On the impact of sensors update in declarative AI for videogames

Denise Angilica[1], Giorgio Michele De Giorgio[1] and Giovambattista Ianni[1]

[1]*University of Calabria*

Abstract

Declarative methods such as Answer Set Programming show potential in cutting down development costs in commercial videogames and real-time applications in general. Sensors update is one of the major bottlenecks preventing their adoption in such dynamic domains. In this work we show a new optimized approach for the sensors update cycle deployed in our *ThinkEngine*, a framework in which a tight integration of declarative formalisms within the typical game development workflow is made possible in the context of the Unity game engine. *ThinkEngine* allows to wire declarative AI modules to the game logic and to move the computational load of reasoning tasks outside the main game loop using an hybrid deliberative/reactive architecture. In this paper, we discuss the crucial role of the sensors update cycle in the run-time performance of our framework and then we propose a new, optimized, sensors update workflow. After describing the new approach, we report about performance improvements.

Keywords

Answer Set Programming, Declarative Methods, Game Design, Knowledge Representation and Reasoning, Unity

1. Introduction

The interest of the videogame industry in AI research is not new, both whether we are talking of inductive/machine learning-based techniques or knowledge-based, deductive techniques [1]. In this respect, declarative methods show potential benefits, like enabling the possibility of specifying parts of the game logic in a few lines of high-level statements. Possible applications range from defining the general game logic, to describing non-player characters, programming tactic and/or strategic credible AI behaviors, to expressing path planning desiderata, non-player resource management policies and so on.

Many examples of the usage of declarative languages in the industrial videogame realm exist, starting from the pioneer F.E.A.R. game [2], which used STRIPS-based planning [3]. Other remarkable examples are the games Halo [4] and Black & White [5]. If we look at videogames from the basic research perspective, it must be noted the longstanding interest in using (video)games as a controllable and reproducible setting in which to face open research

Original work.

ASPOCP 2023 : 16th Workshop on Answer Set Programming and Other Computing Paradigms

denise.angilica@unical.it (D. Angilica); gmdg141995@gmail.com (G. M. De Giorgio); ianni@unical.it (G. Ianni)

0000-0002-4069-978X (D. Angilica); 0000-0003-0534-6425 (G. Ianni)

issues: one might cite the GDL [6], VGDL [7] and Ludocore [8] which are languages adopted for declaratively describing General Game Playing [9]. The Planning Domain Definition Language (PDDL) found natural usage in the videogame realm [10, 11, 12]; among its sister languages, we will herein focus particularly on Answer Set Programming (ASP), the known declarative paradigm with a tradition in modeling planning problems, robotics, computational biology as well as other industrial applications [13]. ASP does not come last in its experimental usage in videogames: it has been used to various extents, e.g., for declaratively generating level maps [14] and artificial architectural buildings [15]; it has been used as an alternative for specifying general game playing [16], for defining artificial players [17] in the Angry Birds AI competition [18], and for modelling resource production in real-time strategy games [19], to cite a few. Despite this potential, however, performance and integration shortcomings are still a limit to the widespread adoption of declarative methods in professional videogames: ASP makes no exception in this respect.

Two aspects are of concern: *i)* integration, i.e., the ease of wiring declarative modules with other standard parts of the game implementation, and *ii)* performance in real-time contexts. Concerning the first issue, we build on our recent proposal of *ThinkEngine* [20], a tool working in the known Unity game engine [21], which allows wiring declaratively-programmed *Brains* to videogame implementations.

As for the performance issues, first recall that an ASP solver works by taking in input some declarative specification S, a description D of the input problem at hand and produces output *answer sets*, which encode a description of possible solutions to the input $S \cup D$.

It is known that ASP solvers do not exhibit an acceptable performance for fast-paced and repeated evaluation. With this in mind, we already moved some steps forward by equipping our *ThinkEngine* with the *Incremental-DLV2* solver [22] that allows for faster and incremental answer sets generation in a variety of domains. However, since the game-world is dynamic, there is the need to collect game's information and translate it in logical assertions to be fed in input to the solver: the impact of this job should not be underestimated as it can be as time consuming as answer set generation, if not properly implemented.

In this paper, after briefly presenting the features of the *ThinkEngine* system, we show a new approach for implementing the sensing update cycle based on a new sensor update workflow and we report about experiments on the new *ThinkEngine* version, compared with its older version whose performance was badly affected by the previous sensor update cycle implementation.

2. *ThinkEngine* overview

ThinkEngine [20] is a system allowing to integrate declarative-based reasoning modules in a videogame or any other kind of software developed in Unity [21]. Unity is a reference cross-platform game engine primarily used to develop videogames and simulations for more than 20 different platforms like mobile, computers, and consoles [23].

