somewhere in the body.

IEnumerator is a .NET type that is used to fragment large collection or files, or simply to pause an iteration. The yield return line is the point at which execution will pause and be resumed the following frame. To set a coroutine running, the StartCoroutine function is used:

---

```

1 StartCoroutine("Fade");

```

---

It is important to notice that any variable or parameter used by a coroutine will be correctly preserved between yields.

By default, a coroutine is resumed on the frame after it yields, but it is also possible to introduce a time delay using WaitForSeconds. This can be used as a way to spread an effect over a period of time, but it is also a useful optimization. Many tasks in a game need to be carried out periodically and the most obvious way to do this is to include them in the Update function. However, this function will typically be called many times per second. When a task doesn't need to be repeated quite so frequently, you can put it in a coroutine to get an update regularly but not every single frame:

---

```

1 yield return new WaitForSeconds(1.0f);

```

---

Last thing to be aware about Coroutines is how to stop them. They are not stopped when a MonoBehaviour is disabled, but only when it is definitely destroyed. Coroutines

can also be stopped using `MonoBehaviour.StopCoroutine` and `MonoBehaviour.StopAllCoroutines`.

Now that we have explained what coroutines are and how they behave, we can continue with the description of the two main Visualization methods. Both are called by the simulation and to both is sent a `HashSet` of actions. In particular, when the LOD is less than or equal to 2, the coroutine `visualize` is called. When the LOD is maximum (i.e. equal to 3), the `interact` coroutine is called.

When the `visualize` method is invoked, it takes each of the actions in the `HashSet` and executes a new coroutine for each one of them, then waits for a response from the coroutine just executed:

---

```
1 foreach(Action a in actions)
2 {
3     Coroutine newCoroutine = StartCoroutine(visualizeSingleAction(a));
4     listOfStartedActions.Add(newCoroutine);
5     yield return null;
6 }
```

---

As can be seen from the snippet, the coroutine that is called for each action is `visualizeSingleAction`. This coroutine has, in fact, the task of showing the effects of every single action that is given to it. It checks the name of the action received and, based on this, produces a specific effect.

Six possible actions have been defined:

- IDLE
- MOVE
- TAKE\_SAMPLE
- DROP\_SAMPLE
- TAKE\_IMAGE
- CHARGE\_BATTERY

For each action (except for IDLE) the script checks which entity has the role ACTIVE. This will be the subject of the action. Let's now look at the behaviour of each individual action in detail.

### **IDLE**

If the input is an IDLE action, the coroutine will wait for 2 seconds without doing anything.

### **MOVE**

The display of the MOVE action is the most complex. When the coroutine receives this action, it first calculates the distance between the entity that is ACTIVE and the destination to be reached. Based on the subject's movement speed, the execution time of the action is then estimated according to the relationship:  $\text{time} = \text{space} / \text{speed}$ .

To understand which of the possible destinations is the one to which the action refers, the relations that form the post-conditions of the action are analysed. Among these,

the one with the "AT" predicate and the RelationValue equal to TRUE is taken. The entity DESTINATION of this relationship will in fact be the destination to which the ACTIVE entity will have to move.

Once the active rover and the destination waypoint are known, a path that makes the rover move from its position to that of the waypoint can be created by executing the following command:

---

```
1 agent.SetDestination(destination.transform.position);
```

---

The rover then starts moving: if it reaches the destination within the initially estimated time, the coroutine *visualizeSingleAction* returns a value of true, otherwise, the value false is returned.

The remaining actions will no longer act on 3D objects, but on the graphic elements that make up the **User Interface (UI)**.

### TAKE\_SAMPLE

The estimated time for this action is 6 seconds. The visualization is made through a sequence of images and words. Initially, a text appears on the screen of the rover that is performing the action, telling that the rover is “*Taking Sample...*”, as shown in Figure 3.12.

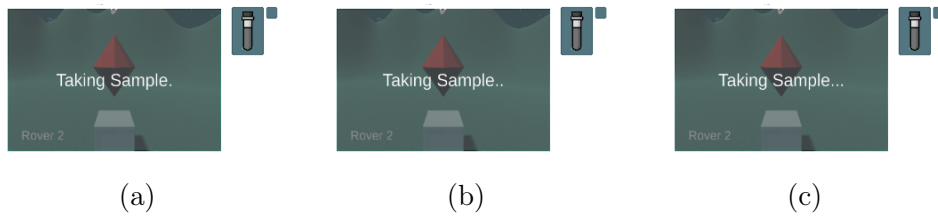


Figure 3.12: Text Sequence for the TAKE\_SAMPLE animation.

