5000 Characters at Video Frame Rate using Declarative Programming

Samuel Hill[1], Ian Horswill[1]

[1] *Northwestern University, 2233 Tech Drive, Evanston, IL, 60208, USA*

Abstract

Video games have historically been limited in the number of characters under AI control. This is all the more true for games that use declarative systems which are generally used for turn-based or otherwise non-real-time games. While a declarative system can obviously be used for a real-time simulation, we wanted to test the limits of how far it can be scaled for large numbers of characters. This paper shows how 5000 characters can run simultaneously and in real-time with 60fps performance on a modern laptop using a declarative logic programming language.

Keywords

Declarative programming, Logic programming, Needs-based AI, crowd simulations [1]

1. Introduction

Performance considerations have traditionally restricted the number of simultaneous NPCs, particularly when those characters are driven by symbolic rules or other declarative methods. Many systems that do support this kind of character AI are also turn-based rather than real-time. In this paper, we show that 5000 The Sims-style characters can be run simultaneously on a modern laptop using a version of logic programming.

Although Sims-style 'needs-based AI' is relatively simple, and the rendering of the characters in our demo is extremely simple (particle-like dots on the screen), this still demonstrates that symbolic, declarative techniques can be pushed well beyond what would have been though possible. In this paper, we will discuss the system, and the techniques used to make it so efficient.

2. Related Work

Many AI-based games have used symbolic declarative systems - logic, symbolic rules, or some kind of rule engine - for social reasoning and character control. Façade [1] used a reactive planner, ABL [2], that incorporated a forward-chaining production system. Prom Week [3] also used a forward-chaining production system, Comme Il Faut [4]. CiF's successor the Ensemble Engine [5] is similarly a forward-chaining production system. The Sims 3 [6] used a rule system to script the interactions between situations, personality traits, and actions available to a given character [7]. MKULTRA [8] and City of Gangsters [9] both used logic programming to implement social reasoning [10]. F.E.A.R. makes use of a planner, GOAP, that allows for declarative definitions of various AIs in the GDBEdit database tool [11].

One of the earliest and most influential symbolic systems for games is the Inform 7 language [12], [13] that allows designers to build interactive narrative systems using declarative statements. The Versu simulationist narrative system [14], [15] used a custom logic programming language, Praxis, based on an exotic modal logic called eremic logic (aka exclusion logic) [16]. The Lume system [17] made extensive use of Prolog's definite clause grammars [18], [19] for text generation. Lapeyrade has also used Prolog for better character decision making [20].

All these games and systems use declarative programming for either some component of character control and/or social reasoning. Some are turn-based games like in the Versu platform, others are batch like jobs used in Townlikes (that still need to run fast but not at any particular frame rate), and others are real-time systems like with GOAP in F.E.A.R.. Even when used for real-time character control the number of NPCs is usually relatively low and the portion of the game logic that is declarative are the relations between possible actions and the world state. The code for how a Goal executes is still usually written in an imperative manner [11].

There are also entire games that have been built using declarative languages, such as those built with the video game description language - VGDL [21]. Currently this technique is used to generate 2D tile arcade style games with the help of ASP and evolutionary search [22], [23]. These games are playable by a human and are fully declarative - everything from character controls to game logic to the tilemap itself are declared in text files that are

11th Experimental Artificial Intelligence in Games Workshop, November 19, 2024, Lexington, Kentucky, USA.

used to generate a game. The purpose of these games is largely to allow for testing of general video game playing (GVGP) algorithms on several different games [24].

In large part because of the expense of AI simulation, games involving large-scale character simulation are relatively rare. The best known is Dwarf Fortress [25], which supports real-time simulation of small hundreds of characters. Achieving this level of performance requires implementation in C++ and significant programmer effort to optimize cache locality and minimize pointer chasing. RimWorld is a very similar game that also involves social simulation for the purpose of storytelling [26]. Caves Of Qud is a game that is more like adventure mode in DF and, similarly, features hundreds of characters that all have various social relations and need to choose what action to take each tick [27]. Unlike DF Fortress mode and RimWorld, this rogue-like experience is turn based and as such doesn't need to run at a particular frame rate. While all these games feature large numbers of characters acting in a world, none use declarative systems for the control of these NPCs.