In Unity the game-world's objects are called `GameObject` (GO) and their behavior is defined by enriching them with `Components` that are scripts encoding specific aspects of GOs. The main elements of *ThinkEngine* are the so called *Brains*. Brains can be attached at will to game characters, they can drive parts of the game logic, and can be used in general for delivering AI

at the tactical or strategic level within the game at hand.

One can have *planner brains* or *reactive brains*, which respectively make different types of decisions: deliberative ones (i.e., *plans*), which, in the terminology of our *ThinkEngine*, are sequences of actions to be executed in a programmable order, or *reactive* decisions which can have an immediate impact on the game scene. Plans work in the spirit of the Goal Oriented Action Planning methodology (GOAP), a popular way of deploying academic planning in the realm of videogames since its introduction in F.E.A.R. [2]. Multiple planner brains can be prioritized to control the same character, thus introducing a form of multiple-behavior programming. Moreover, as game environments are subject to fast changes, one can program the appropriate plan aborting logic.

ThinkEngine has been developed having the integration of declarative ASP modules in mind, but other types of automated reasoning can be in principle wired (e.g., PDDL), by writing the appropriate glue code. ASP specifications are composed of set of rules, hard and soft constraints, by means of which it is possible to express qualitative and quantitative statements. In general, a set of input values F (called *facts*), describing the current state of the world, are fed together with an ASP specification S to a *solver*. Solvers in turn produce sets of outputs $AS(S \cup F)$ called *answer sets*. Answer sets contain the result of a decision-making process expressed in terms of logical assertions. These, depending on the application domain at hand, might encode actions to be made, employee shifts to be scheduled, protein sequences, and so on.

ThinkEngine is integrated in the Unity game engine both at design-time and at run-time. At *design-time*, one can add, wire and program brains in the game editor. A brain can be wired to sensor inputs, and one can decide where a brain acts on the game by connecting *actuators* or defining *plan actions*. Sensor values can be aggregated within a time window according to a policy of choice (like max, min, average, oldest or newest value), while *Triggers* can be defined to program when brain reasoning activities must take place at run-time. Brains can be embedded in reusable objects, called *prefabs* in the Unity terminology. At run-time, most of the game runs in the usual single-threaded game loop [24], while brain reasoning tasks are offloaded to separate threads. An information-passing layer allows communication between brains and the main game loop. In this latter, a sensor update cycle periodically refreshes sensor readings. Inputs (i.e., sensor readings) and outputs (i.e. actuator values or plans) of brains are bidirectionally converted to/from the ASP-Core2 format [25], according to a properly defined mapping discipline between the game world data and ASP logic assertions.

3. The sensors update cycle

Sensors play a crucial role in various domains where the collection of real-time data is essential for decision-making, monitoring, and control. There are several domains where real-time sensor readings are crucial such as healthcare, robotics, gaming, sports, home automation, and more. Sensor technology continues to evolve, enabling new applications and driving advancements in various industries.

The continuous reading of data structures in order to check if values have changed in a time-critical manner poses several challenges and requires avoiding some drawbacks. There are

