

Solving Simulated Application Scenarios with the use of Commonsense Reasoning

Nickolas Zourmpakis

Thesis submitted in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Computer Science Department
Knossou Av., P.O. Box 2208, Heraklion, GR-71409, Greece

Thesis Advisors: Prof. *Dimitris Plexousakis*

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.

This work was partially supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science.

UNIVERSITY OF CRETE
COMPUTER SCIENCE DEPARTMENT

Solving Simulated Application Scenarios with the use of Commonsense Reasoning

Thesis submitted by
Nickolaos Zourmpakis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Nickolaos Zourmpakis

Committee approvals: _____
Dimitris Plexousakis
Professor, Thesis Supervisor

Giorgos Papagiannakis
Professor, Committee Member

Giorgos Flouris
Principal Researcher, Committee Member

Departmental approval: _____
Antonis Argyros
Professor, Director of Graduate Studies

Heraklion, April 2018

Abstract

In our society autonomous robots are becoming the new norm, being intergrated in a number of fields like household maintenance, spaceflight, delivering goods and services. This creates the necessity of constructing robotic assistants that possess highly complex functions and wide variability in order to be able to perform everyday tasks in a robust manner without sacrificing safety. As a consequence, there is a high demand to furnish these systems with a discrete high-level reasoning combined with an adequate and continuous low-level reasoning.

The purpose of our research is to create the basis of a system for training automations to handle and perform simple actions that can be later combined to create the necessary patterns for handling everyday tasks like a normal person would. We introduce a hybrid approach, unifying high-level planning using a logic-based formalism for representing actions and effects, with low-level geometric feasibility checks through the use of an advanced simulation framework with an integrated physics engine. It is an experimental application created for producing a number of viable solutions to a given plain problem, while imitating conditions found in the natural world. Its targeted user group are researchers who wish to test the physical feasibility of scenarios, expressed in a logical language like the event calculus, inside a real-time environment.

The application allows the user to create physically sound objects according to his preferences inside a designated area through the use of an interactive UI. He can cluster the objects into different formations, that he can later use to enhance the Knowledge Base of the system (available patterns). Our approach focuses on testing three alternative types of scenarios, each created to explore the different properties of an object and its interactions with other objects. The user can view in real time all the available outcomes, what are the steps that each plan consists of, and even make small alterations to get a favorable result, without the need to re-execute any high-level planning.

Περίληψη

Στην κοινωνία μας τα αυτόνομα ρομπότ γίνονται ο νέος κανόνας, ενσωματώνονται σε πολλούς τομείς όπως η συντήρηση των νοικοκυριών, η διαστημική πτήση, η παράδοση αγαθών και υπηρεσιών. Αυτό δημιουργεί την ανάγκη κατασκευής ρομποτικών βοηθών που διαθέτουν μεγάλης πολυπλοκότητας δυνατότητες και εύρος ώστε να είναι σε θέση να εκτελούν καθημερινά καθήκοντα με ισχυρό τρόπο χωρίς να θυσιάζουν την ασφάλεια. Ως εκ τούτου, υπάρχει μεγάλη ζήτηση να "οπλιστούν" τα συστήματα αυτά με διακριτή λογική υψηλού επιπέδου σε συνδυασμό με επαρκή και συνεχή λογική χαμηλού επιπέδου.

Σκοπός της έρευνάς μας είναι να δημιουργήσουμε τη εναρκτήρια βάση ενός συστήματος για την εκπαίδευση ρομποτικών βοηθών για να χειρίζονται και να εκτελούν απλές ενέργειες που μπορούν αργότερα να συνδυαστούν για να δημιουργήσουν τα απαραίτητα πρότυπα για το χειρισμό κάθε τύπου εργασίας όπως ένα κανονικό άτομο. Εισάγουμε μια υβριδική προσέγγιση, ενοποιώντας τον προγραμματισμό υψηλού επιπέδου χρησιμοποιώντας έναν λογικό φορμαλισμό για την εκπροσώπηση ενεργειών και αποτελεσμάτων, σε συνδυασμό με χαμηλού επιπέδου γεωμετρικούς ελέγχους επικτότητας μέσω της χρήσης ενός προηγμένου πλαισίου προσομοίωσης με μια ενσωματωμένη μηχανή φυσικής. Πρόκειται για μια πειραματική εφαρμογή που δημιουργήθηκε για την παραγωγή πολλών βιώσιμων λύσεων σε ένα συγκεκριμένο πρόβλημα, ενώ παράλληλα μιμείται τις συνθήκες που απαντώνται στον φυσικό κόσμο. Η ομάδα χρηστών στην οποία απευθύνεται είναι ερευνητές που επιθυμούν να δοκιμάσουν τη φυσική δεινότητα των σεναρίων, που εκφράζονται σε μια λογική γλώσσα όπως ο Λογισμός Συμβάντων, μέσα σε πραγματικό περιβάλλον.

Η εφαρμογή επιτρέπει στον χρήστη να δημιουργεί πραγματικά φυσικά αντικείμενα σύμφωνα με τις προτιμήσεις του μέσα σε μια καθορισμένη περιοχή μέσω της χρήσης ενός διαδραστικού περιβάλλοντος. Μπορεί να συσσωρεύσει τα αντικείμενα σε διαφορετικούς σχηματισμούς, που αργότερα μπορεί να χρησιμοποιήσει για να ενισχύσει τη Γνωσιακή Βάση του συστήματος (διαθέσιμα μοτίβα). Η προσέγγισή μας επικεντρώνεται στη δοκιμή τριών εναλλακτικών τύπων σεναρίων, κάθε ένα από τα οποία δημιουργήθηκε για να διερευνήσει τις διαφορετικές ιδιότητες ενός αντικειμένου και τις αλληλεπιδράσεις του με άλλα αντικείμενα. Ο χρήστης μπορεί να δει σε πραγματικό χρόνο όλα τα διαθέσιμα αποτελέσματα, ποια είναι τα βήματα που κάθε σχέδιο αποτελείται από, και ακόμη και να κάνει μικρές αλλαγές για να πάρει ένα ευνοϊκό αποτέλεσμα, χωρίς την ανάγκη εκ νέου εκτέλεσης οποιουδήποτε υψηλού επιπέδου σχεδιασμού.

Acknowledgements

The realization of this thesis would not have been possible without the continuous support of so many individuals who regrettably it is impossible to mention here to their full extent.

First and foremost, I would like to thank my supervisors, Professor of the Computer Science Department Dimitris Plexousakis, Postdoctoral Researcher Theodore Patkos and Affiliated Researcher Giorgos Flouris of the Institute of Computer Science. Their patience, support and guidance on both educational and personal level is what helped me to push on and aim for a higher academic excellence. In times of distress their passionate participation, rational thinking and invaluable input were key components in successfully conducting this master thesis.

I would also like to acknowledge the Institute of Computer Science (ICS), the Foundation of Research and Technology Hellas (Forth) and especially the academic members and faculty of the Information Systems Laboratory (ISL) who provided me with a pleasant atmosphere to work on this project and for all the informal conversations that brought me new insights, hints and feedback.

Of course, this acknowledgement would be incomplete without thanking my parents and family, who supported me throughout my entire study and encouraged my efforts. They, along with my closest friends, were the strong foundations upon which I built my character and determination and for that I am forever grateful.

Contents

1	Introduction	3
1.1	Motivation	3
1.2	Problem Statement	4
1.3	Contribution	5
2	Background	7
2.1	Answer Set Programming (ASP)	7
2.2	Event Calculus	7
2.3	Unity	8
2.4	Python	9
3	Related Work	11
3.1	Object Manipulation + Logic Programming	11
3.2	Commonsense Simulation	12
3.3	Answer Set Programming with Agents	13
3.4	Physics-Based Planning Systems	14
3.5	Combining High-Level Task Planning and Low Level Checks	15
4	Methodology	17
4.1	The Problem	17
4.2	Our Solution	17
4.3	The Components	19
4.3.1	Event Calculus - Clingo	19
4.4	Unity Engine	21
4.5	Python	22
5	Architecture and Implementation	25
5.1	Architecture	25
5.2	Implementation	30
5.2.1	The UI/Simulator	30
5.2.2	The Solver/Reasoner	33
5.2.3	The Intermediate	35

6	Use Cases	37
6.1	The Tower of Babel (a variation of the blocks world problem) . . .	37
6.2	Test the Balance	40
6.3	Natural Properties	45
7	Conclusions and Future Work	47
7.1	Future Work	47
7.2	Discussion on technical challenges and lessons learnt	48
7.2.1	The Versions	48
7.2.2	The Learning Curve and Rising Issues	49

List of Figures

2.1	Example of how Event Calculus functions	8
2.2	The Unity physics engine	9
4.1	The basic structure of our framework	18
4.2	An example of TinyDB	23
5.1	The General Architecture	26
5.2	How to Create a new Fluent	27
5.3	The Sampled Data	28
5.4	A Clingo Outcome	29
5.5	The Front-End Architecture	31
5.6	The Object Menu	32
5.7	Saving a Scene	32
5.8	The Timeline of Solutions	33
5.9	The Back-End Architecture	34
5.10	The Intermediate Architecture	35
6.1	A random initial state for creating a tower	38
6.2	Timelines for Use Case 1	40
6.3	A random initial state for balancing objects	41
6.4	New Fluents	42
6.5	Timelines for Use Case 2	44
6.6	The set initial state for Use Case 3	45
6.7	Timelines for Use Case 3	46

List of Tables

2.1 Simple Event Calculus Predicates [1]	8
--	---

Chapter 1

Introduction

1.1 Motivation

Automation is a modern term that has its roots set more than 200 years ago when it was patented by Edmund Lee in 1745. It can be defined as the technology by which a procedure or task is performed without human assistance [2]. In a way its purpose is to do the "impossible" and that is by improving the quality attributes of a process. Its first use was for industrial purposes, a trend that still holds strong today, but steadily it expanded to other aspects of our society and especially to almost all things that relate to technology. When we add those two things together for most of us a specific term comes into mind and that is "Robotics" or using a less popular synonym: "Automatons".

Robots are becoming the new norm. They are used for various applications and lately even in fields traditionally occupied by humans. However, in order for a robot to be successful its most eminently necessary capability is understanding the physical interactions with the environment and how every decision has an impact. In Robotics manipulation planning has become an important focus area of research aiming for the automatic production of motion sequences when manipulating objects on the physical plain in order to achieve a desired goal configuration [3]. Basically it tries to answer the question : "How can a robot decide what actions to perform in order to achieve goal arrangements of physical objects ?".

To a human the principle of "motion planning" is relatively simple and an indifferent part of our everyday lives. But for a computer even our day-to-day routines are extremely difficult to duplicate. Most modern state-of-the-art motion planning systems handle problems only on a specific domain addressing a specific sort of manipulation [4]. Even more elaborate approaches choose to focus on the integration between task and motion planning keeping the contribution of the geometric reasoner to a series of external predicates that most of the times focus on simple concepts like collisions. The reason for that is this false still widespread view that motion planning is all about checking and avoiding collisions, where in fact it is so much more than that. Gravity, friction and mass are just a few physical properties

necessary to establish precise relations among a variety of objects. A robot has to know these geometrical constraints in order to reach feasible kinematic solutions.

Our approach takes a step towards a more flexible and expandable framework. It is designed to handle more realistic physical concepts by taking advantage of the capabilities of the Unity game engine as a means to intergrate low-level geometric reasoning to our high-level motion planning. It allows the user to test simple scenarios by filtering the multiple plans of the Clingo reasoning process through the simulations of real time environments created through the use of the Unity game engine. The app may be at a basic level, but the results prove there is a very exciting perspective in future expansion.

1.2 Problem Statement

Developing an application aimed for improving robotic assistance is not an easy task. Object manipulation using only logical programming is difficult to develop correctly when not paired with an appropriate tool to handle obstacles that may occur inside a physical environment. Even then, there is an increasing number of available forms of integration most of which are difficult to expand, alter or even understand to begin with. We want a simple system with powerful components, that each focuses on what they do best, balancing the work, thus increasing the productivity of the developer.

Lets say for instance, that someone wishes to develop software that allows a robotic assistant to help with "cleaning the kitchen". The first obstacle is how to imprint all this existing vast commonsense knowledge beforehand, in order to integrate it into the decision-making mechanism of a robot. A favorable recent tactic is the use of Logic Languages that offer modeling and reasoning capabilities for handling this type of abstract knowledge, presenting significant solutions for such environments. As such, the researcher could write the code with more ease by using a logic programming language like Prolog or in our case ASP. A more serious problem would rise after that with handling the objects. A reasonable action would be to put one item on top of another, but to do that we would need to consider their shapes in order to avoid stability issues. Similar issues could stem if we don't further contemplate on other physical properties like their weight or the maximum pressure they can handle before breaking, all of which would lead to flopping the goal state and even potentially to the destruction of our objects. Imagine loading this code into a robot and letting it loose inside a real kitchen. The number of failures, recoding and retrying would be huge not to mention the damages. Wouldn't it be better to take all the available plans and filter them through with simulation-based temporal projections to root out the infeasible ones? There would be no need to change the original logic program, as well as create the appropriate reasoning for such complex notions like gravity or friction.

Although there are other similar systems out there like [5], [6] and [4], most of them depend on lesser tools for handling their simulations. In many cases, they

focus only on a specific type of problem "overfitting" their solutions, use inflexible ways to add input or require too much information that sometimes it's unnecessary and has zero effect on the outcome. *We wanted to make a system easy to use as possible, flexible in terms of expansion, able to handle multiple types of scenarios and pleasing to the eye if possible.*

1.3 Contribution

As we have already mentioned in the previous section, similar approaches to our own do exist, and in some cases they provide more features than we do. However, to the best of our knowledge, this is the first effort to couple in a unified framework i) a very expressive logical formalism, namely the Event Calculus, that is able to express a multitude of commonsense phenomena, ii) a powerful, non-monotonic reasoner with formal semantics, namely the clingo Answer Set Programming reasoner, that outperforms SAT- or Prolog-based reasoners, and iii) an industrial state-of-the-art game engine, namely Unity, able to perform hundreds of times better than small virtualization apps since its main focus is to deliver an experience as close to reality as possible. After all, it was build with this idea in mind.