Some large-scale crowd simulation exist for use in film and other rendered media forms [28], [29]. Some of these tools even allow the behaviors of the agents to be stated in sentence form for specific decision nodes [30]. These tools target rendered scenes for film and television and as such do not need to run in real-time.

Some crowd simulation techniques have been used to study crowd dynamics and the creation of autonomous agents for visualization [31], [32]. Shao have claimed real-time performance at 30fps when animating up to 1400 characters moving and pathfinding around a 3D scene on 2007 hardware [31]. This was done with over 50,000 lines of C++ and uses a simple action selection method reminiscent of needs-based AI.

3. Needs-based AI

Needs-based AI is a famous technique for controlling characters in video games that was created for The Sims [33], [34], and has been used in other games like Roller Coaster Kingdom and an RPG for Lord Of the Rings [34].

Needs-based AI attempts to fulfill various needs for NPCs by searching over the actions of each object available to them (i.e. advertisements) and weighting the cost of completing the action against each character's needs with some scoring function [34]. Each character goes about completing actions that have been selected/assigned to them, potentially getting rewarded for the various tasks. When no more actions remain in the queue of selected actions the NPC searches for all the advertisements on all the objects around them then scores each action based on their needs and selects the best scoring advertisement. The equation for selecting the best scoring advertisement is $argmax(S)$ where S is the set of all scored advertisements for a given character. The complexity of scoring all advertisements is $O(NLM)$, where N is the number of people, L is the objects nearby, and M is the advertisements at each location. Once you have scored all advertisements for all characters, you can run the argmax equation for each character. Since argmax is operating on the set of all advertisements for each person, the complexity of selecting an action from the scored set is $O(NLM)$, where N is the number of people, L is the objects nearby, and M is the advertisements. Given that these two steps are each of the same complexity, and that the two iterations are sequential, the total complexity of action selection is $O(NLM)$.

The selected advertised action sequence is then added to the queue of actions to be completed. Needs-based AI has always had the ability to sequence actions, a technique called action chaining [34], but in The Sims 4 the developers wanted to be able to have more interesting action sequences. To accomplish this the developers allowed the system to simultaneously dispatch multiple actions from the queue when those actions would not interfere with one another [35].

The Sims 3 had a goal of creating a larger varied living world, and to do this while remaining playable several optimizations were employed [36]. Some execution optimizations such as hierarchical planning and commodity-interaction maps as well as some level of detail (LoD) optimizations such as auto-satisfy curves and story progression [36]. Hierarchical planning takes the usually $O(NLM)$ approach of searching for every person (N), for every location (L), and then for every advertised action (M), and makes it $O(N + L + M)$ by instead selecting each step one at a time. Commodity-interaction maps are essentially dictionaries of needs and the actions that satisfy those needs, and they are used to trade off compute time searching over all advertisements on all objects to see if they satisfy a sim's needs with storage cost. The LoD optimizations tick the actions of sims not in the current focused level when they come into focus (auto-satisfy), and tick certain subsystems like the wants of the town itself at a lower rate (story progression). As a result, although the simulation notionally supports hundreds of characters, the vast majority of them are effectively idle at any given time.

Figure 1: State Fair Sim running in Unity

As well, although The Sims 3 has many more possible motives (needs) than the 8 used in previous versions, any individual character has a small number of those possible motives.

3.1. Scoring functions

The scoring function that we use here is the most sophisticated function listed in Zubeks description of needs-based AI, an attenuated need-delta scoring function that is additionally attenuated based on distance [34]. One difference in our version is that each action can only satisfy one need. In the version of action performance outlined by Zubek, some actions can be sequences, and this can potentially allow for multiple needs to be fulfilled by an advertisement. As such, the algorithm for calculating an advertised action score must sum the attenuated scoring across all needs. Instead of summing across all needs, with each action only satisfying one need, we can simplify the attenuated need-delta scoring with distance attenuation to:

$$score = D(A_{need}(v_{current}) - A_{need}(v_{current} + \delta))$$

where $v_{current}$ is the current need value of a particular need type, δ advertised need delta, A_{need} is an attenuation function, and D is the distance attenuation function. $A_{need}(x) = \frac{10}{x}$ as is discussed in Zubek, but instead of the mentioned $D(x) = \frac{x}{distance^2}$ function ours is $D(x) = \frac{x}{1+\left(\frac{distance^2}{2870}\right)}$ to tune down the degree to which NPCs attenuate distance. The value 2870 was not chosen at random, it relates to the size of the map that the characters move on and effectively scales $distance^2$ to be between 1 and 30 (as opposed to a max of 82369).

One more simplification in the current demo is that all locations only advertise a single action. In effect, a location has a type that correlates to a need. As such, the complexity of this scoring algorithm can be given as $O(NL)$, where N is the number of NPCs that need to select actions and L is the number of locations to search over.

4. Real-time logic programming

To test the scaling limits of declarative symbolic logical methods for real-time NPC control, we built a state-fair simulator, as one might find in a tycoon game, using a logic-programming implementation of needs-based AI. NPC's wander through the state fair eating, drinking, seeking entertainments, and so on. The simulation consists of a map with 367 locations, onto which we spawn a few thousand NPCs. Each character has 6 motivations (hunger, bathroom, games, amusement, music, and shopping) that influence their action selection.

4.1. Needs-based AI in logic programming

The logic for the core steps of scoring all available actions and then selecting the best action for each

character is contained in the few lines of code found in Code Fragment 1 and 2. For a discussion of the programming language used, see [37], [38], but the following fragments can be explained thusly:

Code Fragment 1 defines a predicate that is true when a given person performing a given advertised action at a given location has a given score. It states that it has that score when the location advertises that action, it fulfills some need, and score is the result of distance-based scoring, above, along with some noise to add a small amount of randomness to the action selection:

Code Fragment 1
Action Scoring

```
PersonActionScoring = Predicate("…",
    person.Indexed, actionType.Indexed,
    location.Indexed, score).If(
        ReadyToSelectAction,
        CoordForPerson[person, coords],
        LocationAdvertisements,
        deltaOffset == delta +
                       RandomFloat[-2, 2],
        ActionSatisfiesNeed,
        FairgroundLocations[location, __,
                            otherCoords],
        NeedByType[person, needType, need],
        DistanceScoring[need, deltaOffset,
                        coords, otherCoords,
                        score]);
```

Given this, Code Fragment 2 defines a predicate stating a given action/location pair is best for a given character (note that this is essentially the same as the argmax equation, just written in logic):

Code Fragment 2
Action Selection

```
PersonActionBestScore = Predicate("…",
    person.Key, actionType.Indexed,
    location.Indexed).If(
        ReadyToSelectAction[tempPerson],
        Maximal((person, actionType,
                 location), score,
        PersonActionScoring[tempPerson,
            actionType, location, score] &
        person == tempPerson));
```

4.2. Implementation optimizations

The performance of the system is dependent on several optimizations. Many of these have to do with the specific programming language implementation: strong typing, bottom-up execution, parallel execution, native code compilation. The details of the language implementation are outside the scope of this paper but see [37], [38] for more information.

Another important optimization might be called implicit need decay. Normally, a needs-based AI system would update each need state of each character on every tick. For our system, this would be prohibitively expensive. Instead, we store timestamped need values; a need value is represented as a tuple of (v, t, ρ) stating that the need had value v at time t, and was decaying at rate ρ. We can then compute the current need value on demand as:

$$v_{current} = \max(v - (t - t_{now}) * \rho)$$

with t_{now} being the current time. Since these values are read infrequently but would otherwise be updated frequently, this is much more efficient. As is recommended by Zubek, decay rates for a given need vary slightly from character to character to give them slightly different "personalities". These decay rates are chosen randomly (Appendix A, Code Fragment 3).

4.3. Action States

For characters to perform a selected action they must move to the location of the selected action and then take some steps to complete the action before the need is satisfied. While we could allow for completion of actions instantaneously, to be more in line with The Sims implementation - as specified by Zubek – we instead have some action performance before a need is satisfied.