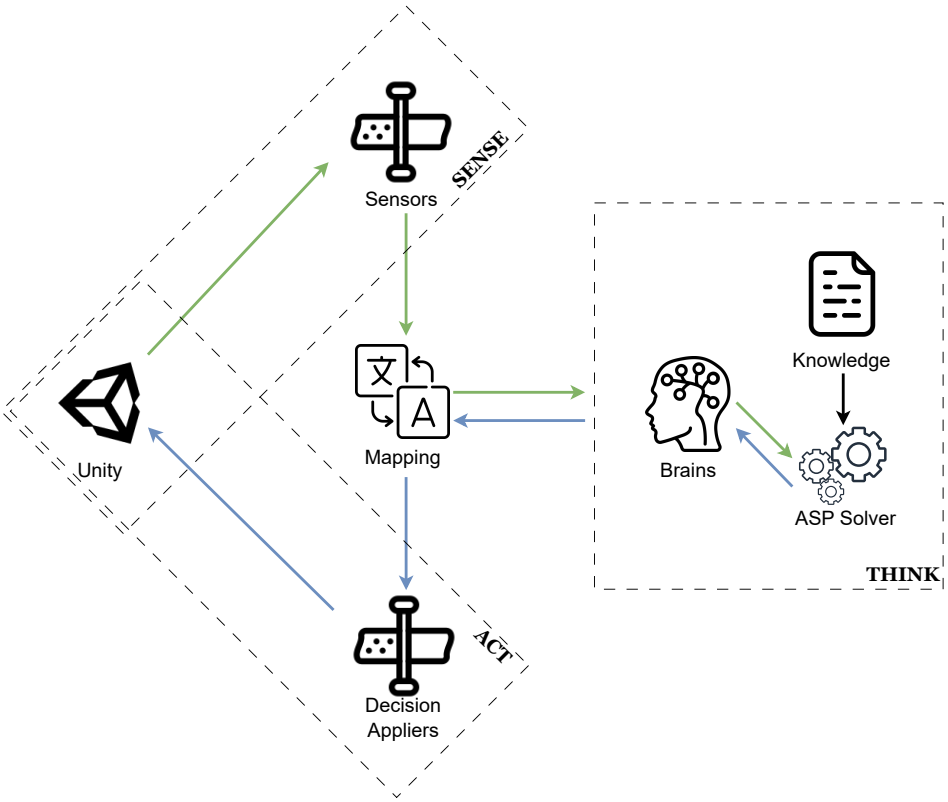

Figure 1: The *ThinkEngine* run-time architecture.

a number of issues associated with the process, especially concerning performance impact and resource utilization.

An informed decision-making reasoning task requires the availability of up-to-date and comprehensive sensor values. In this respect, a fruitful enactment of ASP within a highly dynamic environment cannot ignore this fundamental aspect. Let us assume sensor readings are modeled as a pool of propositional facts F. F should fairly reflect the latest readings from the game scene before being fed as input to the solver of choice.

One can implement the sensors update cycle basically in two distinct ways. Let us assume a sensor collector node CN keeps track of the latest readings F of a set of sensors: *i)* with a *push* approach, sensors notify pro-actively CN of their changes, and CN keeps track of the update; *ii)* in the *pull* approach, CN cyclically reads data from the pool of sensors in order to detect possible changes. Clearly, in the pull approach, CN must have an active role and is subject to a heavier computational load; with the push approach, some computational load is shifted to the remote sensing nodes. The implementation of a push methodology is however not straightforward, and some care must be given in order to avoid that CN is flooded with notifications for even the slightest change: this is especially important when sensors range over continuous values.

It must be however noted that in simulated environments deployed within a single computing

node, the shift of computational load towards remote sensing nodes is mostly fictitious. In the specific case of game engines, one has to recall that the game scene data structures are accessible only within the single-threaded game loop. The Unity game engine does not make an exception in this respect.

Furthermore, although feasible in principle, the push approach would put a further burden on game designers. Indeed, to achieve a push-based update cycle, game designers would have to modify their existing implementation by enriching it with notification glue code attached to all the property values to be sensed. This approach does not fit the spirit of *ThinkEngine* whose goal is to reduce to the bare minimum the effort required to game designers.

We thus opted for a pull-based approach, which led to our old implementation (I_O in the following). I_O made use of a reflection layer that, at run-time, would navigate object properties and retrieve the sensor information to be passed to the reasoning layer [26]. In the next section, we will overview the role of run-time reflection in this process, and we will introduce the new approach I_N where the usage of reflection is moved at design-time, thus reducing the time required to pull data at each iteration.

4. Improving sensors performance of *ThinkEngine*

In previous versions of *ThinkEngine*, the Sensors module consisted mainly of four interacting classes:

- *SensorConfiguration.* This class inherits from `MonoBehaviour`, the class used in Unity for defining Components of GOs. When designers want to observe the properties of a given game object G, they need to add a `SensorConfiguration` to it. Properties of interest in the G can be selected and added to the `SensorConfiguration`. When a property is selected as a sensor, an aggregation function can be chosen, which is applied to the sequence of the last recorded values for the sensor at hand (default is 200). The aggregation function options include maximum, minimum, average, oldest, and newest values.
- *MonoBehaviourSensorsManager.* An instance of this class is automatically added when the designer inserts a `SensorConfiguration` and is meant to handle the instantiation of sensors at run-time. For each `SensorConfiguration` and each property, this class instantiates a single sensor for basic type properties. For complex data structures, such as lists, the number of instantiated sensors matches the number of elements in the collection. In the latter case, it is dynamically checked whether the size of the data structure has changed at run-time. Whenever the data structure increases in size, new sensors are instantiated, while the removal of elements causes the destruction of the homologue unused sensors.
- *Sensor.* A sensor is an object associated with a single property or an element of a data structure. Currently, *ThinkEngine* supports basic types such as integers and booleans, and some data structures such as lists and one-dimensional and two-dimensional arrays.
- *SensorsManager.* A singleton instance of this class performs the sensor update cycle at run-time. This cycle involves the following operations: for a given sensor S attached

on a property P (1) its value is updated (UpdateValue), (2) it is checked whether the corresponding `MonoBehaviourSensorsManager` should instantiate or delete sensors (Manage), and (3) S is serialized to a string value in the format of a logic assertion for the ASP solver (Map).