Then, the result of the action is randomly chosen in the following way:

---

```
1 outcome = Random.Range(0, 100);
2 if (outcome <= 50)
3 {
4     // do something
5     res = true;
6 }
7 else
8 {
9     // do something
10    res = false;
11 }
```

---

As it is shown, there is a 50% of success and 50% of failure. If the action is successful, an image of an intact sample is shown on the rover's screen, and the sample icon in the inventory is updated. Otherwise, the image of a broken sample appears on the rover's screen and the coroutine ends with a false result. Figure 3.13 shows these images.



Figure 3.13: TAKE\_SAMPLE images.

### DROP\_SAMPLE

The behaviour of this action is very similar to the TAKE\_SAMPLE action. The only differences are the text displayed, which for this action is “*Dropping Sample...*”, and the images displayed on the action’s success or failure that are shown in Figure 3.14:



Figure 3.14: DROP\_SAMPLE images.

### TAKE\_IMAGE

Although there seems to be no visualization for this action, it does something. The coroutine, in fact, calls another class, called TakeImage, to take a screenshot of the active rover’s camera. This class creates a folder called Screenshots (if it does not exist yet) and then creates a new thread to save the current camera’s image into a file with a unique, progressive name. Here is the snippet about the creation of the thread:

---

```

1 new System.Threading.Thread(() =>
2 {
3     // create file and write optional header with image bytes
4     var f = System.IO.File.Create(filename);
5     if (fileHeader != null)
6         f.Write(fileHeader, 0, fileHeader.Length);
7     f.Write(fileData, 0, fileData.Length);
8     f.Close();
9 }).Start();

```

---

This thread first creates a file whose name is generated using the UniqueFileName method of this class. This method generates a string that results from the combination of three elements: the destination folder, a counter that is updated when each new image is created, and the chosen format of the image. Then it adds information to the file needed to display the image: a header and the image data.

The header is used only to save the image in Portable PixMap format (PPM); it consists of at least three parts delineated by carriage returns. The first “line” is a *magic number*, [18] it represents the PPM identifier and can be *P3* or *P6*. The next line consists of the width and height of the image as ASCII numbers. The last part of the header gives the maximum value of the colour components for the pixels; this allows the

format to describe more than single byte (from 0 to 255) colour values. The format of the image data itself depends on the PPM identifier. If it is *P3* then the image is given as ASCII text, the numerical value of each pixel ranges from 0 to the maximum value given in the header. If the PPM identifier is *P6* then the image data is stored in byte format, one byte per colour component (red, green, blue - i.e. RGB). *P6* image files are obviously smaller than *P3* and much faster to read. The image data are the raw data that make up the texture. They are used to fill texture pixel memory according to its width, height, data format and mipmap count.

### CHARGE\_BATTERY

The effect of this action is to fill the battery charge, bringing its level to 100%. Its visualization is as simple as its description: the full battery level icon is enabled, while all the others (medium and low) are disabled.

### 3.3.3 Errors Detection and Handling

We considered three possible ways of failure in our game, based on prior work: [6]

- **Bug in the game:** The term *bug* refers to a failure or an error in the software that produces unexpected or incorrect results. For example, imagine that one of the rovers can not reach the destination waypoint due to an unexpected NavMesh malfunction. The rover will continue to try to reach its destination and the simulation will remain suspended waiting for the action's completion.
- **Colliding stories:** If the game has multiple stories occurring, since the resources (characters and objects) are limited, it can happen that a story tries to access a resource which has been made unavailable by the events of another story. This causes the story to stop going forward, waiting for the resource to be available.
- **System out of sync:** It can happen that the simulation checks the current state of the world and starts planning the actions to be performed, but in the meantime the world state changes. These changes can be irrelevant or significant. In the latter case, the simulation could fail and stop.