When an action is initiated, the system tells the unity code to start moving the character, and then marks the character as being in progress towards that location. When the character arrives at the location they are marked as no longer moving towards their destination. When the action is completed, unity informs the system, and the system updates the need states, and generates a new next action (Appendix A, Code Fragment 6, 7, 8, 11). In place of the animations that would normally be running when performing an action, we simulate an animation running by starting a timer that indicates the selected action is being done (Appendix A, Code Fragment 9).

Unlike in The Sims all actions in this simulation are individual, there is no chaining of actions nor are there any concurrent interactions. This simplification is what allows us to use a timer for each person taking an action, when the timer is running the "animation" is playing and when the timer completes the character

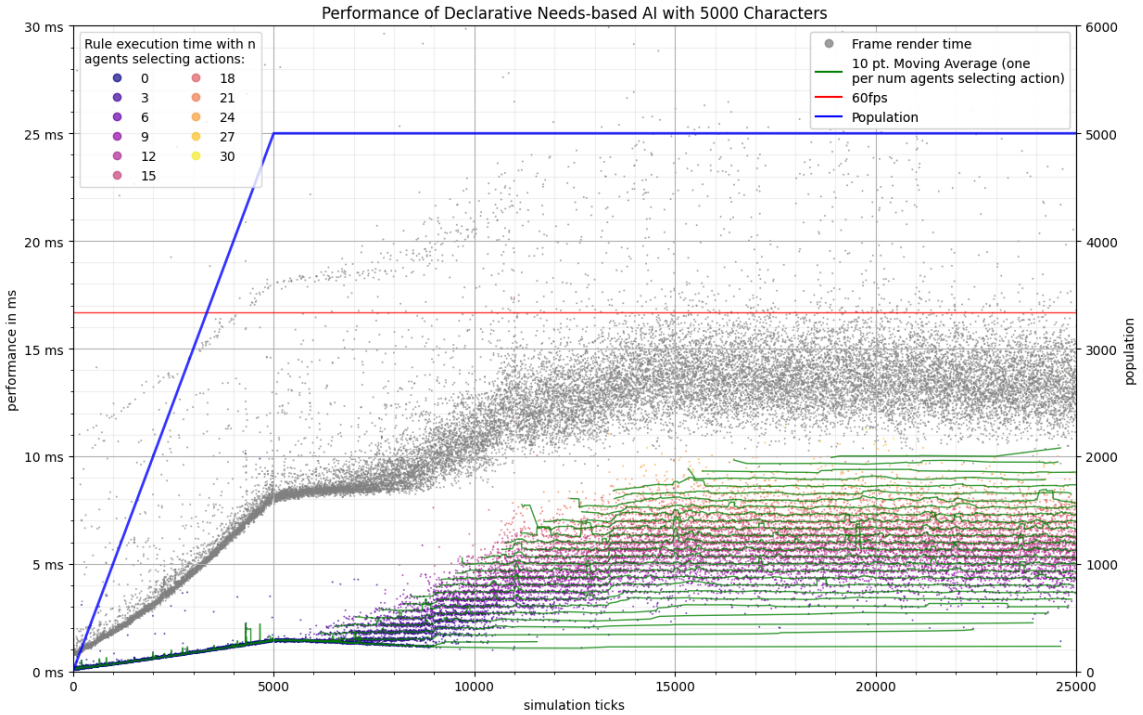

Figure 2: Performance of the State Fair Sim

has completed the action and finished its animation. While we do not have proper animations for each sim, when on a timer for an action the sim is moving randomly around the location that they have selected an action at.

4.4. Destinations & Unity Interface

For control of character movements, we have a simple interface between the action selection simulation and the Unity code that performs movements. The Simulog code provides a table of people and the destination they are moving to (Appendix A, Code Fragment 7) that the Unity code iterates over each tick to adjust NPC positions. The Unity code in turn provides a function that we can call to ask if someone has arrived at their destination. With this simple interface of action selection assigning destinations to the Unity code and Unity only needing to report back when someone has arrived at the destination, we are able to change out the underlying movement implementation without needing to do any modifications to the action selection algorithm. Currently, the movement is all simple vector addition moving sims at a set speed straight towards their destination.