In the I_O implementation, all the instantiation, management and update steps use reflection techniques [27] to achieve their goals. Recall that in object-oriented programming reflection refers to the ability of a program to examine and modify its own structure and behavior at runtime. In a sense, reflection corresponds to higher-order reasoning and reification in logic. For instance, by means of reflection, one can list and modify properties and methods of a given object at run-time, although this capability comes at the price of slower access to data structures.

In order to give an idea of the actual update process, let *GO* be a game object of interest for the AI module, let *C* be a component of *GO*, let *Obj* be a property of *C* and *list* a property of *Obj* pointing to a list of integers L. At run-time, the property retrieving process would work as described in Algorithm 1. This process must be repeated each time that an instantiation (Algorithm 2) or manage (Algorithm 3) or update (Algorithm 4) operation has to be performed. Indeed, throughout the game, the value of properties can change at any level of the hierarchy. In the three latter algorithms, *SC* is a Sensor Configuration associated with *GO*. Elements in *SC*.PropertyNames point to all the properties relevant for the AI module. Each element is a list of strings, each i-th element corresponding to the name of a property at the i-th level of the hierarchy. The serialization operation of the sensor values, i.e. the final translation in logical assertions, does not need to retrieve the coordinates of the property at hand since all the needed information, at that point, is stored in the sensor.

Algorithm 1 Property Retrieval

Input: {}
Output: Property P
$Comp$ = GO.RetrieveComponentByName(C);
if $Comp$ is not null **then**
    $P1$ = $Comp$.Type().GetPropertyByName(Obj);
    if $P1$ is not null **then**
        P = $P1$.Type().GetPropertyByName($list$);
        return P
    end if
end if

Algorithm 2 Sensors Instantiation

Input: List of property names $SC.PropertyNames$
Output: {}
$Properties$ = SC.PropertyNames
while $Properties$ is not empty **do**
    P = $Properties$.Pop()
    $P1$ = RetrieveProperty(P)
    if $P1$ is not null **then**
        $Mapper$ = GetMapper($P1$.Type())
        $Mapper$.Instantiate($P1$)
    end if
end while

Algorithm 3 Sensors Management

```
1: Input: List of property names SC.PropertyNames
2: Output: {}
3: Properties = SC.PropertyNames
4: while Properties is not empty do
5:     P = Properties.Pop()
6:     P1 = RetrieveProperty(P)
7:     if P1.size > OldSize(P1) then
8:         Mapper = GetMapper(P1.Type())
9:         Mapper.InstantiateNewSensors(P1)
10:    end if
11: end while
```

Algorithm 4 Sensors Update

```
1: Input: List of property names SC.PropertyNames
2: Output: {}
3: Properties = SC.PropertyNames
4: while Properties is not empty do
5:     P = Properties.Pop()
6:     P1 = RetrieveProperty(P)
7:     Mapper = GetMapper(P1.Type())
8:     Mapper.UpdateSensors(P1)
9: end while
```

In order to overcome performance impact issues, in a previous version of *ThinkEngine* [20] we introduced a workload balancing strategy by distributing the whole update cycle (comprising both the management step and the update step for each given sensor) over a number of frames. This strategy is effective in spreading the computational load of the update cycle over longer intervals of time. However, it is implied some aging of older sensor readings and it affects the responsiveness of Brains. These have to wait for a full update cycle before being triggered. The culprit of this loss of performance is mainly due to the heavy usage of reflection at run-time.

We thus opted for shifting the operations of reflection to design-time by providing the ability to generate and compile ad-hoc run-time scripts for each target property. In other words, Algorithm 1 is now executed at design-time and it is used in an iterative algorithm that generates reflection-independent scripts to be executed at run-time. Each of these scripts is tailored to a specific property P so that at run-time all information about P itself is available without navigating the property tree. The impact on the designer's effort is limited to prompting a code generation task via Unity's GUI.

Given a property P, its sensor script generation process is as follows: first, we retrieve via reflection the features of P by navigating the property hierarchy of the GO that P belongs to. In particular, depending on the type T of P, we find the appropriate `IDataMapper`. The code of a specific `IDataMapper` implements the serialization strategy transforming an instance of type T into a set of logical assertions in ASP syntax.