To solve the first two errors, I implemented a timeout system. As pointed out in Halldórsson, Gylfason, Gunnlaugsson and Thue's technical report, [6] the logic that estimates the maximum execution time of an action and the start of the timer should be on the game environment side, and not on the simulation side. This is because estimating the duration of actions depends on the implementation of the game and the respective timer should start exactly when the action starts and not when it is simulated. Thus, each action has a way to calculate a maximum time that is required for its completion (as described during each action's explanation). If the visualization takes more time than the estimated one, the action is interrupted, the result "false" is returned to the simulation, and the system is rolled back to the previous state.

To roll back, parameters and objects involved in every action are saved before it starts, and then are restored in case of failure. For example, when the action MOVE is requested, the system saves the initial position of the active rover and then, in case of failure, the initial position is restored.

### 3.4 Summary of the Chapter

Here is a short summary helpful to go through the chapter:

- Planning Framework. It provides a way to define actions and the current world state.
  - Relations. They define the properties valid for a world state.
  - Entities. They represent each object that can be involved in an action.
  - Predicates. They define the properties valid for a relation.
  - Actions. They are the only way to move from one state to another.
- Simulation. It gives a way to progress through states, picking up a random action among the possible ones.
- Visualization. It takes care of displaying the chosen actions.
  - The GUI. It describes how the game looks like.
  - The "Gameplay". It contains the ways a user can interact with the game.
  - Actions Description. It describes the effects of all the actions in the project.
- Errors Detection and Handling. It explain our approach on detecting the possible failures of the visualization and how to solve them.



## 4. Discussion

In Section 1.3 (Criteria for Success) we have defined two parameters through which to evaluate the visualization. Now, we want to analyze the results obtained for each of these points:

- **Correctness:** Every action has a set of pre-conditions to be satisfied so that it can be performed, and a set of post-conditions that represents its final result. Even if how the visualization is made cannot be defined with the action, we used its effects as references to get a correct display. If all the relationships that make up the post-conditions (which represent the effects of the action) are satisfied by the visualization, the latter can be considered correct.
- **Efficiency:** To obtain this property, the simulation starts a coroutine to require that the actions are displayed. In this way, the simulation will wait for a result, while the visualization, as soon as the request is received, will execute the necessary functions. Moreover, each action's duration is calculated respecting the parameters decided during the authoring process. For example, the duration of the MOVE action is estimated taking into consideration the distance to be covered and the speed of the rovers.

At present, the only control performed to detect the actions result is the estimation of a time limit for each action. Once the estimated time expires, if the action has not returned a result yet, it is automatically stopped and the system is rolled back to the previous state. For our study case, this system was good enough to always detect when the action fails, but future development could require a more robust system. For example, the estimated-time system can not be used to handle the system going out of sync. In this case, a way to fix the problem is to check the pre-conditions of actions both in the simulation system and in the game environment. [6]

Our system is able to use a planning method (the PDDL) to simulate game actions. In particular, we managed to get the information available to the player about the state of the world to be dependent on the current LOD. In particular, the information that are useless for the purpose of the visualization are cut off, making the actions rendering easier and faster. For example, considering our project, if the player is not enough close to the planet, they can not see the battery level of the rovers, so that information is not given to the visualization.

Such a system had not yet been realized, so our work offers a new approach to the problem of rendering management based on the LOD. Given the need to test this system, a scene in Unity and the visualization of the actions were realized. About this part, it was important the error detection task that we solved providing a good solution (that of estimating the execution time of the actions) valid for the test case dealt with.

There are two main limitations, which can however be compensated in the future:

- The states are defined using only unary and binary relations that limit the expressiveness of the project.
- The absence of a "smart" system to pick a goal state and find out the best sequence of actions to reach it.

## 4.1 Future Work

As already said, one of the first features to add to the project could be the creation of a goal state in the planning framework. It would make sense of the sequence of actions for each rover, which is now random.

Also, expanding the framework so that it can take into consideration relationships of a greater degree than 2 may be a future development. Such a modification would give the system the ability to generate increasingly complex states and actions, creating a more realistic simulation.

Another development of the project could concern the possibility of "fixing" the simulation in case errors occur, instead of rolling back to the previous state, thus avoiding the teleportation effect of objects. Also, as explained in the previous section, the visualization can be expanded by adding a way to avoid the system from going out of sync (the "double-check" of preconditions).

Finally, for reasons of time and complexity, a large-scale evaluation could not be carried out. Therefore, it is advisable to perform more and more detailed tests, to check that the evaluation made in this project is still valid.