5. Performance

With 5000 NPCs running this simulation on an Apple M3 Pro we are able to achieve 60fps or better for 95% of frames when running some performance tests. Our average inner-frame time was 13.87ms with a standard deviation of 2.077ms. Figure 3 shows our performance curve once stable (from 12500 to 25000 ticks) which is somewhat bell curve shaped with a longer tail to the right. Most of the tail to the right is likely garbage collection or some other Unity process as that performance tail is not present in Figure 4 when we show the performance curve for just the rule execution times.

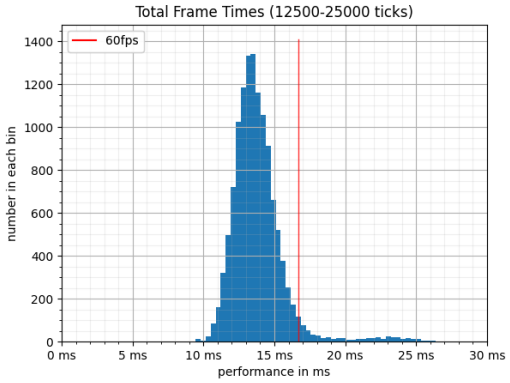

Figure 3: State Fair Sim Inter-Frame Intervals

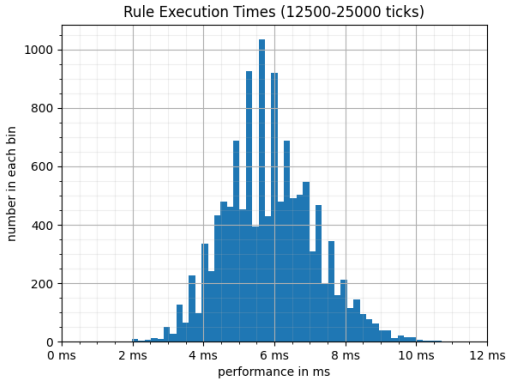

Figure 4: Action Selection Rule Execution Times

The graph in Figure 2 shows the progression of the simulation over a 25,000-tick run time. The first 5000 ticks spawn one person every tick causing the slow rise in compute time for both the action selection mechanism and the inner-frame updates. For the next 7500 ticks characters begin to arrive at their first destination and need to select their next action, creating a tiered slope in the action selection performance as various numbers of characters need to select an action. From 12500 ticks on we have a steady state where performance is consistent and linear with NPCs now moving around and selecting actions at a rate of 13.7 actions selected on average per tick for our sample with a σ of 3.8 actions.

The cost of moving 5000 NPCs and rendering the scene (amongst other functions that Unity runs) is the difference between the action selection rule execution time and the inner-frame time. As can be seen in Figure 1, this difference (looking from 5000 ticks to 9000 ticks where action selection has not yet taken off) is about 7ms. There is also a baseline cost to running the rest of the declarative simulation when not performing action selection. As can be seen in Figures 2 and 4, the cost of running our declarative simulation on 5000 characters without any action selection takes about 1ms.

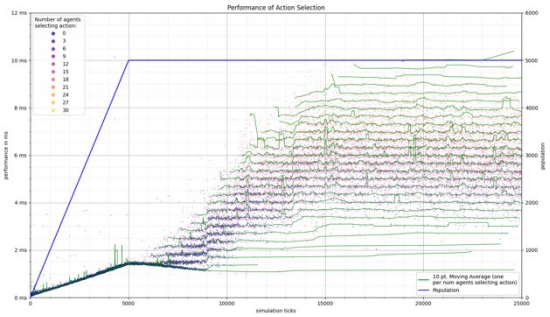

Figure 5: Performance of Action Selection

After these baseline costs, the variation in performance largely comes down to the number of characters selecting actions on any given tick. The performance cost for one NPC selecting an action averages to just under 0.3ms. This gap between each number of agents selecting is clearly visible in Figure 2 and 5 with both the color mapped data points and green lines indicating the moving average for each tier. The same gaps are not visible in the inter-frame time data (grey dots in Figure 2), although there is clearly a correlation between the range of performance for rule execution and for total inter-frame time.