Once this information is collected, we dynamically generate some code to be compiled and used at runtime using a template class that is completed in its parametric parts. The generated sensor class inherits from an abstract class called Sensor and, therefore, implements three

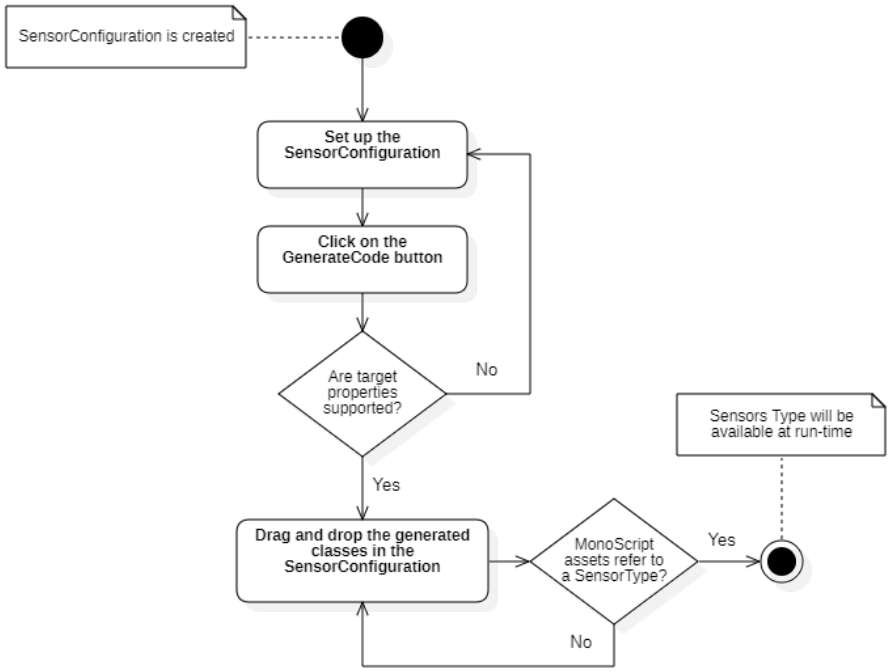

Figure 2: Designer interaction with *ThinkEngine*.

methods:

- The Initialize method is responsible for initializing the sensor's data structures;
- The Update method, called every frame, in the same way as I_O, performs both the UpdateValue and Manage operations. In I_O, the target property was visited twice: once to determine if multiple sensors needed instantiation for complex properties like lists or arrays, and again to observe the actual new value of the property. In I_N, everything is done in a single pass within the Update method of the Sensor object. Additionally, there is no longer a need to instantiate a sensor for data structures. Each sensor, if required by the property type, has data structures that can dynamically accommodate a variable number of information, which can change at run-time;
- The Map method returns a string representing the fact(s) for this sensor to pass as input to the ASP solver.

These methods collectively handle the update and mapping of the sensor's data, providing the necessary functionality for integrating the sensor with *ThinkEngine*.

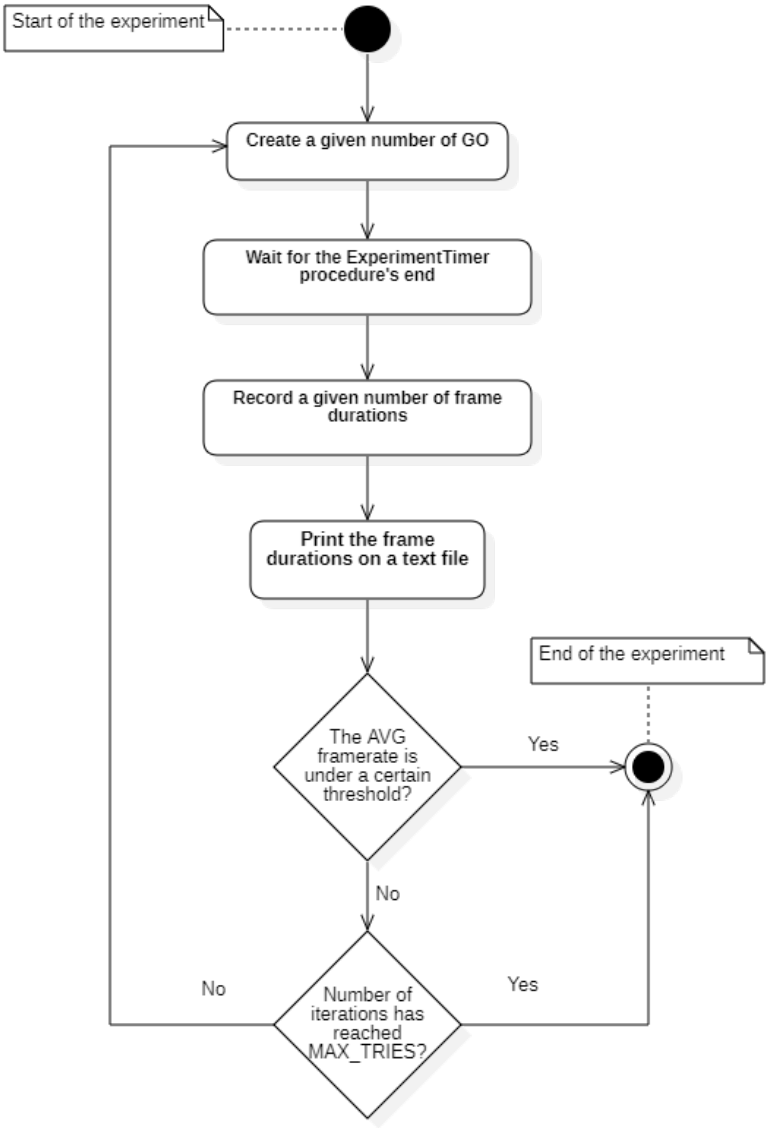

Figure 3: Structure of the test.