The system provides a simple environment with an accompanying menu where the user can create a number of objects for testing according to his own specifications. He can add his own clingo files, interactively teach the system of the actions it needs to know, and test the physical validity of the resulting plans. We try to keep the user away from any additional code concerning the simulations, which is a great advantage for those who don't have hands-on experience with that. We also organized the architecture in such a way that it is flexible, easy to adapt in future changes, and has as much exposed parts as possible, allowing for tinkering, for those who wish to take things a step further.

To summarize, our main contributions can be distilled into the following points:

- we developed a unified framework able to reason with commonsense knowledge using logical formalisms at a higher level, while considering the low-level physical properties of objects.
- we implemented a fully operational system, offering a simple GUI for the user to create objects with different properties
- we designed a decentralized modular architecture, open to future expansions
- we offer a simulation methodology for handling extremely complex physics and for adding new concepts as add-ons from its online database (for future expansion)
- we designed and implemented a set of use cases of diverse complexity that, although simple, capture various features that are typically met in a variety of domains and can easily scale to more complex ones.

- relying on recent comparative studies on combining high and low-level reasoning [4], we developed an optimized approach for enhancing timeline simulations, based on filtering failed patterns found in candidate plans, speeding the process of finding the feasible ones.
- finally, we offer a system that is completely portable between Pcs (not supporting other OSs at the moment)

Chapter 2

Background

2.1 Answer Set Programming (ASP)

Answer Set Programming (ASP) is a powerful formalism for Knowledge Representation and Reasoning [7]. It is based under the stable model semantics (DLP) and its increasing success is a result of the high expressive power of its language: in an ASP program every finite structure can be precisely expressed (properties etc) as a first order function-free structure with a very simple and elegant encoding even with a large variety of problems. Search in ASP is basically reduced to computing stable models and employing answer set solver for performing the search. Using answer set solvers is a positive addition since most of them base their computational process on an algorithm that avoids infinite loops and, in principle, always terminates (DPLL). In our implementation we used Clingo, a reasoner that combines the answer set solver Clasp (works on variable-free programs) and the Gringo grounder (a tool for transforming a program with first-order variables to a variable-free program) into a monolithic system that has increased control over the whole solving process.

2.2 Event Calculus

The event calculus is a logic-based formalism for representing actions and their effects. It is based on a first-order predicate calculus and can be applied to represent a large variety of phenomena, such as actions with indirect effects or nondeterministic effects, simultaneous actions, compound actions, and the occurrence of continuous change [1]. It is a mechanism that combines the knowledge of "what happens when" and "what actions do" in order to infer "what's true when". It's like a narrating a series of events and describing the effects of actions. A simple example can be seen on Figure 1 below:

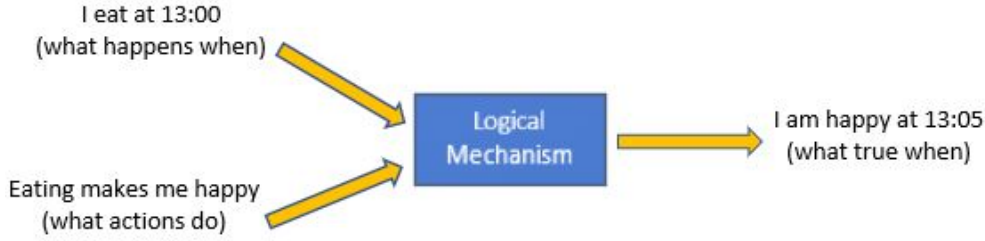


Figure 2.1: Example of how Event Calculus functions

The fluents in EC are reified, meaning that they are no longer formalized by means of predicates but by means of functions. Basically, fluents are quantified in order to be used as arguments to predicates. Those predicates help us describe the initial state of a problem, the effects of actions and what fluents hold at what times. Table 1 is a first introduction of the linguistic elements of the EC, in a more slim version of the Original Event Calculus that eliminates the notion of incompatible fluents and replaces time periods and event occurrences with timepoints and event types [8].

Formula	Meaning
$\text{Initiates}(\alpha, \beta, \tau)$	Fluent β starts to hold after action α at time τ
$\text{Terminates}(\alpha, \beta, \tau)$	Fluent β ceases to hold after action α at time τ
$\text{Initially}(\beta)$	Fluent β holds from time 0
$\tau_1 < \tau_2$	Time point τ_1 is before time point τ_2
$\text{Happens}(\alpha, \tau)$	Action α occurs at time τ
$\text{HoldsAt}(\beta, \tau)$	Fluent β holds at time τ
$\text{Clipped}(\tau_1, \beta, \tau_2)$	Fluent β is terminated between times τ_1 and τ_2

Table 2.1: Simple Event Calculus Predicates [1]

2.3 Unity

The Unity (game engine) is a cross-platform game engine that was developed by Unity Technologies in 2005 and up until December 2017, six major versions have been released. Its primary use is for the development of two dimensional and three-dimensional games and simulations for a number of devices such as pc and mobile phones with that list getting up to 27 different platforms. Its supported scripting language is C# while most of its operations are based on a drag-and-drop functionality from a wide collection of items that can be enhanced from an online store where users can sell their own creations for free or for a price.

Within 3D games, Unity provides support for a number of effects such as bump mapping (simulating bumps and wrinkles on the surface of an object), reflection

mapping, parallax mapping (greater surface realism), dynamic shadows, fragments, tessellation and most importantly a powerful physics engine that can provide simulation for accelerations, collisions, gravity and other forces. When these advantages are combined with the support of a huge online community it is quite obvious why thirty-four percent of the top games are made with Unity and also why it is at the forefront of the growing VR market, with fifty-three percent of the Oculus Rift games being made with Unity.



Figure 2.2: The Unity physics engine

2.4 Python

Python is an interpreted high-level programming language that emphasizes in code readability, with a syntax that allows the expression of concepts in fewer lines of code. In comparison to most object-oriented programming languages, rather than having all of its functionality build into its core, it was designed to be highly extensible, making it really easy to be used as a means to add programmable interfaces to existing applications [9]. Also, due to its extensive mathematics library (in addition to the third-party library NumPy), Python is frequently used for scientific scripting for problems such as numerical data manipulation and processing. In the last decade, Python has also been implemented to artificial intelligence projects and more specifically for the use of natural language processing, since it possesses a high number of text processing tools.

Chapter 3

Related Work

Tackling the problem of object manipulation using logical programming on a simulation-based environment is a relatively small underdeveloped area of research, that is gaining a fast momentum in the last few years thanks to the development of new sophisticated programming tools that are becoming easier to access from less advanced users. Considering our approach, in the following sections we analyze a few similar methods/applications/papers and how they measure up against our own system.

3.1 Object Manipulation + Logic Programming

Autonomous robots are one of most "hot areas" of research being intergrated in a number of fields such as spaceflight, household maintenance, delivering goods and services. In 2011 Larz Kunze, Hihai Emanuel Dolha and Michael Beetz [3] proposed an integration of logic programming and simulation-based temporal projections in order to provide robots with the ability to mentally simulate the outcomes of different actions before committing them, thus being able to cut down the number of unsuccessful results.

Their approach is very similar to our own since it couples the reasoning capabilities of the logic programming language PROLOG with a simulation sensor-based engine called Gazebo. There are three performed experiments measuring the results of performing tasks that require manipulating everyday objects. The overall process of their system is composed of five basic steps: a knowledge base is translated into a physical simulation, the simulation is then executed, the resulting objects and structures are logged in, the logged simulations are translated into timelines, and finally those timelines are being interpreted by PROLOG.

Taking into consideration the complexity of the manipulation actions they focus on, their process is very solid, taking full advantage of the Web Ontology Language (OWL) in order to handle the detailed information describing their objects, and seperating their results into first-order representations (timelines) that can be easily turned into predicates from the Event Calculus in order to be reasoned with and

evaluated. The use of the physics engine is not just for considering actions on an abstract level (in contrast to most approaches at that point in time) but they focus on the physical details and the phenomena that occur during the simulations. A very strong aspect of their work is the fact that the objects they handle can be deformed during the execution, thus increasing the difficulty of procuring a viable solution.

On the other hand, some of their choices regarding tools and processes, sets a number of limitations to the project. The Gazebo engine is truly one of a kind in terms of robot simulations, utilizing four different physics engines and is capable of handling hundreds of simple robots or a few very complex ones at the same time. However, this processing power needs the appropriate hardware response and the necessary budget that comes with it. Even then, Gazebo cannot compare with the available game engines in terms of learning curve, available tools and flexibility in handling anything other than robots. Our second objection, has to do with the fact that all the information stored about the objects, has to be translated and assessed by the logic programming language in order to reach a solution. All this transformation of data back and forth is unnecessary and can further clog the available hardware. Our approach is much more balanced, distributing the load between the different parts, taking into account where they excel at.

3.2 Commonsense Simulation

IsisWorld is an open source kitchen simulator written in Python with a highly-customizable environment, developed for observing the behavior of commonsense reasoning systems. This application is classified as a metareasoner and it is a very important operation for solving problems with a limited amount of information and computational resources [10]. This type of system has three basic components: a set of problem domains, a group of reasoners for solving the problems and a metareasoner or ensemble of metareasoners. The goal is to build an AI that possesses a human-level intelligence, can simulate human linguistic and conceptual competencies and all this in order to face "commonsense" reasoning problems.

The simulator is build with expandability in mind, with easy to write scenario templates, procedurally generated environment descriptions according to commonsense, a rigid-body physics engine based on the Open Dynamics Engine and continuous update of available objects, actions and events with commonsense behaviors. The project supports all three major computer platforms (Windows, Mac, Linux) with a comprehensive tutorial on how to install and run. Using the simulator is accomplished with the help of the Panda3D game engine with a very simplistic UI environment. The user can control the simulation with key commands, entering text or even through a programmed agent. There are a number of actions that can be performed, many of them even at the same time with compositional constraints encoded by a planner.

All in all, this implementation is very well-refined aiming to solve a "space of

tasks" and not just a particular one, a common trap for most researchers that "overfit" their solutions to meet the task's requirements. This tendency is a prohibiting factor for many AIs that later on cannot generalize to other tasks, even though the original goal was to do that very thing. IsisWorld is trying to develop a multi-step approach in order to generate multiple tasks that encourage the AI to be flexible in different levels of problems like motor controls, conflicting goals, lack of information, etc. However, the platform falls a little short on a couple of things, such as the weak physics engine and the low level graphics of the small Panda3D game engine (performance-wise), elements that all modern game engines integrate into one. Also, considering the current level of the hardware industry, modern pc can handle a lot of processing, being able to simulate heavily detailed environments as close to reality (visually and physically) as possible, removing the conundrum of performance over visual representation.

3.3 Answer Set Programming with Agents

HumanSim is an agent platform combining the BDI-paradigm with Answer Set Programming in order to simulate entities in virtual (three-dimensional) environments [11]. Its main use (at the current moment) is for enabling realistic, real-time medical training, creating dynamic medical scenarios and environments that demand quick medical decision making.

In HumanSim, the human models are simulated with the help of BDI-agents, meaning they possess Beliefs: knowledge about the environment, Desires: long-term goals, and Intentions: actions that must be performed to reach the current goal. Any interaction with the environment is performed in an autonomous and realistic way, and only if the agent understands what it is "seeing". In order to acquire knowledge, the way to annotate it into the available environment and objects is through the use of semantic information that is declared with ASP. Furthermore, they use ASP for adding commonsense reasoning to the agents, thus imitating a near-close foresighted human acting. Similarly to our approach, this platform takes advantage the capabilities of ASP in order to specify additional features to the problem domain, like indirect effects and background knowledge. Accounting for the possible effects of an action is accomplished with the help of the Discrete Event Calculus (DEC), empowering an agent to project his current knowledge into the future and plan a course of actions to achieve his goals. To combine all the above, HumanSim takes a layered approach using a three layer architecture, with each layer responsible for dealing with specific aspects of the simulation.

In conclusion, HumanSim is an excellent tool, designed to be as collaborative and dynamic as possible, with many of its components allowing the simultaneous use of the same scene from multiple users. Its use of ASP paired with the layered architecture is very similar to our approach, with a few small differences, like the fact that all entities are annotated using ASP rules. In contrast, our platform

"layers" information, choosing to represent complex knowledge to the format that is most easily possible to. A second objection is the simulation engine of HumanSim, which is a combination of a web editor called COMPASS and a virtual environment server named FIVES. All those components are still at a very early stage, and although they are very promising, there is very little data (manuals, forums, etc) about how they work and more importantly what physics engine (if-any) they integrate.

3.4 Physics-Based Planning Systems

[12] is a system that focuses on a physics-based planning beyond just detecting collisions and avoiding them, but purposefully manipulating non-actuated bodies with varying degrees of controllability. It implements an efficient physics-based algorithm (testing two variations) and reduces the motion action space by taking advantage of the agent's high-level behaviors. They have also developed a model that infuses such behaviors into a randomized motion planner. Finally, they created a state and transition model that approximates interbody-dynamics by employing rigid body simulations (by NVIDIA PhysX).

A vast body of work such as [13], [14], [15] uses variations of the Rapidly-Exploring Random Trees (RRT) algorithm in order to develop a search tree in the configuration space, hoping that one of the tree's leafs will reach the goal state. RRT employs sampling-techniques and its search tree can rapidly cover the desired configuration space. It has been shown that RRT is most favorable in navigation planning problems that involve kinematic and dynamic (kinodynamic) constraints. However, algorithms like RRT are typically employed for collision-free robot navigation problems, and are unsuitable for cases where objects have to be manipulated in order to reach the desirable goal-state.