5.1. Pushing character counts

The performance shown above is only possible when we first compile our rules down to native C# code, the process for which is described in [37], [38], and then we run our compiled code in parallel. The type of parallelism employed is per predicate, not per sim, utilizing the dependency graph for all predicates to spin up tasks that run the compiled rules for a given predicate only when its dependencies have finished. Without these optimizations the maximum character count at nearly the same performance is 2000 NPCs.

One optimization that could be employed to further stabilize performance (and potentially allow for more NPCs) is to throttle the number of characters that are selecting actions on a given tick. With a limit on the number of characters selecting actions that is slightly higher than the un-limited average number selecting you could effectively cap the cost of doing action selection while still eventually performing the same sets of needs-based calculations. This technique was not employed in this demo but a commercial system that cares more about never dropping below 60fps may want to implement such a throttle.

6. Conclusion

AI character control based on declarative programming can be surprisingly performant. Through a combination of compilation, parallel evaluation, and careful optimization, thousands of characters can be run at video frame rate. Although this demonstration involves simple needs-based AI, we intend to investigate more complicated techniques in the future.

References

[1] M. Mateas and A. Stern, *Façade*. (2005).

[2] M. Mateas and A. Stern, "A behavior language for story-based believable agents," *IEEE Intell. Syst.*, vol. 17, no. 4, pp. 39–47, Jul. 2002, doi: 10.1109/MIS.2002.1024751.

[3] J. McCoy, M. Treanor, B. Samuel, A. A. Reed, N. Wardrip-Fruin, and M. Mateas, "Prom week," in *Proceedings of the International Conference on the Foundations of Digital Games*, in FDG '12. New York, NY, USA: Association for Computing Machinery, May 2012, pp. 235–237. doi: 10.1145/2282338.2282384.

[4] J. McCoy, M. Treanor, B. Samuel, N. Wardrip-Fruin, and M. Mateas, "Comme il Faut: A System for Authoring Playable Social Models," *Proc. AAAI Conf. Artif. Intell. Interact. Digit. Entertain.*, vol. 7, no. 1, pp. 158–163, Oct. 2011, doi: 10.1609/aiide.v7i1.12454.

[5] B. Samuel, A. A. Reed, P. Maddaloni, M. Mateas, and N. Wardrip-Fruin, "The Ensemble Engine: Next-Generation Social Physics".

[6] *The Sims 3*. (2009). Maxis.

[7] R. Evans, "AI challenges in Sims 3," *Artif. Intell. Interact. Digit. Entertain.*, 2009.

[8] I. Horswill, "Postmortem: MKULTRA, An Experimental AI-Based Game," *Proc. AAAI Conf. Artif. Intell. Interact. Digit. Entertain.*, vol. 14, no. 1, Art. no. 1, Sep. 2018, doi: 10.1609/aiide.v14i1.13027.

[9] *City of Gangsters*. (2021). SomaSim, Chicago.

[10] R. Zubek, I. Horswill, E. Robison, and M. Viglione, "Social Modeling via Logic Programming in City of Gangsters," *Proc. AAAI Conf. Artif. Intell. Interact. Digit. Entertain.*, vol. 17, no. 1, pp. 220–226, Oct. 2021, doi: 10.1609/aiide.v17i1.18912.

[11] J. Orkin, "Three States and a Plan: The A.I. of F.E.A.R.," 2006.

[12] G. Nelson, *Inform 7*. (2006).

[13] G. Nelson, "NATURAL LANGUAGE, SEMANTIC ANALYSIS AND INTERACTIVE FICTION," 2006.

[14] R. Evans and E. Short, "Versu—A Simulationist Storytelling System," *IEEE Trans. Comput. Intell. AI Games*, vol. 6, no. 2, pp. 113–130, Jun. 2014, doi: 10.1109/TCIAIG.2013.2287297.

[15] R. P. Evans and E. Short, "The AI Architecture of Versu".