5. Experiments

We conducted experiments on I_N and I_O in order to assess performance improvements and identify strengths and weaknesses. Given the particular setting in which *ThinkEngine* works, we devised a proper data collection experimental setting as described in Figure 3.

First of all, we are interested in recording frame durations for a given test that is run in an experimental game scene. Intuitively, the shorter a frame cycle lasts, the higher framerate a

game can achieve. A test is run multiple times: in each run we increased the workload by adding more sensors to the scene. This is repeated for a certain number of iterations or until the frame rate dropped below a specific threshold.

To ensure meaningful measurements and isolate the performance of just the sensor update, we took further measures:

- The computational load of the sensor update cycle should be dominant compared to the rest of the application. We thus avoided to add Brains or any other computationally heavy objects in the scene;
- We started the test in a build of the application rather than through the Unity Editor. This avoids any interference from the Editor's work that could affect the application's performance;
- Since the `SensorsManager` of I_O adaptively manages the workload of the update cycle, we disabled it. I.e. during the experiments, the `SensorsManager` of I_O is forced to perform updates for all active sensors in the same frame.

More in detail, the test procedure is as follows:

- Create a given number of GameObjects, each containing five sensors (i.e. we add a `SensorConfiguration` with five properties of interest);
- Suspend the experiment for a fixed amount of time, in order to avoid inaccurate transient measurements;
- Begin recording the duration of a given number of frames;
- Save all the recorded durations to a text file;
- If the average duration falls below a certain threshold or a certain number of iterations is reached, stop the experiment. Otherwise, repeat all the steps.

The threshold for the average frame rate is set to terminate the experiment when the performance deteriorates too much. Since the number of recorded frames is fixed, a very low frame rate would result in a prohibitively long duration for the experiment. The suspension of the experiment is done to avoid recording a transient performance impact caused by the instantiation of GameObjects and writing to a file.

Three experiments[1] were conducted: *i)* the first one compared I_O and I_N on the same set of parameters; *ii)* the second one aimed to determine how many sensors can be managed before the performance becomes unacceptable with I_N; *iii)* the third experiment focused on identifying the bottlenecks in I_N.

The performance of I_N is significantly better than the previous one (Figure 4). The new version achieves a much higher average of frame per second (FPS) with the same number of sensors, but it also scales much better when the number of sensors is increased. It must be noted that I_O goes below the acceptable threshold of 60 FPS as soon as half a thousand of sensors are added. In the second experiment we kept the setting of the first experiment but we tested I_N

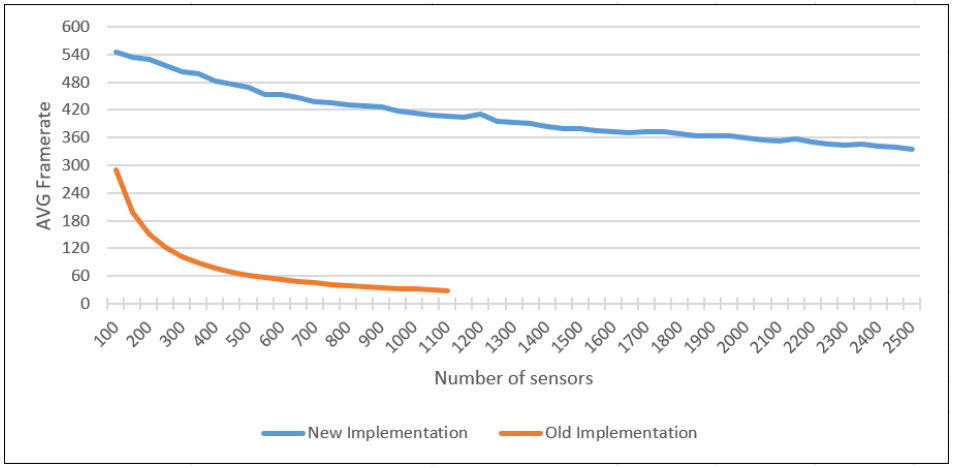

Figure 4: Comparing the two implementations.