A number of approaches like [16] and [17] have integrated behavioral models into the dynamic-based motion planning. The goal was to solve problems in the computer graphics domain, thus they take control of all the agents and all other bodies within the domain. For that same reason they were considered to be inapplicable in solving physics-based planning problems, where manipulating passive bodies was a requirement. A second objection was to the fact that the "behaviors" used in graphics were nothing more than pre-recorded computer animation sequences and there was no dynamic decision making in the process. These kind of restrictions have been eradicated from the modern graphic simulations, with the use of the powerful game engines like the one we employ.

In terms of physical planning many of the approaches above take better advantage of the environment and the objects in it, in more detail than our system, however the controls in these platforms are determined through the exploration of a search tree of states, an approach which is much more rigid than the logic-based formalism (EC) we use in our implementation.

3.5 Combining High-Level Task Planning and Low Level Checks

This subsection is very interesting since it contains a number of implementations that follow the same general strategy of combining high-level reasoning with low-level kinematic/geometric feasibility checks, presenting different types of integration in regards to the level that's being done.

In studies such as [18], [19], [20], [21] the integration happens at the search level using a search algorithm in order to incrementally build the task plan, while a motion planner is making kinematic feasibility checks. The zestful thing with each of these projects is how the task planner assists the process of search during motion planning, by employing different methods in utilizing the information in order to narrow down the available configuration space and speed up the search. The only drawback here is the fact that these approaches do not consider a general interface between task and motion planning, but present specialized combinations that are difficult to alter if needed.

On the other end, approaches such as the ones presented in [22], [23], [5], [6] choose to perform the integration at the representation level, using a general interface and external predicates/functions computed with the help of an outside mechanism such a C++ or Java program (very similar to ours). The difference here is that basically the motion planner guides the task planner at the representation level using the predicates. It is very similar to our case, since some of them even employ ASP and reasoners like CCalc or Clasp to achieve this. One very notable study is [4] which goes a step further from the other ones by considering a more flexible framework with modular integration via an interface in order to better embed continuous low-level reasoning with high-level reasoning at various levels. The reason for its high importance is the fact that our framework is a realization of one of the two methods presented for post-checking where all low-level (geometric) feasibility checks are performed after the planning has been computed. ASP is also present here for expressing formalisms, but actions and change are described with HEX programs. In general, the key component that separates our work from any of these papers is the use of the Unity engine that provides a much more spherical and complete tool for performing geometric feasibility checks, without the need to burden the reasoner with complex predicates and functions.

Chapter 4

Methodology

4.1 The Problem

Before going into any details about the components that constitute the practical part of our dissertation, we need to better address "the conundrum" we set out to solve. Everyday, we perform sets of tasks, that may seem mundane, but they involve a number of elaborate calculations on our behalf, that most of the time are unintentional and effortless. So the main question here is how a machine can "strategize" the same way, or even better, in order to solve daily chores that implicate arrangements of physical objects. Manipulating objects on the physical plain is the current focus area in robotics and the next necessary step in achieving goals the way humans do. To summarize, our problem is to create a system that uses the advantages of a high-level reasoner and filters its results through a dynamic geometric simulator to produce real feasible plans. We should break the problem down into the following bullet points for consideration :

- Many task are complex
- Motion planning is not only about collisions
- A single tool cannot excel at everything

4.2 Our Solution

Our framework was decided after carefully considering the related work that has been done on the field so far and where they lack. Since this field is still in development, most existing papers referenced each other pointing out their weaknesses, thus making it easier for us to focus and improve in some of those aspects, going a step forward with our work. Most solutions like [18, 19] take the path of improving either the task or motion planner, meliorate the integration between them [6, 4], introduce more complex algorithms for searching available paths to the goal state [13, 14, 15], or finally by adding more knowledge to the environment and the objects

it contains [11], as well as adding ways to capitalize on that (commonsense reasoning). The common denominator between all of them was the fact that physics and thus geometry was always backstage, usually simulated by predicates and almost solely focused on collision detection while moving inside a pre-determined space. Even in the few exceptions where physics got a little more "spotlight", the newly physical notions introduced, were severely limited due to their complexity or the lack of the tools that were used in order to integrate them. Symbolic or numerical methods are not enough to handle the problem at hand and further straining them will only have the opposite effect.

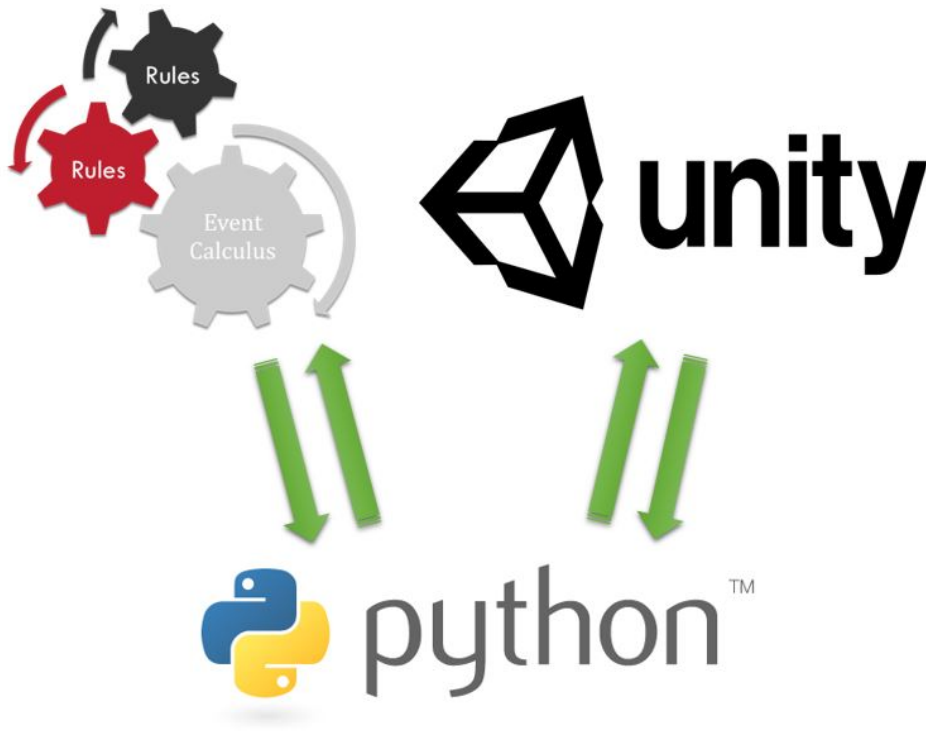


Figure 4.1: The basic structure of our framework

The general focus of our solution is still (like the ones above) on the spectrum of "motion planning", and how a computer/robot can use it in order to handle problems on the physical domain by improving the motion and task planning with commonsense reasoning. However, instead of adapting integrations that set task and motion planners as the focalized elements of reason, we decide to also increase the contribution of the geometric reasoner into a component that can handle more than just collisions. To achieve something like that, we first use a method of "divide and conquer" by splitting the computations between each component of framework considering at what area they excel the most. The reasoner combined with the Event Calculus handles the part of performing the planning and com-

monsense reasoning in terms of actions, abstractions for goal states, constraints, etc. The geometric reasoning is performed by using the physical capabilities of the Unity engine that is also responsible for the available UI. Concepts like friction, gravity, mass or impacts are all handled by Unity internally without the need to get into any details of how these notions work. Bridging those two parts together and performing all the data manipulation and the rest of the mathematical computations needed, is the responsibility of the programming language Python. If there is something that can't be expressed with physics or commonsense reasoning, we can represent it as an easy to understand programming function with it.

4.3 The Components

In order for someone to better understand the framework we have created, we will dedicate this section in presenting its key components from a theoretical perspective, so as to set the groundwork for the following chapter where we will analyze the application in terms of functionality. Details as to what each component is, have already been presented in Chapter Two, along with any correlating terminology that we will employ here.

4.3.1 Event Calculus - Clingo

Using the Event Calculus (EC) is a vital part of our framework and it is possible through the use of the Clingo reasoner. We use a number of the available predicates from the Simple Event Calculus in order to map our available objects and the interactions that occur between them, all things that if computed with a traditional programming language, it would take hundreds of lines of code to even cover the most basic scenarios not to mention the modeling of commonsense rules for each individual object, such as the law of inertia, ramifications of actions etc. Following, we present a number of ways we use EC to our benefit:

Fluents are one of the most important parts of our system, since the user can employ them to model an "instance" of sort. They are the data transformed and transmitted between the Unity engine and the Clingo reasoner. For clarifying things, an example, would be putting a cube on top of another. The objects here are not so important, rather we focus on what they represent. By normal "programming standards" we would need to make correlations between the objects using spatial coordinates and conditions that they would need to uphold, but with EC we basically need only to create a fluent with a distinguishable name like *onTopOf* involving two separate objects. After that, for a fluent to apply between objects, we use the predicates available by EC, details as to how, we discuss on the appropriate subsection below. Fluents are very significant to our application since we use them to describe the initial state of things inside a virtual environment, where we need to solve a set of problems.

`% Put an object on top of another`


```
fluent(onTopOf(X,Y)):- object(X), object(Y), X !=Y.
```

Declaring that a fluent is an "instance" is not very appropriate, since they basically describe all those parameters of the environment that may change their value from one timepoint to another. A more suitable term for triggering those changes is **Actions**. An action may initiate or terminate one or more fluents as depicted by *initiates* and *terminates* respectively. Once an action is executed, from that moment on the truth state of the involved fluents will either start to hold (be true) or will cease to apply (be false). Surely, a beginning and an end is not enough to describe an action sufficiently, since the execution of some action may be dependent on the truth values of certain fluents, acting as preconditions for a successful execution. This leads us to the **Constraints**, a set of causality rules and constraints, which collectively axiomatize our commonsense formalization of the domain of interest. For example, putting an object on top of another presupposes that no other object is already there on top. The execution may seem too simplistic, but it is actually this level of abstraction that makes EC so beneficiary to employ.

```
% Definition of axioms for putting an object on top of another object
```

```
initiates(moveObjTo(X,Y),onTopOf(X,Y),t) :-  
object(X), object(Y).
```

```
terminates(moveObjTo(X,Y),onTopOf(X,Z),t) :-  
object(X), object(Y), object(Z), Y!=Z,  
holdsAt(onTopOf(X,Z),t).
```

```
above(X,Y,t) :- holdsAt(onTopOf(X,Y),t).  
above(X,Y,t) :- holdsAt(onTopOf(X,Z),t), above(Z,Y,t).
```

As mentioned above, fluents are the building blocks for setting the initial state of a scene inside a virtual environment. The same inception can be applied for the goal state or the solution to the problem we investigate. However, using the aforementioned ambiguity of EC we can create simple predicates that describe the entire spectrum of our solution, using even existing fluents, without needing any more detail than that. A suitable example is the moving all the objects from the floor of a room to a table. Instead, of declaring how every single object must be on top of the table, we can just declare that our goal is that there is no active fluent *onTopOf* between an object and the floor.

```
% Predicates for moving all objects away from the floor  
notClear(floor, t) :- holdsAt(onTopOf(Obj,floor),t), movableObj(Obj).  
clear(floor, t) :- not notClear(floor, t).
```

Combining all the above, we can deduct that planning in the event calculus is an abductive task. We have a given domain, a conjunction of formulae describing

the initial situation and one representing the goals. All that leads to a number of plans-solutions that are a consistent conjunction of fluents and temporal ordering formulae. To be more specific, what we get as a result are actions and the moment in time that they occur. Thanks to that, we are able to separate the solutions into timelines, meaning that if two solutions follow the same sequence of actions up to a point in time, and one of them fails inside that sequence, then it automatically also sets the second solution as nonviable.

All in all, the EC is a very handy tool in representing objects and actions inside an environment, however, as with all things, it has limitations (deriving mostly from computational restrictions) especially in defining complex concepts like gravity or mass, variables that are vital in the physical world. Another problematic aspect is performing mathematical calculations, such as estimating the new position of an *object a* when set above an *object b*. Abstractions in this kind of situations are the opposite of helpful. That is why we don't use EC to model everything, but take a selective approach outsourcing the "difficult" parts to the other components.

4.4 Unity Engine

The Unity game engine is at the forefront of our application, since it provides the necessary UI interface in order for the user to interact with the rest of the components and observe the results of the test cases with regards to his/her modifications. However, Unity is much more than that since it comes with all the set of tools and add-ons that make our framework as close to the physical world as possible.

Developing a **UI** with Unity is one of most attractive features (in terms of simplicity) of the application. Being a game engine, Unity provides ready UI components such as buttons, sliders, scrollbars, transitions etc, allowing the developer to drag and drop everything to the exact location, and choose how they react from the available options. The components can be set to scale depending on the screen resolution, an addition most welcome for developers who target mobile devices. Creating a minimalistic menu with a few visual enhancements here and there was the best option, considering the time table, since any fancy implementations would require getting more familiar with the inside code behind the UI components or paying for already available ones at the online store, both options unnecessary to our goal, but still welcome for future improvements.

The **physics engine** integrated into Unity is one of best that exist, simulating a number of potential forces ranging from gravity, momentum of a vehicle, to even powerful ones like explosions. To our benefit, in our simulations we are employing gravity, mass, collisions and friction all of which are already available in the core version of unity without needing to download external libraries from the existing unity online store. Every object pre-existing or created from the user is impacted by those forces and they cease to exist only when the user desires to make a modification, meaning that pilling up objects, for example, is much more doable if gravity is not interfering. Once the user has finished making his alterations to the

environment (through the UI and only), everything is back to the normal physical parameters. One important detail here is that our approach of the simulation is that of a "sandbox". This means that beside the player-camera everything else is part of the simulation and subjected to it, so accidentally knocking things around is not possible, and this extra freedom can help the user in making more precise alterations. However, during the simulation the user becomes just an observer and all actions occurring are out of his control and completely agnostic as to the characteristics of the perpetrator, since our objective is how to solve a particular scenario and not to investigate who is trying to solve it.