[16] R. Evans, "Introducing Exclusion Logic as a Deontic Logic," in *Deontic Logic in Computer Science*, vol. 6181, G. Governatori and G. Sartor, Eds., in Lecture Notes in Computer Science, vol. 6181. , Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 179–195. doi: 10.1007/978-3-642-14183-6_14.

[17] S. Mason, C. Stagg, and N. Wardrip-Fruin, "Lume: a system for procedural story generation," in *Proceedings of the 14th International Conference on the Foundations of Digital Games*, San Luis Obispo California USA: ACM, Aug. 2019, pp. 1–9. doi: 10.1145/3337722.3337759.

[18] F. C. N. Pereira and D. H. D. Warren, "Definite clause grammars for language analysis—A survey of the formalism and a comparison with augmented transition networks," *Artif. Intell.*, vol. 13, no. 3, pp. 231–278, May 1980, doi: 10.1016/0004-3702(80)90003-X.

[19] F. Pereira and S. M. Shieber, "Prolog and Natural-Language Analysis," 1987. [Online]. Available: https://api.semanticscholar.org/CorpusID:264203475

[20] S. Lapeyrade, "Reasoning with Ontologies for Non-player Character's Decision-Making in Games," *Proc. AAAI Conf. Artif. Intell. Interact. Digit. Entertain.*, vol. 18, no. 1, Art. no. 1, Oct. 2022, doi: 10.1609/aiide.v18i1.21980.

[21] T. Schaul, "A video game description language for model-based or interactive learning," in *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, Niagara Falls, ON, Canada: IEEE, Aug. 2013, pp. 1–8. doi: 10.1109/CIG.2013.6633610.

[22] T. S. Nielsen, G. A. B. Barros, J. Togelius, and M. J. Nelson, "Towards generating arcade game rules with VGDL," in *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, Tainan: IEEE, Aug. 2015, pp. 185–192. doi: 10.1109/CIG.2015.7317941.

[23] M. J. Nelson and M. Mateas, "Towards Automated Game Design," in *AI*IA 2007: Artificial Intelligence and Human-Oriented Computing*, vol. 4733, R. Basili and M. T. Pazienza, Eds., in Lecture Notes in Computer Science, vol. 4733. , Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 626–637. doi: 10.1007/978-3-540-74782-6_54.

[24] J. Levine *et al.*, "General Video Game Playing," Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2013, p. 7 pages, 336517 bytes. doi: 10.4230/DFU.VOL6.12191.77.

[25] T. Adams and Z. Adams, *Slaves to Armok: God of Blood Chapter II: Dwarf Fortress*. (2006). Bay 12 Games.

[26] T. Sylvester, *RimWorld*. (Oct. 2018). Ludeon Studios.

[27] *Caves of Qud*. (Jul. 15, 2015). Freehold Games, South Bend, IN and Berkeley, CA.
[28] "MIARMY | Home." Accessed: Aug. 29, 2024. [Online]. Available: https://www.miarmy.com/#/
[29] "Massive Software." Accessed: Aug. 29, 2024. [Online]. Available: https://www.massivesoftware.com/
[30] "Defuzz Process - Feishu Docs." Accessed: Aug. 29, 2024. [Online]. Available: https://basefount.feishu.cn/wiki/GhgswVsrfidfOVkmFcgc1XoYnKA
[31] W. Shao and D. Terzopoulos, "Autonomous pedestrians," *Graph. Models*, vol. 69, no. 5–6, pp. 246–274, Sep. 2007, doi: 10.1016/j.gmod.2007.09.001.
[32] Q. Yu and D. Terzopoulos, "A Decision Network Framework for the Behavioral Animation of Virtual Humans".
[33] W. Wright, *The Sims*. (2000).
[34] R. Zubek, "Needs-based AI," in *Game Programming Gems 8*, A. Lake., Cengage Learning, Florence, KY, 2010.
[35] "Concurrent Interactions in The Sims 4." Accessed: Aug. 30, 2024. [Online]. Available: https://gdcvault.com/play/1020190/Concurrent-Interactions-in-The-Sims
[36] "Modeling Individual Personalities in The Sims 3." Accessed: Aug. 29, 2024. [Online]. Available: https://www.gdcvault.com/play/1012450/Modeling-Individual-Personalities-in-The
[37] I. Horswill and S. Hill, "Fast, Declarative, Character Simulation Using Bottom-Up Logic Programming," presented at the AIIDE Workshop on Experimental Artificial Intelligence in Games, University of Utah, Utah, USA, Oct. 2023.
[38] I. Horswill and S. Hill, "Fast, Declarative, Character Simulation Using Bottom-Up Logic Programming," in *AIIDE 24*,