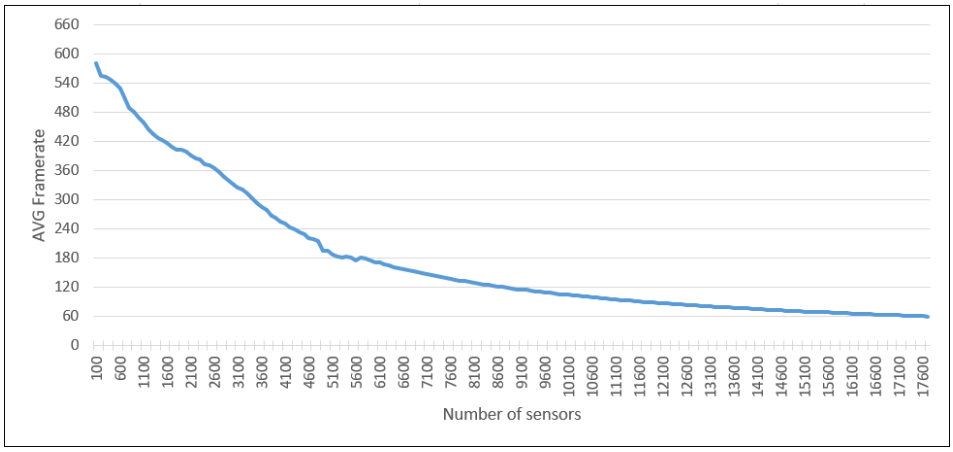

Figure 5: FPS achieved based on the number of sensors in the scene.

only. It was possible to reach several thousand sensors before the test stopped when the lower bound threshold of 60 FPS was reached.

Additional information was sought regarding the behavior of I_N, specifically in order to identify possible remaining bottlenecks in the update cycle. The third experiment was conducted by measuring the computational cost of the Update method of the `SensorsManager`, and comparing it with the additional work required to generate the mapping string. The results indicate that a significant portion of the workload is attributed to the Map operation(Figure 6). The serialization to logical facts made of strings results indeed in a considerable amount of garbage generation, causing the Garbage Collector (GC) to work more frequently and for longer periods, thus causing a faster decaying framerate.

[1]https://github.com/DeMaCS-UNICAL/ThinkEngine-Sensors-Performance-Experiments

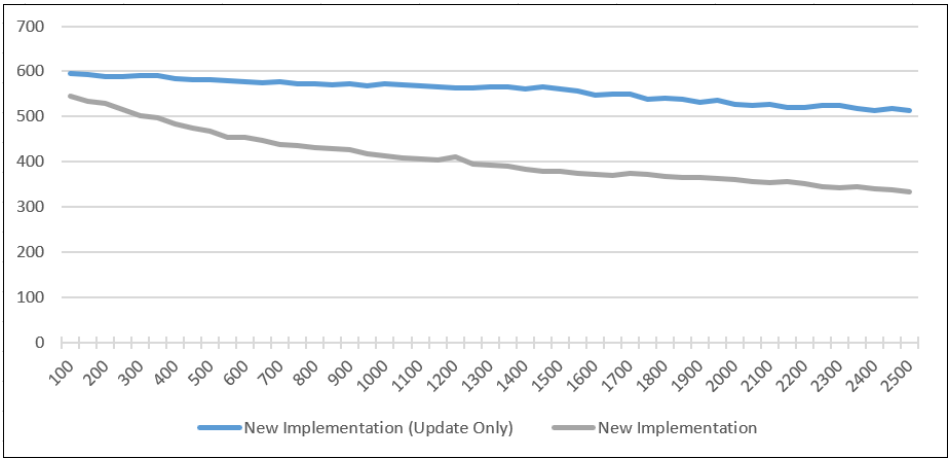

Figure 6: The new bottleneck is the logical assertion generation.

6. Conclusions

While the videogame industry has long recognized the value of using games as controlled environments for addressing open research issues, the integration and performance challenges associated with declarative methods have hindered their widespread adoption. Our contribution addresses these challenges by introducing the *ThinkEngine* system, a tool built on the Unity game engine that facilitates the integration of declaratively-programmed "Brains" into videogame implementations. This system has demonstrated improved performance over previous iterations, overcoming limitations associated with the sensor update cycle.

Our experiments have shown that the new implementation of the sensing update workflow in *ThinkEngine* outperforms its predecessor in all aspects. By leveraging a new approach to collecting and translating game information into logical assertions for the ASP solver, we have significantly reduced the time required for producing action to be executed. This improvement enhances the real-time capabilities of *ThinkEngine*, making it a more practical and efficient solution for incorporating declarative methods into professional videogames.

The results of our study highlight the increasing potential for declarative methods in the videogame industry. Future work can focus on further optimizing performance, expanding the range of declarative functionalities, and exploring additional applications of *ThinkEngine* in different types of videogames.