Although it may seem that Unity is providing our application with just two elements for use, the overall contribution is much more significant than we can portray. Our simulation environment is a rather bleak room, but we could easily have switched to a full scale house, a city street, a green valley and much more from a huge selection of 3D frames that are even free to download. The reason behind our choice was opting to solve basic problems in the simplest of environments without needing to consider any outside parameters, like wind for example. Even with that, the simulations performed by Unity are very realistic, with a number of effects occurring that we didn't even account at first. We can even alter the speed of the simulation, in order to observe multiple probable solutions in a fraction of the time that would need under normal circumstances. All these, made Unity a must use tool for our framework, with many more possible additions for future expansion.

4.5 Python

The code written with the Python programming language is the conjunctive "substance" that interlinks all the components together into a single framework. It receives all the available data from the UI, transfers it to the Clingo app and returns the results back to Unity for the simulation. However, this is just an oversimplification of the actual work being done, omitting a number of operations that are crucial to the functionality of our project, and so worth mentioning below.

Data transformation is the first pivotal task of Python. Unity expresses objects by using 3-dimensional axis data, a type of information that is not recognized by the Event Calculus, thus it has to be processed and altered into suitable fluents. Once a fluent has been declared, Python stores the 3-dimensional pattern and uses it to distinguish when it applies for a pair of objects. Reversing the process also falls under its jurisdiction, since the best clingo can do is output the solution in sets of fluents. At each step the newly occurred action must be located and the involved objects have to be transformed into correct axis data in order for Unity to apply the changes. In a gist, Python performs the role of a "translator" between two high-level languages that don't have a common ground for communicating directly.

Recognizing patterns in order to alter data to a suitable form, entails the pre-existence of proper knowledge. In order for Python to achieve something like that,

it has to possess **storage capabilities**, dynamic enough to be portable, and easy enough to access and debug in case of occurring faults. So instead of using a traditional database framework like MySQL, MongoDB etc, we used TinyDB, a small external python library that unitilizes a document oriented database optimized for portability. All the data are stored in a dictionary form inside json files, making it quite easy to read even with a simple text editor like notepad. The fact that no extra process is involved makes it ideal for using in multiple computers that in any other case, had to have an already installed version of the database in order for it to work. Thanks to that, we can store any number of data, ranging from the pattern of fluents, scenes saved for future use, occurring errors, to even all possible solutions for a specific scenario, step by step with all the requisite info for proper simulation in the Unity engine. An important note here, is the fact that Python has to be present for all this to work, however, we have created a portable version of it, eliminating almost all other dependencies that may occur (OS is still a limit).

```
>>> from tinydb import TinyDB, Query
>>> db = TinyDB('path/to/db.json')
>>> User = Query()
>>> db.insert({'name': 'John', 'age': 22})
>>> db.search(User.name == 'John')
[{'name': 'John', 'age': 22}]
```

Figure 4.2: An example of TinyDB

On a final note, the most important task of python is performing the **mathematical calculations** needed for adding and moving the objects inside a simulation. In combination with data trasformation, python is responsible for estimating the correct location of a moving object in regards to its size and rotation as well as to the second object that it will relate to. It is a tedious procedure, blending a lot of structural data, that if appointed to Unity, it would significantly slow down the simulation. For that reason, all the calculations are done beforehand in the backend by python, leaving the frontend of Unity to better focus on the running of the simulations. One thing worth mentioning is the introduction of "randomness" when moving objects, by adding a random small number to all actions, as a way to simulate the "human error". A second reason for that is that by default Unity gives the 3-d axis info from the center of mass of an object, meaning that if for example you stuck up objects using their mass center, you will achieve balance no matter what the object may be. For humans, something like that, although not impossible, is not probable when executing daily routines. However, this random offset technique, presented us the opportunity to actively change a set of solutions for a specific problem, by introducing a user specified offset that is added dynamically by python to the final data sent to Unity, without the need to recalculate all

steps from the beginning, making the whole process of simulation even faster.

Chapter 5

Architecture and Implementation

In the previous chapter we presented the basic technologies/tools that we have unified in order to create our framework. The purpose of this segment is to explicate as to how these pieces come together to form the architectural base of our system. We will use information flow diagrams as a means to better convey the interconnections that take part in order for the application to simulate a given scenario.

5.1 Architecture

Presenting the architecture of our framework will be kept to as plain as possible, since a number of functions entail too much technical details that will only confuse rather than clarify the whole process. We will depict a simple flow of information among the different parts by using a basic example, pointing some vital operations that the reader has to know for better understanding the overall system.

To set things up, we first begin by mentioning the versions of the tools we used. For Unity we began with version 3.4 but eventually settled down to 4.6.4 since it had more available tools, was more stable, and implemented easier physics mechanics than its predecessor. At this moment in time, there are more advanced versions available, but we will present in a different subsection why we decided to stick with this one and avoid the transition to the latest. For Clingo we used the version 4.5.4 from the beginning till the end without any issues, so updating to a newer version (considering the changelog) was deemed unnecessary. As for Python, we decided to go with the version 3.4.3, the latest stable configuration that most python developers use, being compatible with a great number of add-on libraries that where previously only available to Python 2.

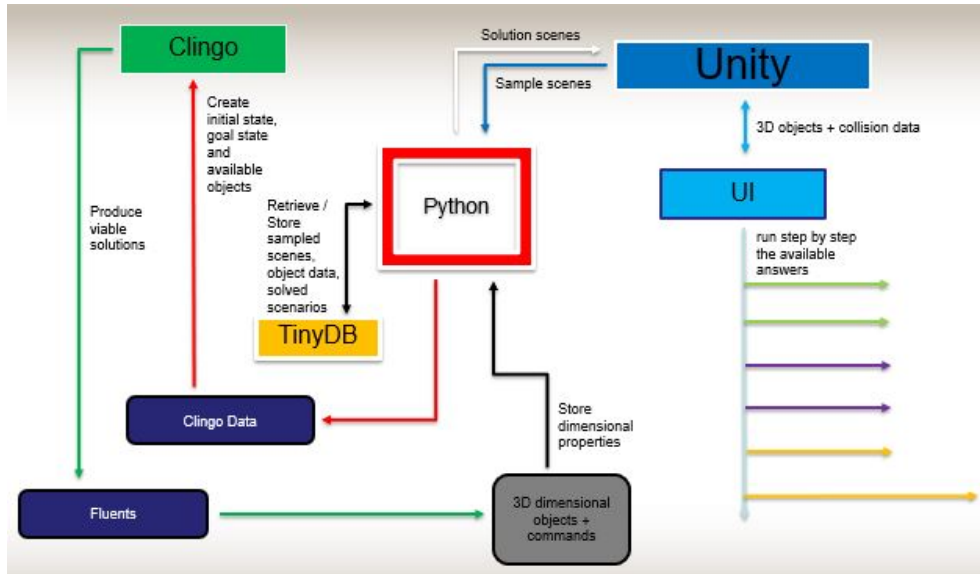


Figure 5.1: The General Architecture

Figure 5.1 shows a rather simplified diagram of our project's architecture in order to acquire a rough idea of how it functions:

- At first we have the Unity part which integrates the UI portion of the framework. The user interacts with the system there by creating information (objects etc) and setting up "scenes" for testing various scenarios at real time.

To create a scene the user has to first create some objects to inhabit the pre-existing environment. For example he can make two cubes of any desired size, with a number of properties and even of pre-defined material like wood or metal. The user can alter all the properties of an object (except the basic shape) at any given time, move it around, change its rotation, and even create structures by clustering them together and using the custom tags.

Once we have created all the desired objects, we then have to create fluents in order to set up patterns for Unity to look for. For instance, the user can create the fluent `onTopOf` by transferring an object A on top of an object B and setting their connection as two colliding objects with the one in focus being higher in the Y-axis than the second one. This part is very important since it will produce the initial state of the environment necessary for our clingo files. Without any set fluents Unity will just save the positions of each object, meaning that basic connections like "object A is on the floor" will not be apparent to it. In 5.2. we see a representation of the example of the `onTopOf` fluent. What someone needs to pay attention to is the *Status* and the *Dominant Parameter* keys. The first one refers to the three axis X,Y and Z in our 3-D world and by that order it shows what is happening between our main object and the secondary one in those specific terms. The Y-axis

reads "Higher" meaning our object is above the secondary one in terms of height and what's more its actually on top of it since their *Proximity* is set on "Colliding". The reason why the X-axis and the Z-axis are blank is because of the *Dominant Parameter* key that turns the focus on the "Y Dimension" and informs the system to ignore all others. One minor key element here is the fact the objects we used for our example are not important and we could have omitted them since their purpose was to be used as a point of reference.

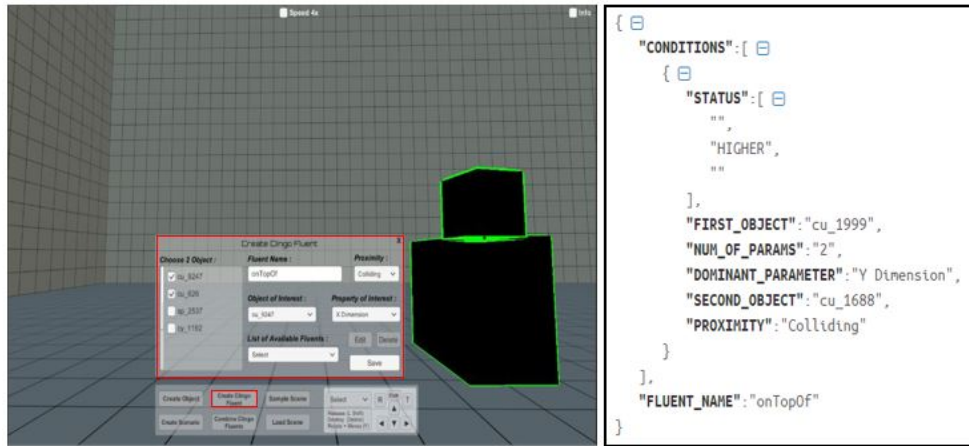


Figure 5.2: How to Create a new Fluent

- All the information is extracted and sent to the Python module, where data from the user (the fluents) will be used as guidelines to transcode the rest of the knowledge from the environment and then be stored into json files using the add-on database TinyDB.

The data is stored in a form easy to inspect by any user who wishes to see it for any number of reasons. In the following Figure 5.3 there are photo segments, with the first one being the raw formula someone will observe when opening the file with a common notepad. There are a number of tools and online applications to shape up the information into the second segment, a well defined xml-like form with a "key" connected to each info for clarification of what it depicts. It is important to understand that the json files are the actual database, meaning that any change will have impact on the corresponding scenario. We can even use that to create alterations of the same scene, like testing the possibilities of building a tower with only cubes, or only cylinders etc.


```
[{"default": {"16": {"COLLISIONS": [{"cu_9247;floor],[cu_626;floor],[sp_2537;floor],[cy_1192;floor]"}, {"OBJECTS": [{"X_SPACE": "2.586569", "BOUNCINESS": "0", "DYNAMIC_FRICTION": "0.45", "Z_SPACE": "1.517825", "X_SCALE_GLOBAL": "0", "OBJECT_TYPE": "cu", "Y_SCALE": "1", "X_SCALE_GLOBAL": "0", "OBJECT_NAME": "cy_1192", "Z_SCALE": "2", "MATERIAL": "Wood", "X_SCALE_GLOBAL": "0", "STATIC_FRICTION": "0.45", "TAG": "Clingo", "X_ROTATION": "-0.0001329864", "MASS": "1", "Y_ROTATION": "-6.758945E-11", "Y_SPACE": "1.500005", "X_SCALE": "3", "Z_ROTATION": "5.824041E-05"}, {"X_SPACE": "7.667865", "BOUNCINESS": "0.5", "DYNAMIC_FRICTION": "1", "Z_SPACE": "1.601178", "Y_SCALE_GLOBAL": "3", "OBJECT_TYPE": "sp", "Y_SCALE": "3", "X_SCALE_GLOBAL": "3", "OBJECT_NAME": "sp_2537", "Z_SCALE": "3", "MATERIAL": "Rubber", "X_SCALE_GLOBAL": "3", "STATIC_FRICTION": "1", "TAG": "Clingo", "X_ROTATION": "-0.0001017358", "MASS": "1", "Y_ROTATION": "-1.599901E-10", "Y_SPACE": "1.5", "X_SCALE": "3", "Z_ROTATION": "0.000180207", "X_SPACE": "-1.811259", "BOUNCINESS": "0", "DYNAMIC_FRICTION": "0.15", "Z_SPACE": "6.150385", "X_SCALE_GLOBAL": "4", "OBJECT_TYPE": "cu", "Y_SCALE": "4", "X_SCALE_GLOBAL": "3", "OBJECT_NAME": "cu_626", "Z_SCALE": "5", "MATERIAL": "Metal", "X_SCALE_GLOBAL": "5", "STATIC_FRICTION": "0.15", "TAG": "Clingo", "X_ROTATION": "2.392592E-05", "MASS": "1", "Y_ROTATION": "388.1117", "Y_SPACE": "2", "X_SCALE": "3", "Z_ROTATION": "-7.274040E-06"}, {"X_SPACE": "-5.463859", "BOUNCINESS": "0", "DYNAMIC_FRICTION": "0", "Z_SPACE": "1.426427", "Y_SCALE_GLOBAL": "2", "OBJECT_TYPE": "cu", "Y_SCALE": "2", "X_SCALE_GLOBAL": "2", "OBJECT_NAME": "cu_9247", "Z_SCALE": "2", "MATERIAL": "Custom", "X_SCALE_GLOBAL": "2", "STATIC_FRICTION": "0", "TAG": "Clingo", "X_ROTATION": "1.238335E-05", "MASS": "1", "X_ROTATION": "347.3904", "Y_SPACE": "0.99999999", "X_SCALE": "2", "Z_ROTATION": "-1.86653E-06", "X_SPACE": "0", "BOUNCINESS": "0", "DYNAMIC_FRICTION": "0", "Z_SPACE": "0", "Y_SCALE_GLOBAL": "0", "OBJECT_TYPE": "cu", "X_SCALE": "80", "X_SCALE_GLOBAL": "0", "OBJECT_NAME": "floor", "Z_SCALE": "80", "MATERIAL": "Terrain", "Z_SCALE_GLOBAL": "0", "STATIC_FRICTION": "0", "TAG": "Terrain", "X_ROTATION": "90", "MASS": "0", "Y_ROTATION": "0", "Y_SPACE": "0", "X_SCALE": "80", "Z_ROTATION": "0"}, {"SCENE_NAME": "Building"}]}}
```

```
{ "default": { "16": { "COLLISIONS": [{"cu_9247;floor],[cu_626;floor],[sp_2537;floor],[cy_1192;floor]"}, "OBJECTS": { "X_SPACE": "2.586569", "BOUNCINESS": "0", "DYNAMIC_FRICTION": "0.45", "Z_SPACE": "1.517825", "Y_SCALE_GLOBAL": "0", "OBJECT_TYPE": "cy", "Y_SCALE": "1", "X_SCALE_GLOBAL": "0", "OBJECT_NAME": "cy_1192", "Z_SCALE": "2", "MATERIAL": "Wood", "Z_SCALE_GLOBAL": "0", "STATIC_FRICTION": "0.45", "TAG": "Clingo", "X_ROTATION": "-0.0001329864", "MASS": "1", "Y_ROTATION": "-6.758945E-11", "Y_SPACE": "1.500005", "X_SCALE": "3", "Z_ROTATION": "5.824041E-05" } } }
```