A. Code Appendix

Other than the two code fragments inside the body of this document, the remain fragments in this appendix contain the entirety of the Simulation code written in TED/Simulog.

Code Fragment 1
NPCs (people) and their Needs

```
People = Exists("People", person);
NeedByType = Predicate("NeedByType",
    person.JointKey, needType.JointKey, need);
```

Code Fragment 2
NPC start and end conditions

```
Unsatisfied = Definition("Unsatisfied",
    person).Is(NeedByType[person, __, need],
                NeedValue[need] == Zero);
People.StartWhen(People.Population[count],
                  count < 5000, RandomPerson)
      .EndWhen(People[person],
                Unsatisfied[person]);
```

Code Fragment 3
Need assignments

```
foreach (var needOfType in NeedTypeList)
    People.StartCauses(Add(NeedByType[person,
                        needOfType, need]).If(
        NewNeed[RandomFloat[0.002f, 0.007f],
                need]));
```

Code Fragment 4
Locations and their Advertisements

```
FairgroundLocations = Parse(
    "FairgroundLocations", location.Key,
    locationType.Indexed, coords.Indexed,
    ParseFairgroundLocations());
LocationAdvertisements = Parse(
    "LocationAdvertisements",
    location.Indexed, actionType.Indexed,
    delta, ParseLocationAdvertisements());
```

Code Fragment 5
Action satisfies need table

```
ActionSatisfiesNeed = Parse("ActionSatisfi…",
    actionType, needType,
    DefaultActionSatisfiesNeed.Select(
        pair => (pair.Key, pair.Value)));
```

Code Fragment 6
Person action state table

```
PersonActionAt = Predicate("PersonActionAt",
    person.Key, actionType.Indexed,
    location.Indexed, moving.Indexed);
PersonActionAt.Overwrite = true;
PersonMovingTo = Definition("PersonMovingTo",
    person, location).Is(PersonActionAt[
        person, __, location, true]);
```

Code Fragment 7
Destinations and Arrivals

```
Destinations = Predicate("Destinations",
                          person, coords)
    .If(People[person], PersonMovingTo[person,
        location], FairgroundLocations);
ArrivedAtDestination = Predicate("…", person)
    .If(People[person], PersonMovingTo[person,
        __], HasPersonArrived[person]);
```

Code Fragment 8
Arrival state update

```
PersonActionAt.Set(person, moving, false)
    .If(ArrivedAtDestination);
```

Code Fragment 9
Action timer

```
ActionTimer = Timer("Action", person);
ActionTimer.StartWhen(ArrivedAtDestination,
    PersonActionAt,
    ActionTypeInteractionTime[actionType,
                              count]);
```

Code Fragment 10
Ready to select action

```
ReadyToSelectAction = Event("…", person)
    .OccursWhen(People[person],
                !People.Start[person],
                !PersonMovingTo[person, __],
                ActionTimer.NotOnTimer);
```

Code Fragment 11
Action assignment

```
PersonActionAt.Add[person, actionType,
    location, true].If(PersonActionBestScore);
```

Code Fragment 12
Action completion and reward

```
CompletedAction = Definition("…", person,
    needType, delta).Is(
        ActionTimer.TimerFinished,
        PersonActionAt,
        ActionSatisfiesNeed[actionType,
            needType],
        LocationAdvertisements);
NeedByType.Set((person, needType), need).If(
    CompletedAction[person, needType, delta],
    NeedByType[person, needType, needCol],
    UpdateNeed[needCol, delta, need]);
```