Acknowledgments

This work was partially supported by: the PNRR MUR project PE0000013-FAIR, Spoke 9 - Green-aware AI – WP9.1; the project “Declarative Reasoning over Streams” (CUP H24I17000080001) – PRIN 2017 and by the LAIA laboratory (part of the SILA laboratory network at University of Calabria).

References

[1] J. Pfau, J. D. Smeddinck, R. Malaka, The case for usable AI: what industry professionals make of academic AI in video games, in: CHI PLAY (Companion), ACM, 2020, pp. 330–334.

[2] J. Orkin, Three states and a plan: the AI of F.E.A.R., in: Game Developers Conference, 2006.

[3] N. Nilsson, STRIPS planning systems, Artificial Intelligence: A New Synthesis (1998) 373–400.

[4] Halo, 2001. URL: https://www.xbox.com/en-US/games/halo.

[5] Black & White, 2001. URL: https://www.ea.com/games/black-and-white.

[6] M. R. Genesereth, N. Love, B. Pell, General game playing: Overview of the AAAI competition, AI Mag. 26 (2005) 62–72.

[7] T. Schaul, A video game description language for model-based or interactive learning, in: CIG, IEEE, 2013, pp. 1–8.

[8] A. M. Smith, M. J. Nelson, M. Mateas, LUDOCORE: A logical game engine for modeling videogames, in: CIG, IEEE, 2010, pp. 91–98.

[9] D. P. Liebana, S. Samothrakis, J. Togelius, T. Schaul, S. M. Lucas, General video game AI: competition, challenges and opportunities, in: AAAI, 2016, pp. 4335–4337.

[10] O. Bartheye, E. Jacopin, A real-time PDDL-based planning component for video games, in: AIIDE, The AAAI Press, 2009.

[11] J. Robertson, R. M. Young, The general mediation engine, Experimental AI in Games: Papers from the 2014 AIIDE Workshop. AAAI Technical Report WS-14-16 10 (2014) 65–66.

[12] J. Robertson, R. M. Young, Automated gameplay generation from declarative world representations, in: AIIDE, AAAI Press, 2015, pp. 72–78.

[13] E. Erdem, M. Gelfond, N. Leone, Applications of answer set programming, AI Mag. 37 (2016) 53–68.

[14] A. M. Smith, M. Mateas, Answer set programming for procedural content generation: A design space approach, IEEE Trans. Comput. Intell. AI Games 3 (2011) 187–200.

[15] L. van Aanholt, R. Bidarra, Declarative procedural generation of architecture with semantic architectural profiles, in: CoG, IEEE, 2020, pp. 351–358.

[16] M. Thielscher, Answer set programming for single-player games in general game playing, in: ICLP, volume 5649 of *LNCS*, Springer, 2009, pp. 327–341.

[17] F. Calimeri, M. Fink, S. Germano, A. Humenberger, G. Ianni, C. Redl, D. Stepanova, A. Tucci, A. Wimmer, Angry-hex: An artificial player for angry birds based on declarative knowledge bases, IEEE Trans. Comput. Intell. AI Games 8 (2016) 128–139.

[18] J. Renz, X. Ge, S. Gould, P. Zhang, The angry birds AI competition, AI Mag. 36 (2015) 85–87.

[19] M. Stanescu, M. Certický, Predicting opponent's production in real-time strategy games with answer set programming, IEEE Trans. Comput. Intell. AI Games 8 (2016) 89–94.

[20] D. Angilica, G. Ianni, F. Pacenza, Declarative AI design in Unity using answer set programming, in: CoG, IEEE, 2022, pp. 417–424.

[21] Unity 3D game engine, Last accessed: Jan 2023. URL: https://unity3d.com/unity.

[22] D. Angilica, G. Ianni, F. Pacenza, J. Zangari, Integrating asp-based incremental reasoning in the videogame development workflow (application paper), in: PADL, volume 13880 of

Lecture Notes in Computer Science, Springer, 2023, pp. 96–106.
[23] F. Messaoudi, G. Simon, A. Ksentini, Dissecting games engines: The case of unity3d, in: NetGames, IEEE, 2015, pp. 1–6.
[24] Unity, order of execution for event functions, Last accessed March 2023. URL: https://docs.unity3d.com/Manual/ExecutionOrder.html.
[25] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca, T. Schaub, ASP-Core-2 input language format, Theory Pract. Log. Program. 20 (2020) 294–309.
[26] D. Angilica, G. Ianni, et al., Tight integration of rule-based tools in game development, in: AI*IA, 2019.
[27] M. H. Ibrahim, Reflection in object-oriented programming, Int. J. Artif. Intell. Tools 1 (1992) 117–136.