Figure 5.3: The Sampled Data

- After that, Python will use the pre-available clingo files (provided by the user) to create new files adding the "scene data" from Unity in the form recognized by the Clingo solver. One key note here are the pre-existing clingo files, which is the main reason why our application is not for novice users, but it targets those who have experience with ASP (Answer Set Programming) and the Event Calculus. To create a fluent like onTopOf is only enough for the Unity Engine. For Clingo to run, we need to create the predicates, events, rules and the goal state in ASP in order for it to work. On the following snippet we depict the domain axiomatization for our first use case as it exists inside the file :

```
% Domain Axiomatization
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
initiates(moveObjTo(X,Y),onTopOf(X,Y),t) :-
object(X), object(Y),
free(X,t), free(Y,t).
```

```
terminates(moveObjTo(X,Y),onTopOf(X,Z),t) :-
object(X), object(Y),
free(X,t), free(Y,t),
holdsAt(onTopOf(X,Z),t).
```

```
above(X,Y,t) :- holdsAt(onTopOf(X,Y),t).
above(X,Y,t) :- holdsAt(onTopOf(X,Z),t), above(Z,Y,t).
```

```
% If there is an object on top of another, the latter is covered.
```

```
% The floor is never covered.
covered(Y,t) :- holdsAt(onTopOf(X,Y),t), Y!=floor.
free(X,t) :- object(X), not covered(X,t).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

- Clingo will run the scenarios and produce plans, which are sequences of valid actions that lead to the goal state. An example can be viewed in Figure 5.4 below, where we tried a variation of the first use case, resulting in 6 possible models that solve the problem of stacking objects on top of each other. Every action has a numeric indicating the time moment it "holds" true, and at every moment only one action can occur. In this example, besides the initial state, it takes two moves for each plan to reach the goal state.

```
Answer: 5
holdsAt(onTopOf(cu_9931,floor),1) holdsAt(onTopOf(cu_7652,floor),1) holdsAt(onTo
pOf(cu_8995,floor),1) holdsAt(onTopOf(cu_9931,floor),2) holdsAt(onTopOf(cu_7652,
floor),2) holdsAt(onTopOf(cu_8995,cu_9931),2) holdsAt(onTopOf(cu_9931,floor),3)
holdsAt(onTopOf(cu_8995,cu_9931),3) holdsAt(onTopOf(cu_7652,cu_8995),3)
Answer: 6
holdsAt(onTopOf(cu_9931,floor),1) holdsAt(onTopOf(cu_7652,floor),1) holdsAt(onTo
pOf(cu_8995,floor),1) holdsAt(onTopOf(cu_7652,floor),2) holdsAt(onTopOf(cu_8995,
floor),2) holdsAt(onTopOf(cu_9931,cu_7652),2) holdsAt(onTopOf(cu_7652,floor),3)
holdsAt(onTopOf(cu_8995,cu_9931),3) holdsAt(onTopOf(cu_9931,cu_7652),3)
SATISFIABLE
Models      : 6
Calls       : 4
Time        : 0.450s (Solving: 0.43s 1st Model: 0.00s Unsat: 0.08s)
CPU Time    : 0.016s
```

Figure 5.4: A Clingo Outcome

- The plans are parsed by Python, transcoded into 3-D data and saved on the database. The format is the exact same in Figure 5.3, which is the one that Unity recognizes.
- The user can finally simulate any solution by a corresponding list that reads the available data from the appropriate file in the database and step by step moves the object to their correct location. A small snippet of the code for moving the objects is right below:

```
IEnumerator alter_existing_object(List<string> new_object, GameObject existing_
{

    // Get the 3 size variables of the current object
    float size_x = float.Parse (new_object [2]);
    float size_y = float.Parse (new_object [3]);
    float size_z = float.Parse (new_object [4]);

    // Stop the object from moving, load its correct rotation and mass
```

```

Rigidbody old_rb = existing_obj.gameObject.GetComponent<Rigidbody> ();
old_rb.velocity = Vector3.zero;
old_rb.angularVelocity = Vector3.zero;
existing_obj.transform.eulerAngles = new Vector3 (float.Parse (new_object [8]), float.Parse (new_object [9]), float.Parse (new_object [10]));
old_rb.mass = float.Parse (new_object [11]) * 10;

// Find the shape of the Object
Collider capColl = new Collider ();
if (new_object [1] == "cu") {
capColl = existing_obj.GetComponent<BoxCollider> ();
existing_obj.transform.localScale = new Vector3 (size_x, size_y, size_z);

} else if (new_object [1] == "sp") {
capColl = existing_obj.GetComponent<SphereCollider> ();
existing_obj.transform.localScale = new Vector3 (size_x, size_y, size_z);

} else if (new_object [1] == "cy") {
capColl = existing_obj.GetComponent<CapsuleCollider> ();
existing_obj.transform.localScale = new Vector3 (size_x, size_y, size_z);