## 5. Conclusion

Through this project we presented the basis for a planning framework that is capable of recreating the behaviour of a game, and that is able to choose a set of actions (which for now are randomly chosen) to be performed. Moreover, this system has the ability to cut off the useless information, depending on the current LOD, which is something that has not been investigated.

Then I focused on the definition of a complete and efficient visualization of actions. This part is strictly dependent on how the actions are meant to be displayed, so it can not be generally valid for each game scene and the respective source code needs to be rewritten according to the needs.

Finally, I figured out a way to handle the possible failures of the visualization, and I proposed future developments.



# Bibliography

- [1] Larry Bergman, Henry Fuchs, Eric Grant, and Susan Spach. Image rendering by adaptive refinement. *David C. Evans and Russell J. Athay (eds.)*, pages 29–37, 1986.
- [2] Alex J. Champandard. Planning in games: An overview and lessons learned, 2013. URL <http://aigamedev.com/open/review/planning-in-games/>.
- [3] D. Cline, H.W. Hofstetter, and J.R. Griffin. *Dictionary of Visual Science*, 4th edition. Butterworth-Heinemann, Boston, 1996.
- [4] David Dabner, Sandra Stewart, and Eric Zempel. Graphic design school: The principles and practice of graphic design, 5<sup>a</sup> edition. *John Wiley & Sons Inc*, August 2013.
- [5] Andrew Glassner. *Interactive Storytelling Techniques for 21st Century Fiction*. Imprint A K Peters/CRC Press, New York, 2004.
- [6] Gunnar Gylfason, Davíð Guðni Halldórsson, and Hafþór Gunnlaugsson. Narrative mediation in practice. Technical report, School of Computer Science, Reykjavik University, 2016.
- [7] Damian Isla. Handling complexity in the halo 2 ai. In *Proceedings of GDC-05*, 2005.
- [8] Bernard J. Jansen. The graphical user interface: An introduction. *SIGCHI Bulletin*, 30(2):22–26, 1998.
- [9] John Paul Kelly, Adi Botea, and Sven Koenig. Offline planning with hierarchical task networks in video games. In *AIIDE*, 2008.
- [10] George A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63(2):81–97, 1956.
- [11] Jeff Orkin. Constraining autonomous character behavior with human concepts. *AI Game Programming Wisdom 2*, pages 189–198, 2003.
- [12] Renato Pajarola. Efficient level-of-details for point based rendering. In *Computer Graphics and Imaging*, pages 141–146, 2003.
- [13] Philip T. Quinlan and Richard N. Wilton. Grouping by proximity or similarity? competition between the gestalt principles in vision. *Perception*, 27(4):417–430, 1998.

- [14] Mark Riedl, C. J. Saretto, and R. Michael Young. Managing interaction between users and agents in a multi-agent storytelling environment. AAMAS '03, pages 741–748, New York, NY, USA, 2003. Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems.
- [15] Ariel Shamir and Valerio Pascucci. Temporal and spatial level of details for dynamic meshes. *C. Shaw & W. Wang (eds.)*, pages 77–84, 2001.
- [16] Pushpa Suri and Meenakshi Sharma. A comparative study between the performance of relational and object oriented database in data warehousing. *International Journal of Database Management Systems (IJDMS)*, 3(2):116–127, May 2011.
- [17] Wikipedia. Behavior tree, . URL [https://en.wikipedia.org/wiki/Behavior\\_tree\\_\(artificial\\_intelligence,\\_robotics\\_and\\_control\)](https://en.wikipedia.org/wiki/Behavior_tree_(artificial_intelligence,_robotics_and_control)).
- [18] Wikipedia. Magic number (programming), . URL [https://en.wikipedia.org/wiki/Magic\\_number\\_\(programming\)](https://en.wikipedia.org/wiki/Magic_number_(programming)).
- [19] Wikipedia. Hierarchical database model, . URL [https://en.wikipedia.org/wiki/Hierarchical\\_database\\_model](https://en.wikipedia.org/wiki/Hierarchical_database_model).
- [20] Terry Winograd. From programming environments to environments for designing. *Commun. ACM*, 38(6):65–74, Jun 1995.

# Acknowledgements

I would like to thank all the people who helped me during this period.