}

```

5.2 Implementation

Each of the following subsections presents the general functionality of a part of the framework in terms of implementation information flow under the unified architecture. We have broken down the general architecture diagram presented in Figure 5.1 to a more detail depiction for each "machinery", a necessary step in order to fully realize the scenarios portrayed in chapter 6.

5.2.1 The UI/Simulator

The Unity Engine is an extremely handy platform since it not only automatically integrates the physics engine in all the scenarios we simulate, but it also provides us with a visual toolset for creating a user interface for all the necessary interactions. Figure 5.5 shows the basic setup for the front-end of the application. The User Interface is the main access point from which the user can set up the components needed for a scenario. At this time the available input are the 3-D objects and the Fluenta.

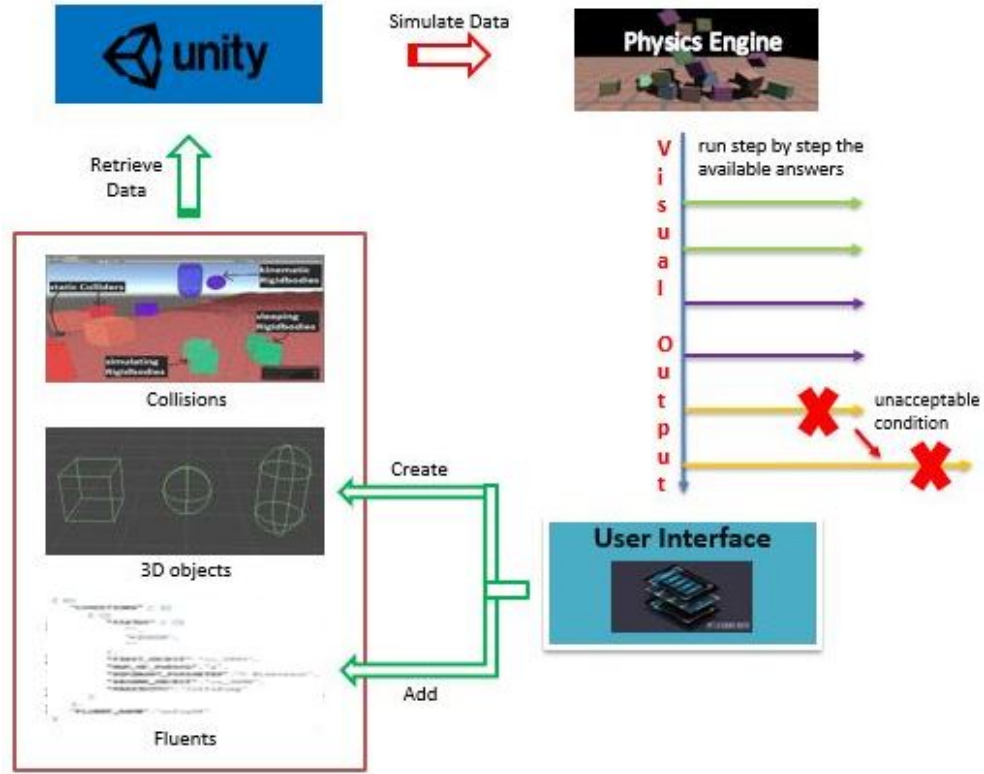


Figure 5.5: The Front-End Architecture

Considering the complexity, we chose to keep things at an elementary level and not introduce any premade intricate objects, for reasons we will explain at a later subsection. Since we have already covered a good portion of the functionality of the UI in the subsection above, here we will mostly focus on how this functionality is provided to the user.

The first thing someone will do is create objects and build up a scene. This is very easily done through a corresponding menu providing a number of options all visible in Figure 5.6. The user can only alter the properties of a custom object. If another type of "fixed material" is selected, those properties are already set by Unity itself, representing actual materials in the real world. After creating an object, the user can move it around by selecting it from the movepad list on the central menu, and even alter it from a similar menu as the one he created it (everything except the shape).

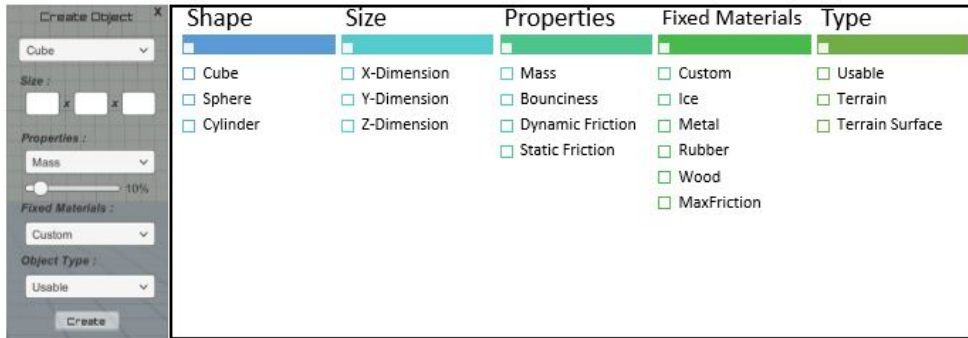


Figure 5.6: The Object Menu

As we have already mentioned in the previous subsection, adding a fluent to the system is not so much as creating something new, but rather teaching the system how to interpret specific object groups, in order to translate their orientation inside the environment into a format that can be later understood by Clingo on the back-end. The representation of fluents in Figure 5.5 can be viewed clearly in Figure 5.2 along with UI for creating it.

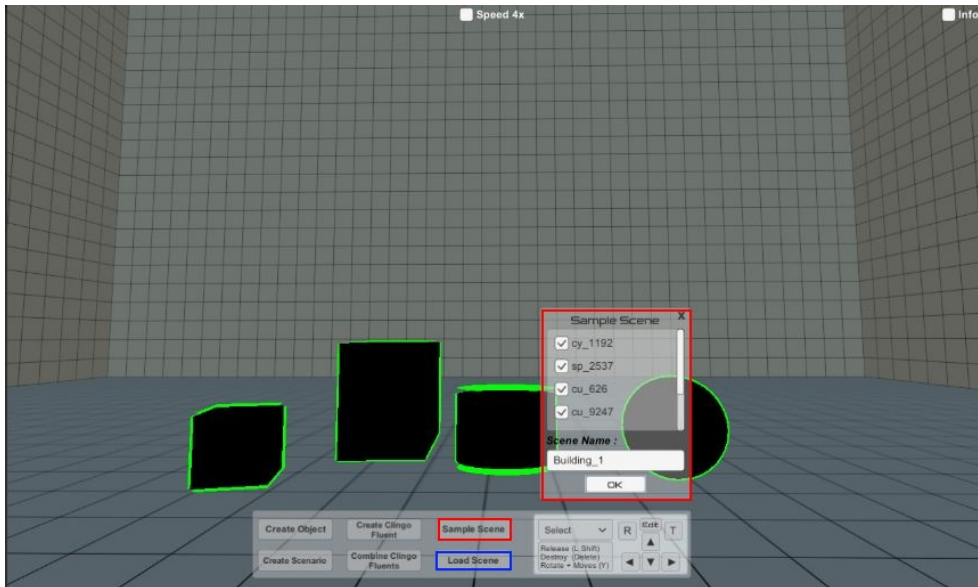


Figure 5.7: Saving a Scene

All the info about the objects and their inner interactions (collisions) is then *sampled* by the Unity engine by using the button *Sample Scene* and choosing all the objects we wish to include in our simulation. Every chosen objects is highlighted for the convenience of the user. In Figure 5.7 we can see the corresponding menu and the selected items ready to be saved. We can enter a small name for disguising our new creation in case of future use from a list presented to us when pressing the

Load Scene button as it also marked with a blue outline. One small detail here is the fact that the names of the objects are decided at random by the system and not the user preventing the use of any special characters that might cause an error when parsing the data later on.

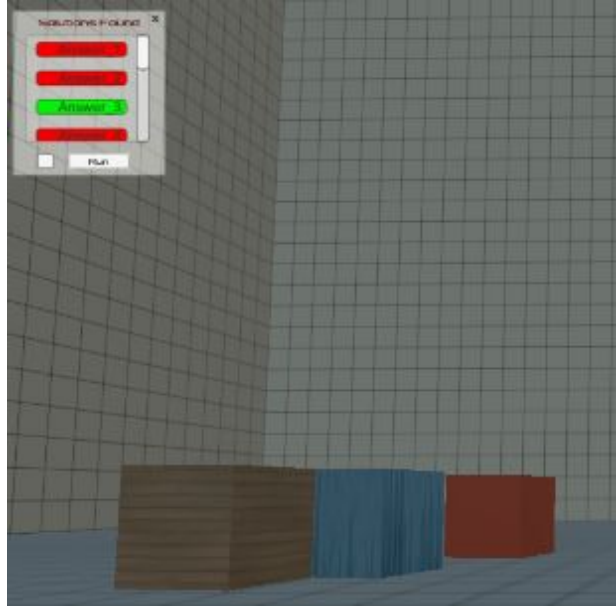


Figure 5.8: The Timeline of Solutions

Once it is processed by the back-end the possible solutions come back in the same form, separated in different answer sets with each further analyzed to a number of steps. From there, each step is subjected to the rules of the physics engine, in terms of validity. If an answer fails, answers with similar conditions will also be flagged as failures. An example can be seen in Figure 5.8 where we chose to test only solutions 1 and 3, but because the first one failed it also flagged solutions 2 and 4 that had a similar pattern of steps. All the simulations are available for viewing by the user with the option to speed up the time flow in case there is a large number of possible plans. To make things easier, we also provided information about the fluents involved in each solution. Finally, when simulating a scenario, actions are not depicted as actually seeing objects move around in the environment, but rather as instant placements to the end location of each step. Animating actions was not of interest to our experiments and they would require additional computations and a lot of time even at a minimal level.

5.2.2 The Solver/Reasoner

The Clingo Reasoner is the central component of the back-end of the application. It's where the predicates of the Event Calculus that map the objects and the interactions between them, are used in order to produce a number of possible

solutions that will then be tested from the physics engine, determining their actual validity for use.

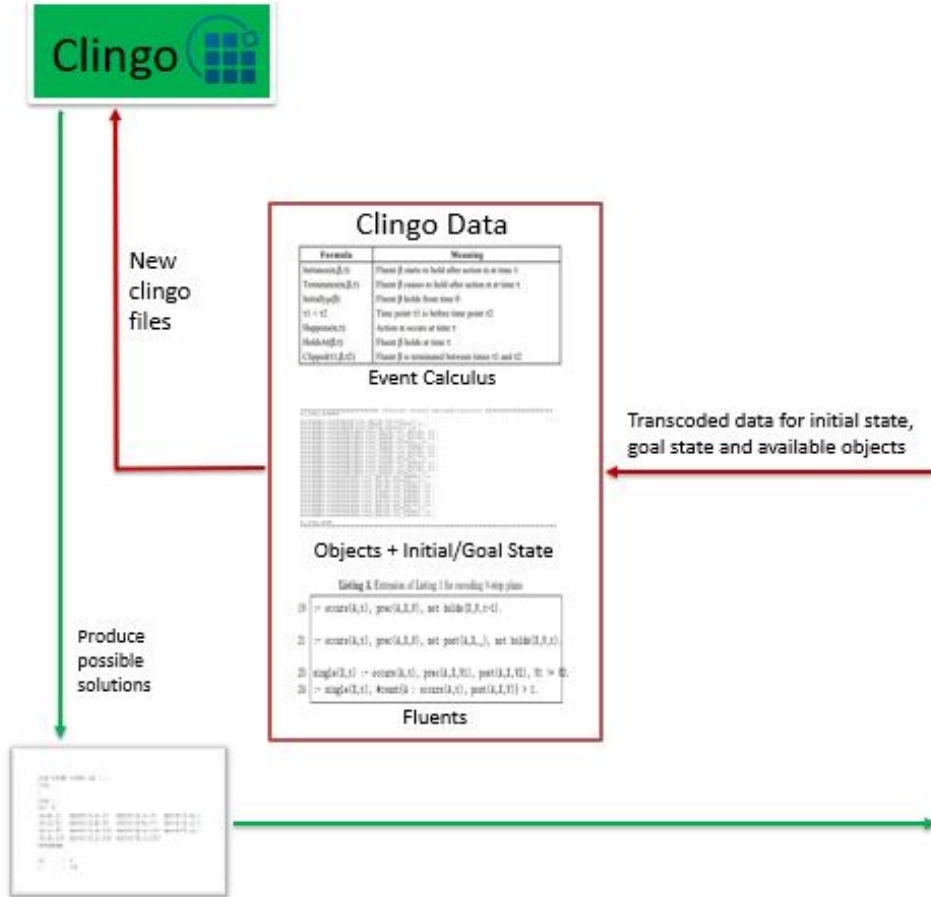


Figure 5.9: The Back-End Architecture

As someone can observe from Figure 5.9 the data needed for executing a clingo scenario is already transcoded to the appropriate form by the Python module. We have our objects, the fluents that exist inside the database and the initial state of the scenario (the position of each item inside the room as selected from the user) expressed in these fluents. All the rest of the information like the constraints and the causality rules, any possible events and finally the goal state, pre-exist inside each clingo file. Once a possible scenario has been chosen, the system will combine all this data to create a new final *.lp* file that will be sent to Clingo for execution. Once all possible outcomes have been calculated in the form of the already known fluents, Python will receive the entire result and start the reverse process of transcoding it back to axis data. One significant part here, is the fact that the most important functionality of Clingo happens inside of it and it entails the

rules and constraints that dictate the actions allowed at each step of determining a possible solution.

5.2.3 The Intermediate

The link between the back-end and the front-end of the application is the domain of the Python module. The choice for developing all these necessary functions with Python was based on the sole premise of an existing extensive background in back-end software development using the Python programming language. Still, there are numerous advantages one of which is portability, meaning there is no need for someone to install external libraries in order to run the application.

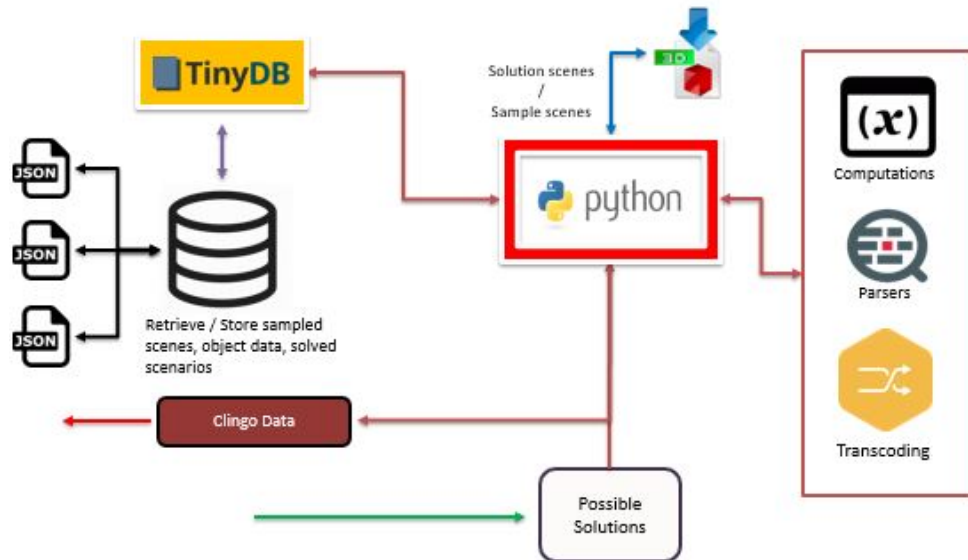


Figure 5.10: The Intermediate Architecture

In the above Figure 5.10 we get a glimpse of the responsibilities of Python, which we will immediately list in a little more detail for better comprehension:

- Once Unity samples a "scene" all the information is transferred to the Python module where it will be sorted out into different types.
- The outcome will then be stored to the TinyDB database which is a document-oriented system, with each file being a "table" using the traditional terminology for databases. TinyDB is perfect not only because there is no need for a background process to be running but also for the fact that each file has an organized, user-friendly structure that can be easily accessed with just the use of a simple notepad app.

- After that, Python will combine the available data in order to produce the predicates and fluents necessary for the correct execution of the clingo files. Initially, it has to parse the pre-existing files provided by the user, insert the new lines in the correct locations, and create new files that are ready for use. Finally, it will "summon" the clingo application providing all the input (including possible extra parameters) for it to run.
- Once clingo finishes the scenarios and produces the solutions in the form of (recognizable) fluents, Python will again parse all the outcome, divide it into different solutions with multiple steps, calculate the location (rotation etc) of each object in each step (if that changes) and save everything on the database.
- The last step is to send the transcoded 3-D axis data to Unity in order for it to instantiate the objects into their correct locations.

Chapter 6

Use Cases

The purpose of this chapter is to effectively demonstrate the innovations presented by our application. In order to achieve that we are going to present different cases-scenarios, starting from our intentions in selecting each circumstance, explicating what makes it so challenging, and finally correlating them with real setting situations, thus revealing their overall experimental value.

6.1 The Tower of Babel (a variation of the blocks world problem)

Scenario: *We are given an arbitrary number of objects of different shapes and sizes. We are asked to find one or more ways to build a single, stable vertical block (tower) using all objects.*

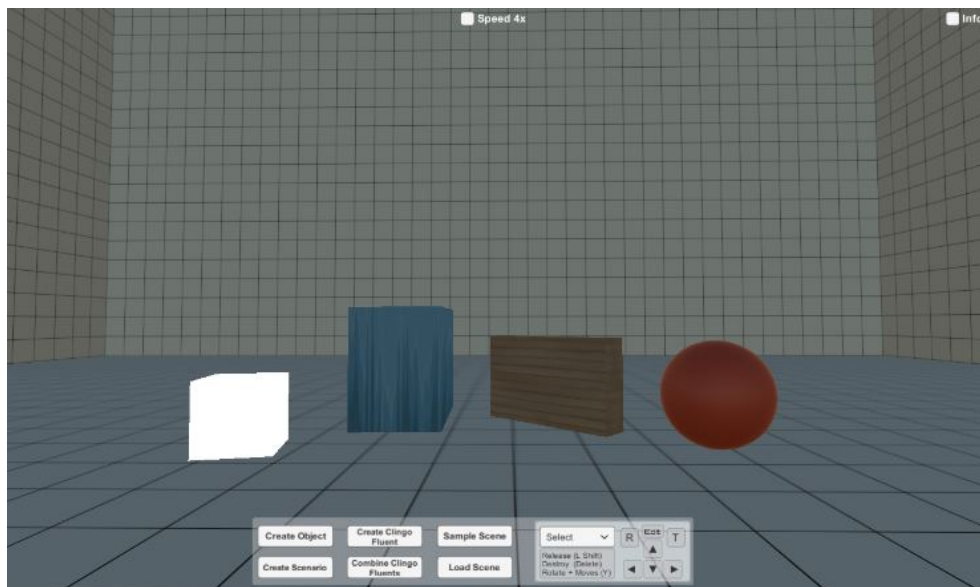


Figure 6.1: A random initial state for creating a tower

The name for this scenario is a little extravagant, however its fundamental purpose is to explore the interactions between arbitrary types of objects by trying to stuck them together creating a single solid structure. As we can see in Figure 6.1 we have 4 items each of different shape, size and weight, and our target is to test the feasibility of all the combinations that exist in constructing a "tower" out of them. The means to produce these objects has already been demonstrated in subchapter 5.2.1 while presenting the implementation of the UI/Simulator. What we omitted was the capability of rotating our items by any angle in all three of the existing axis.

Since our scene is ready we then have to consider how our system will understand the task of building a tower from our provided objects. What we need to teach our application is the concept of putting two objects together by adding one on top of the other. The way to do that is by creating a new *Clingo Fluent* from our main menu. We already have presented such an example in the previous chapter in Figure 5.2 along with the information on how to create it using the corresponding pop-up menu. The next step is to "sample" the scene storing it in our database.

Now we are going to select the *Create Scenario* option, choose the previously saved scene which will be the initial state of the problem and pick as the goal state the scenario we want to test, as we can clearly see in Figure 5.3. The initial state is easy to understand since it's the objects at specific places. The "odd" is how to set the goal state in an abstract way, so that it is applicable to all kinds of objects and not just the ones we have created here. To do that we used the EC to express the code snippet below, which basically states that " at any given time, our tower is not done, if there are at least two different objects sitting on the floor", so in order for a plan to be created, *there can be only one*.

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
tower(notDone,t) :-
    object(A), object(B), A != B,
    holdsAt(onTopOf(A,floor),t),
    holdsAt(onTopOf(B,floor),t).

tower(done, t) :- not tower(notDone,t).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Definition of Goal State %%%%%%%%%%%%%%
%_GOAL_START

:- query(t), not tower(done,t).

%_GOAL_END
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

Once all the possible plans have been created, a new window will appear on the left top corner, containing each possible plan, ready to be tested for its feasibility. For better understanding, there is an *option* on the right corner where the user can view the moves performed in each answer in the form of the fluents he created.

As luck would have it, running the first answer gave us a viable solution, placing the sphere, which is the most problematic object, at the top of the tower as seen in (a) of Figure 6.2. In (b) we choose to simulate answer 3, which uses the sphere as the basis of the tower. The scenario fails, marking the answers 3 and 4 with red, since they follow a similar process. What this means, is that in both cases the white cube is placed on top of the sphere while the other two objects are placed on top of it with two possible ways, hence answers 3 and 4 for each one them. At this point our first test is over. At the beginning, we also mentioned weight as a varying factor but considering the simplicity of the final structure here, we will better explore this parameter in the second and third use cases, that contain a more appropriate set up.

The aim of this first use case, if not apparent, correlates to how humans tend to group objects in order to perform some task. A possible example is cleaning the table after a meal. We have all sorts of utensils like plates, glasses of water, cutlery, bottles etc. The usual approach is to stuck those items and try to move as many as possible at once, without breaking anything. To us, this task is pretty easy and even then there are unfortunate cases where we miscalculated and something fell. If we wanted to use a robotic assistant, he would have to be able to understand how to place the objects in order to transfer them safely, without the need to let him test and brake a few dozen of them until he gets it right.

Finally, it is imperative to mention that even though this use case seems quite simple, there was a great challenge connected to the geometrical parameters of the available objects and more specifically with the aspect of rotation, which is

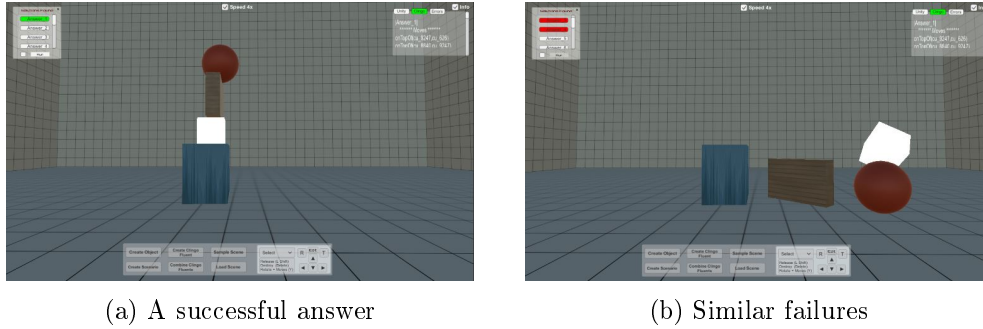


Figure 6.2: Timelines for Use Case 1

important since our system moves and places the objects by using the center of mass as the anchor, while keeping the original rotation intact. What this means, is that the process of bringing two objects together is much more complex, because we have to take into account which of their sides is coming into contact. A good example is a bottle, and the ways we place it on a table, either by its flat bottom, or its round side. In either way, the distance between the table and the bottles center of mass changes so we needed to create a code inside Unity in order to produce a new triplet of dimensions for each object based on their rotation, conventional to the one that our environment is based on.

6.2 Test the Balance

Scenario: *Inside the room we have a tabouret like structure. We need to move the objects from the floor on the tabouret without tilting it or causing any object to fall down. To make things a little more complex, we start by already having an item placed on top of the tabouret.*

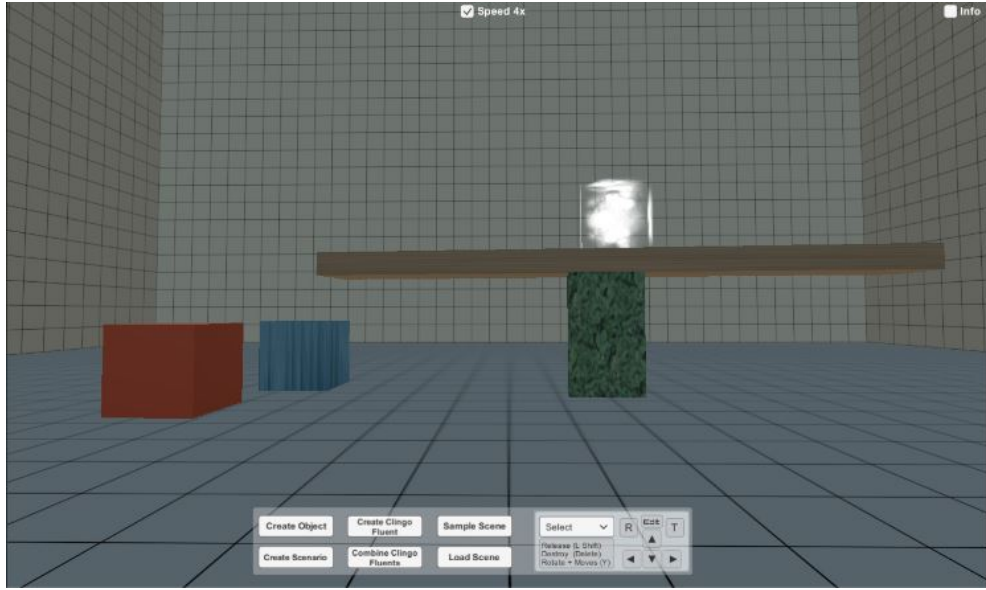


Figure 6.3: A random initial state for balancing objects

The purpose of this scenario is to create and introduce a physical structure to our environment that the system has to use in order to reach a goal state by setting object to interact with it. A more visual example would be a robot transferring items using a tray (like a waiter) or an improvised tray. As we can see in Figure 6.3 the objects are similar in shape, however their weight is different. Additionally, our new table item has a new dynamic allowing more than one items to be on top of it. Finally, the user has a lot more participation to the outcome, since he can make alterations that may turn rejected plans to feasible ones.

To start things of, in order for the user to create the tabouret, he will typically start by creating the two parts, a base and a surface. The important part here is the fact that these items have to be different in one aspect, they are not viable for moving around. To do that, we are going to use the *Object Type* field as presented in Figure 5.6, which has three potential values. The *Usable* option is for simple object we can move, the *Terrain* option is for creating items that should be completely out of reach from our system (can't move, can't interact and can't affect our plans) and finally the *Terrain Surface* is for objects that can't be moved but have the properties of a surface by adding multiple items on it. The base of the table is a *Terrain* object while the surface as someone can easily guess is a *Terrain Surface*. After that, we move all parts to their respected places and save the scene to our database.



Figure 6.4: New Fluents

In this use case, we introduce two new similar fluents that enable us to place items right next to each other from either side. The process is similar to the "onTopOf" fluent in the first use case and the resulting information is visible in Figure 6.4. What we need to notice here is the proximity, which is set to "Global", meaning that two items don't have to touch for the connection to apply. The consequence is that more than one object can be on the left or on the right of a specified item, creating a high complexity that it is too difficult to handle in the EC, thus we added a restraining code in Python, to only keep one set of each active fluent for each object. One important aspect here is the fact that these two new fluents are actually connected to each other, because by adding an object A on the right of an object B works bidirectionally with object B being on the left of object A and the same principle applies when removing it. The following ASP code shows that in order to create the new fluents we need two new actions of "picking" and "holding" an object, meaning that moving an object needs two timeslots, one where we pick it up, and one where we set it down.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% You cannot pick up an object if another one is
% on top of it or if you hold another object
:- happens(pickObj(X),t), not free(X,t).
:- happens(pickObj(X),t), holdsAt(holding(Y),t), X!=Y.
```

```
% In order to put an object to a place, you need to hold it first
:- happens(moveObjTo(X,Y),t), not holdsAt(holding(X),t).
:- happens(moveObjToTableAndRightOf(X,Y),t), not holdsAt(holding(X),t).
:- happens(moveObjToTableAndLeftOf(X,Y),t), not holdsAt(holding(X),t).
```

```

initiates(pickObj(X), holding(X),t) :- movableObj(X).

terminates(moveObjTo(X,Y), holding(X),t) :-
object(X), object(Y),
holdsAt(holding(X),t).
terminates(moveObjToTableAndRightOf(X,Y), holding(X),t) :-
object(X), object(Y),
holdsAt(holding(X),t).
terminates(moveObjToTableAndLeftOf(X,Y), holding(X),t) :-
object(X), object(Y),
holdsAt(holding(X),t).

terminates(pickObj(X), onTheRight(X,Y),t) :-
holdsAt(onTheRight(X,Y),t).
terminates(pickObj(Y), onTheRight(X,Y),t) :-
holdsAt(onTheRight(X,Y),t).
terminates(pickObj(X), onTheLeft(X,Y),t) :-
holdsAt(onTheLeft(X,Y),t).
terminates(pickObj(Y), onTheLeft(X,Y),t) :-
holdsAt(onTheLeft(X,Y),t).
terminates(pickObj(X), onTopOf(X,Y),t) :-
holdsAt(onTopOf(X,Y),t).

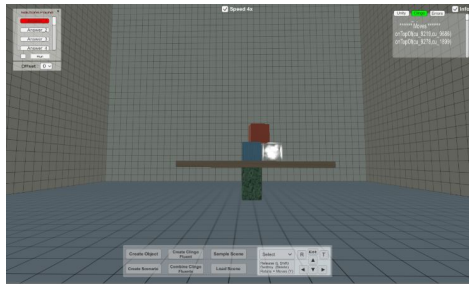
```

%%%

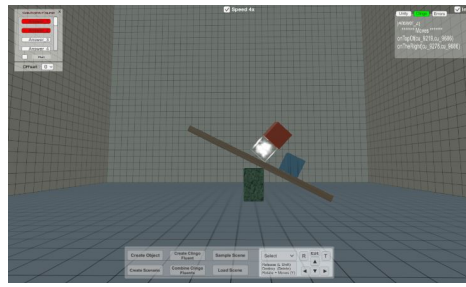
At this point, we run the application and get the available list of plans for simulation. Running a number of those plans, presents us with very interesting results, some of which need a little context for understanding.

Subfigure (a) below was a one-time result, that may seem successful but is actually occurred because the item on top of the table was not set at the center of it, and when Unity tried to place the new object there, instead of causing a volatile collision, the item just shifted away thanks to its properties, resulting in the outcome we see. This is proof, that random factors do exist inside Unity, since we tried many times to replicate the result to no avail. To take into account such conclusion, we decided to remove the restrictions about objects causing a failure to the plan when moving shifting away from their set location due to collisions, thus making the simulation a little more realistic.

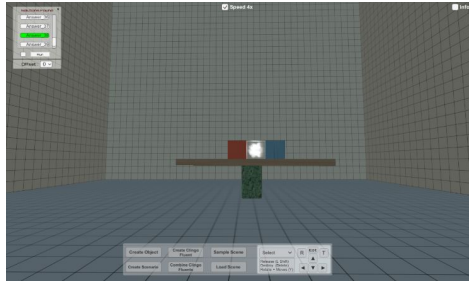
The next figure (b) is a usual fail outcome that causes other similar ones to be flagged along with it. Whats really important here, are figures (c) and (d) because they actually demonstrate the same plan. The reason why the (d) fails is because we allow the user here to be able to change the distance between the central object and the ones we put next to it. This proves that the objects have mass and are



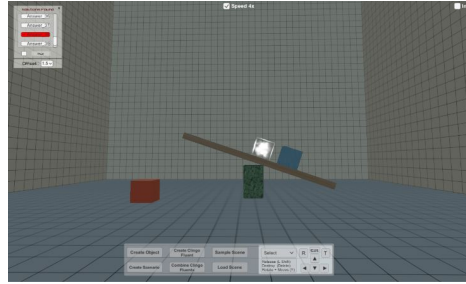
(a) A successful/unsuccessful answer



(b) Similar failures



(c) One object at each side



(d) User input can alter the existing result

Figure 6.5: Timelines for Use Case 2

subjected to real gravity. This also means, that if someone does the math correctly, he can create a pair of objects with the proper mass to make this plan succeed even when adding this "offset".

The purpose of this second use case is to better understand a real environment and can be summarized in the following bullets:

- There is too much information inside an environment and not all of it is useful.
- Surfaces have a capacity and adding more that they can handle in number or weight will result in failure.
- Just because a set plan is successful doesn't mean similar alterations will have the same result and vice versa.
- It is a good practise to make alteration to the plans using the low-level reasoner, since it can lead to better results faster, rather than starting from the beginning resetting the high-level reasoner with the new data.

This use case was much more challenging since we had to take into account a number of parameters that increased the complexity on both the front-end and the back-end of the application, forcing us to limit the range of our information, excluding some very interesting but intricate cases. Still, this set a good basis for creating more realistic simulations for a wider range of problems.

6.3 Natural Properties

Scenario: *There is surface set at an angle with four different objects close to it. We drop the objects on the surface to see which combination will lead to all of them staying on top of it and not "sliding" away.*

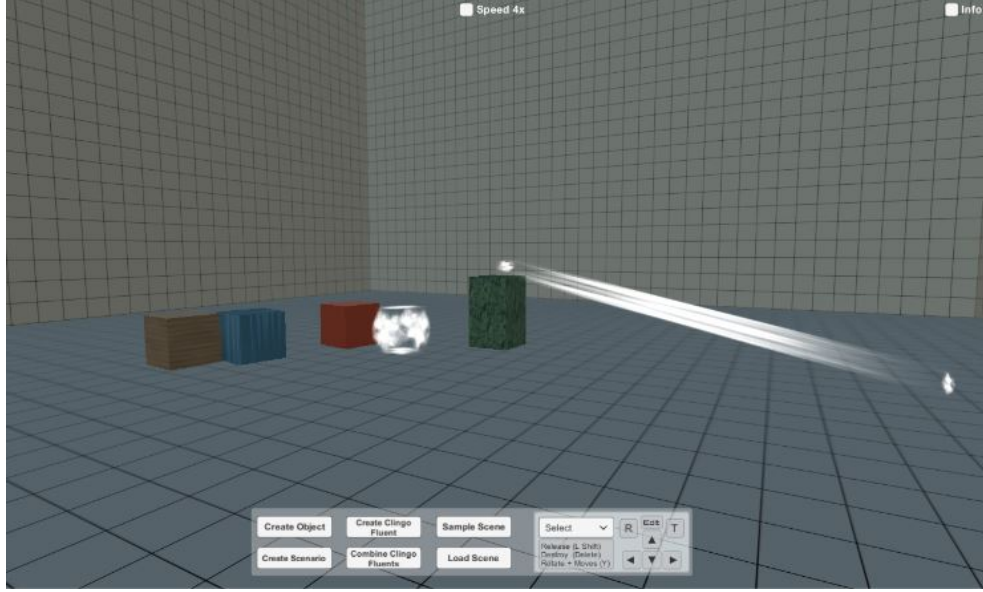
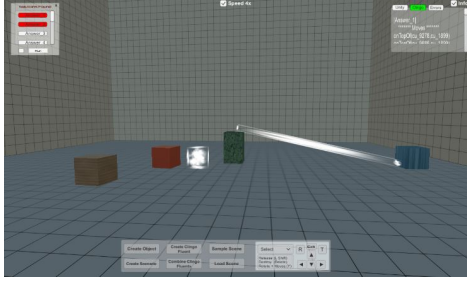


Figure 6.6: The set initial state for Use Case 3

The purpose of this scenario is not so much to solve a specific task in mind, but rather to give more details about the properties that our objects possess. As we can see in the above figure, and also at the ones before it, our objects come with different colors. These actually represent materials like metal, rubber or wood, and all of which come with a preset in elasticity, dynamic friction and static friction, in order to mimic the real material. All this information is available at the get go from Unity, so we only to select it and test it inside a simulation which is what we do in this user case.

Again, we start of with making the necessary objects, and setting them to their proper positions. The surface was selected to be made of ice since it was the material with the least friction, allowing more versatile results than with any of the other materials. For the objects we have 4 similar cubes made from ice, wood, metal and rubber. The pillar on the one end of the surface is made from a material called max-friction which is pretty similar to rubber, so we didn't use it besides that.

For fluents, the one we created in the first use case called "onTopOf" is enough to do the trick. Additionally, as for the EC rules we used to "get" the final goal state, we kept the same structure from the first case, and we only changed a single line, restricting the available place to put an object to only one, on top of the



(a) Similar failures



(b) A successful/unsuccessful answer

Figure 6.7: Timelines for Use Case 3

table. In order to avoid any collisions when transferring the objects, instead of adding multiple locations inside the clingo file meaning more rules and latency, we let Python handle it after the high-level reasoning and during the transcoding back to the UI.

```
% Use Case 1
event (moveObjTo(X,Y)) :- object(X), object(Y), X != Y.

% Use Case 3
event (moveObjTo(X,table)) :- object(X).
```

In the above multi-figure we only see two distinct cases worth mentioning. The first one is the classical failed scenario and how it strikes out the similar timelines. However, on the second one we have the only scenario that was a success even though the picture tells otherwise. The reason this happens is due to the time limit we have before moving on to a new action or a new plan. The last item fell and tilted the surface after the time-limit, creating the conundrum of "how long should we wait, until we declare the end of an experiment ??". It is also important to mention that all the items were dropped from a small height on to the surface affecting its low stability (made of ice) . If we had used a another material for it, we would have more feasible plans. For this instance, if we had a sturdy surface with a similar low friction, only the scenarios that had the rubber as their first item, would be the ones to pass the test.

This final use case was more or less an addition to testing the properties of items from the second case. Our original plan was to try and transfer these types of objects from one tilted surface to another, however we needed a lot more computations for dealing with adding objects on inclined surfaces, a case for future study.

Chapter 7

Conclusions and Future Work

7.1 Future Work

Remarkable progress is being done every day on research into intelligent robots, more specifically, service robots. In recent years, there is a surging interest to better integrate high and low-level reasoning to create more sophisticated and human-wise mechanisms ranging from simple single action tasks to even very complex multi-step processes. One of the most common problems is to sufficiently test and refine the mechanics developed with high-level reasoning in a human environment with real-time conditions. Our work was set to address this problem, by providing a basic framework for easily developing such hybrid systems. The application combines the abstraction of the Event Calculus with the geometrical feasibility checks of the Unity game engine, producing credible outcomes that a robotic system can use in order to be trained to perform a task efficiently.

As we have mentioned multiple times, this software is experimental and is only the basis of what could become a very robust and powerful system given the appropriate time to develop. At this point, only simple tasks connected to movement can be added and further simulated. One of the first steps would be to redesign the system to address more complex objects and clusters, while providing a wider set of custom tags that may be connected to other states an object can experience like heat or fragility.

Furthermore, the mechanics for handling Fluents will be updated, allowing for the creation of a group of fluents that can relate to performing a greater task like "Cleaning the room". We already begun adding this functionality as someone can attest from the option on the central UI, however the complexity was too much to handle at this stage, and many smaller issues must be addressed first before getting to that point.

On a final note, one of the first actions in improving our application, is to redesign the part connected to the Unity engine, updating all its components to a more recent and stable version, that is certain to solve in time some of the inconsistencies and "bad" code practices we had to employ in order to make certain

parts work. This means, a faster, better and more user friendly experience when handling our application.

7.2 Discussion on technical challenges and lessons learnt

So far creating this project might seem as a relatively simple thing, since according to our narrative our components are being pretty easy to handle with a wide range of tools that automatically degrade any complex task to a simple routine, but the truth is far from it. Getting to this point was a rather tedious process with a number of obstacles that are worth mentioning in order to better establish the value of developing such an application.

7.2.1 The Versions

The Unity engine is a remarkable tool vital to simulating the scenarios of our application, however the learning curve was more difficult than someone would expect. One of the main reasons is the fact that Unity even though it is a well established engine (initial release in 2005), in 2016 with the version 4.7.1 that we first installed, it was still underdeveloped in terms of features when compared to its main competitor the Unreal Engine. In the span of two years, Unity had regular updates almost every two months, presenting changes that not only added new components, but in many cases completely redesigned some of the available ones. The first update to Unity 5 broke all the existing objects we had created up to that point, by altering the mesh rendering system that was responsible for handling the physical properties of an object. A second update a few month later did the same altering the way 3-D objects and menus are instantiated inside a virtual environment, stalling the development for almost a month. Considering the implications of future issues, we decided to keep the current version and reject any new updates, a wise decision considering that the first stable version was released on December 2017 and it is far from anything we compared with the ones we use. Taking into account the available changelog, transitioning is a no option since even the add-on simulation environment we have is no compatible at all with that specific version.

The Python programming language may have been one of our initial choices, however it was not in its current form. The development coding language for Unity is C#, an object-oriented programming language which was developed by Microsoft within its .NET initiative. The first idea was to create a completely flexible application by using a version of Python called IronPython which was created for the purpose of being internally compatible with the .NET framework, meaning we could use the Python syntax to write code compatible with .NET object as provided by the Unity engine. Unfortunately, the Unity version we settled on was not yet compatible with the .NET 3.5 version (only up to 2.0) an addition that was much later introduced, forcing us to use the standard Python library, which meant we had to involve command prompt (cmd) of Windows in order to

run the scripts for transferring data from the front-end of the application to the back-end. This tactic has its limits since we can't pass a large number of data at once without causing a sort of "overload" to the cmd. Also, this means that the whole framework is confined inside the Windows OS, even though Unity provides the option for converting the project forms compatible with multiple OSs.

7.2.2 The Learning Curve and Rising Issues

One very important issue when developing a project is how familiar someone is with the components he uses. The Answer Set Programming with Clingo and the Python language were tools already intimate to us from previous academic projects. The same thing cannot be applied to the Unity engine, thus we had to begin from ground zero with everything related to it. At first the existing tutorials were enough to develop simple and later more complex environments. The problems begun to rise after updating the application due to the vast changes be done in many key elements. A more experienced user could more easily follow up with the changes, leaving novice users like us to struggle and wait for new tutorials to emerge within the online community. In some cases, entire segments of code had be rewritten and even now there are a lot of them, that with our current knowledge we can clearly determine far better ways to develop, but are unable to do so without having to alter dozens of functions, leading to months of retesting the code in order to "iron out" all the new errors.

When it comes to writing code, game programming is perhaps one of the most complex tasks, especially for those without previous experience. The main problem is how to "debug" the code when a fault occurs. Traditionally the order for testing a function is in a sequential manner, following a specified order. In game development things are different with multiple functions running in parallel, meaning for example that a specific object might be manipulated by more than one segment of code at the same time. In such cases, it is very difficult to solve an issue when you are not sure which code is responsible, having to do repeated testing again and again in order to determine the culprit. Again, experience is vital for creating a more appropriate type of code for such an application as this.

Another serious issue here, is the overall complexity of the project that is raised exponentially. The more features we added to the project, the more difficult it became to maintain a proper functioning order. For example, moving an object from a location A to a location B is quite simple. However, when adding the concept of rotation, things get a lot trickier by having to figure out the direction of each surface the object has, in order to correctly place it in the new location. A less confusing paradigm is one that most people playing RPG games have definitely encountered at least once, with their character being stuck inside a solid object. Remember that these are professionally developed games, taking years to create by teams consisting of dozens of experienced programmers, and even then we have these kind of unexpected problems occurring. Our original intentions where to add much more complex scenarios, but the severity of issues that rose inhibited us from

going any further than we have.

Finally, as we mentioned above, there are errors that even the most adept programmers miss, and the same thing applies for the tools we use. This specific issue is connected to the physics engine inside Unity and it concerns moving objects. As we have already explained, we have created objects with various properties like friction. In one of our scenarios one of the objects, possessing a reduced friction value, is "sliding" away due to a collision. On a follow-up step we eliminate all physical forces before resetting the objects to their initial state. However, for this particular object, there was still an unknown force applied to it, even though we explicitly demanded from Unity to seize all possible forces inside the environment. This "bug" was even impossible to figure out by a more experienced Unity developer we asked for assistance, setting the whole process of development back for more than a month, and finally being solved using extreme methods that slow down the overall performance.

Bibliography

- [1] S. M., “The event calculus explained.” in *Artificial Intelligence Today. Lecture Notes in Computer Science (Lecture Notes in Artificial Intelligence)*. Springer, Berlin, Heidelberg, 1999, p. vol 1600.
- [2] M. Groover, in *Fundamentals of Modern Manufacturing: Materials, Processes, and Systems*, 2014.
- [3] L. Kunze, M. E. Dolha, and M. Beetz, “Logic programming with simulation-based temporal projection for everyday robot object manipulation,” in *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*. IEEE, 2011, pp. 3172–3178.
- [4] E. Erdem, V. Patoglu, and P. Schüller, “A systematic analysis of levels of integration between high-level task planning and low-level feasibility checks,” in *AI Communications*, vol. 29, 03 2016, pp. 319–349.
- [5] E. Erdem, E. Aker, and V. Patoglu, “Answer set programming for collaborative housekeeping robotics: Representation, reasoning, and execution,” in *Intelligent Service Robotics*, vol. 5, 10 2012.
- [6] E. Erdem, K. Haspalamutgil, C. Palaz, V. Patoglu, and T. Uras, “Combining high-level causal reasoning with low-level geometric reasoning and motion planning for robotic manipulation,” in *Proc. of ICRA*, 05 2011, pp. 4575–4581.
- [7] P. Bonatti, F. Calimeri, N. Leone, and F. Ricca, “Answer set programming,” in *A 25-year perspective on logic programming*. Springer-Verlag, 2010, pp. 159–182.
- [8] E. T. Mueller, “Event calculus,” vol. 3. Elsevier, 2008, pp. 671–708.
- [9] A. B. Downey, “Think python: How to think like a computer scientist.” O’Reilly Media, Inc, 2012.
- [10] D. Smith and B. Morgan, “Isisworld: An open source commonsense simulator for ai researchers,” 2010.

- [11] A. Antakli, I. Zinnikus, and M. Klusch, “Asp-driven bdi-planning agents in virtual 3d environments,” in *German Conference on Multiagent System Technologies*. Springer, 2016, pp. 198–214.
- [12] S. Zickler and M. Veloso, “Efficient physics-based planning: sampling search via non-deterministic tactics and skills,” in *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*. International Foundation for Autonomous Agents and Multiagent Systems, 2009, pp. 27–33.
- [13] J. R. Bruce and M. M. Veloso, “Safe multirobot navigation within dynamics constraints,” vol. 94, no. 7. IEEE, 2006, pp. 1398–1411.
- [14] N. A. Melchior, J.-y. Kwak, and R. Simmons, “Particle rrt for path planning in very rough terrain,” in *NASA Science Technology Conference 2007 (NSTC 2007)*. Citeseer, 2007.
- [15] N. Vahrenkamp, C. Scheurer, T. Asfour, J. Kuffner, and R. Dillmann, “Adaptive motion planning for humanoid robots,” in *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*. IEEE, 2008, pp. 2127–2132.
- [16] M. Lau and J. J. Kuffner, “Behavior planning for character animation,” in *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*. ACM, 2005, pp. 271–280.
- [17] K. Yamane, J. J. Kuffner, and J. K. Hodgins, “Synthesizing animations of human manipulation tasks,” in *ACM Transactions on Graphics (TOG)*, vol. 23, no. 3. ACM, 2004, pp. 532–539.
- [18] F. Gravot, S. Cambon, and R. Alami, “asymov: A planner that deals with intricate symbolic and geometric problems,” in *ISRR*, 2003.
- [19] K. Hauser and J.-C. Latombe, “Integrating task and prm motion planning: Dealing with many infeasible motion planning queries,” in *Proc. Bridging the Gap Between Task and Motion Planning, ICAPS Workshop, BTAMP’09*, 01 2009.
- [20] L. Pack Kaelbling and T. Lozano-Perez, “Hierarchical task and motion planning in the now,” in *AAAI Workshop - Technical Report*, 06 2011, pp. 1470 – 1477.
- [21] E. Plaku, “Planning in discrete and continuous spaces: From ltl tasks to robot motions,” in *TAROS*, 2012.
- [22] E. Aker, V. Patoglu, and E. Erdem, “Answer set programming for reasoning with semantic knowledge in collaborative housekeeping robotics,” in *SyRoCo*, 2012.

- [23] O. Caldiran, K. Haspalamutgil, A. Ok, C. Palaz, E. Erdem, and V. Patoglu, “Bridging the gap between high-level reasoning and low-level control,” 09 2009, pp. 342–354.