34

1000 NPCs at 60 FPS

Robert Zubek

34.1 Introduction

In this chapter we look at the AI used to implement characters in the game *Project Highrise* by SomaSim, a skyscraper construction management sim where players build, lease out, and manage a series of highrise buildings.

The AI goal in this simulation game was to implement a "living building," simulating the everyday lives of hundreds of inhabitants of the player's skyscraper, their daily lives and routines. As described in more detail in the next section, we gave ourselves a benchmark goal to hit: In order to meet gameplay needs, up to 1000 NPCs should be able to live in the player's building simultaneously, without dropping below 60 FPS on commodity hardware.

This chapter will describe how we approached the AI implementation that achieves this goal. We will first look at the game itself to illustrate the motivations and constraints behind the AI problem, then in the subsequent section, we will describe two action selection mechanisms we implemented (and why we settled on using just one of them), and following that, a performant system for actually executing these actions.

34.1.1 About the Game

Let us quickly introduce the game itself, as shown in Figure 34.1. The player's job in *Project Highrise* is to invest in the construction of highrise buildings, and then manage them successfully: Get tenants to move in, keep them happy, make sure everybody gets what

Figure 34.1

Screenshot from early game in *Project Highrise*.

they need, and pays rent on time. As in management simulation games of this type, the game board is populated by an ever-increasing variety and number of units, such as offices and restaurants renting space in the building, and characters going in and out of those units, going to work, getting lunch, coming home at night, and so on.

We will not go into further details on the gameplay or the economic simulation in the game, since they are beyond the scope of this chapter, except to point out that they all introduced a common goal: We needed the building to feel alive, to be filled with computer characters whose daily comings and goings would fill the building with irresistible bustle. In addition to the aesthetic feel of that, the NPCs' economic activity drives the economy of the building, which provides the main challenge for the player, so we needed the NPCs to be simulated on an ongoing basis instead of being simply instanced into view and then culled.

We gave ourselves a concrete performance goal: The game needed to be able to simulate and display 1000 NPCs, running at 60 FPS on a reasonably recent desktop-grade personal computer. Furthermore, there was a resource challenge: We were a tiny team, and we knew that during most of the development, we would only have one developer on staff whose job would involve not just AI, but also building *the entire rest of the game* as well. So we needed an AI system that was very fast to build, required very little ongoing maintenance once built—and primarily, helped us reach our performance goal.

Early on we decided to keep things very simple, and to split our character AI into two parts, with *action selection* driving decision-making, and separate *action performance module* acting on those orders.

34.2 Action Selection

Action selection is a fundamental decision process in character AI: What should I be doing at the given point in time? During the game's development, we actually tried two different implementations of action selection: first, a propositional planner, and second, a much simpler time-based scheduler of daily routine scripts.

The goal was to reproduce everyday human behavior: Office workers coming in to work in the morning, maybe taking a lunch break, and working at their desk most of the time until it is time to go home; apartment dwellers coming home in the evening and puttering about until bedtime; maintenance and janitorial crews doing their nightly work, whereas everybody else is sound asleep, and so on.

We did not have to worry too much about animation fidelity as the characters were just 2D sprites. Rather, our focus was on overall character behavior, because the actions of your residents and workers directly drive the in-game economy. For example, your food court restaurants need those office workers taking their lunch breaks so they can turn a profit, and you in turn depend on those restaurants paying rent, so you can recoup your investment. Character behavior is central to the economy core loop.

34.2.1 First System: A Propositional Planner

The planning system was our first stab at action selection for NPCs. This happened early in preproduction, and at that point we did not yet know how much "smarts" we would need or want from our characters, but we knew flexibility was crucial. Since the game simulates people with fairly simple goals, it made sense to use a planner to try to string together sequences of actions to achieve them.

We had some concerns about the runtime performance of a planner vis-à-vis the performance goal, so we decided to implement it using a *propositional planner,* due to its potential for very efficient implementation. A detailed description of such a planner is beyond the scope of this short chapter, but we can describe it briefly at a high level. By a propositional planner, we mean one whose pre- and postconditions come from a finite set of grounded propositions, instead of being expressed as predicates. For example, a planning rule in a propositional planner might look like this (using "~" for negation):

```
Rule: Preconditions:  at-home & is-hungry
      Action:         go-to-restaurant
      Postconditions: ~at-home & at-restaurant
```

Compare this with a rule that uses predicate logic, which is more common in planners descended from STRIPS (Fikes and Nilsson 1971):

```
Rule: Preconditions:  (at-location X) &
                      (desired-location Y) & ~(equal X Y)
      Action:         (go-to Y)
      Postconditions: (at-location Y) & ~(at-location X)
```

Predicate rules are more compact than propositional ones—they let us describe relationships between entire families of entities.* At the same time, this expressiveness is

* Of course one can also convert predicate rules into propositional rules by doing Cartesian products of the predicates and all the possible values for their free variables, at the cost of exploding the set of propositions.

expensive: The planner has to find not just the right sequence of actions to reach the goal, but also the right set of value bindings for all those free variables.

Propositional rules allow some interesting optimizations, however. Since all propositions are known at compile time, pre- and postcondition can be represented as simple bit vectors—then during planning, the process of checking preconditions and applying postconditions reduces down to very fast bitwise operations. Another benefit is easy plan caching: It is easy to annotate each propositional plan with a bitmask that describes world states in which this plan could be reused, so that it can be cached and reapplied verbatim in the future. At the same time, propositional rules also have a clear shortcoming: they lack the expressiveness of predicate logic, and require more propositions and rules, which complicates the decision of when it makes sense to use them.

Once implemented, the planner's performance exceeded expectations. Although this was early in production and we did not have a full set of NPCs defined yet, we knew intuitively that search space fan-out was not going to be a big issue.* Even so, plan caching was immediately useful: with the relatively small number of NPC types living fairly stereotyped lives, only a handful of distinct plans actually got created and then cached, so the planner only had to run that many times during the course of the game.

We ended up switching away from planning for a reason unrelated to performance, as discussed next. But even so, our takeaways were that (1) we had a positive experience with planning as a way to prototype AI behavior and explore the design space and (2) there are good ways to make it performant enough for use in games (e.g., using propositional planning, or compact GOAP planners such as presented in [Jacopin 2015]).

34.2.2 Second System: Daily Scripts

In the process of implementing character AI, our understanding of our own design changed. We realized that we wanted to have our NPCs live very stereotyped, routinized lives—they should be pretty predictable, because there were too many of them in the building for the player to care about them in detail. We also wanted a lot of designer control over our peoples' daily routines, to enhance the fiction of the game: so that worker and resident behavior would match the socio-economic status of their workplace or apartment, but at the same time, have a lot of variety and quirks for "flavor."

In the end, we realized that *by employing planning, we were working on the wrong level of abstraction.* We were authoring individual planning steps and trying to figure out how to turn them turn into the right behaviors at runtime—but what we actually wanted to do, was to author peoples' *entire daily routines* at a high level, so that we could have strong authorial control over when things happened and how they varied. We needed to author content not on the level of "how am I going to react to this situation," but on the order of "what does my workday look like today, and tomorrow, and the day after tomorrow."

The representation of behavior in terms of *routines* certainly has a rich history in AI. Some of the anthropologically-oriented research (e.g., Schank and Abelson 1977, Suchman 1987), makes a compelling case that our everyday human interactions are

* We saw this later re-affirmed by Jacopin in his empirical studies of planning in games (Conway 2015): in many games, NPC planning tends to result in numerous short plans, and relatively small search spaces.

indeed highly routinized: that given standard situations, people learn (or figure out) what to do and when, without having to rederive it from first principles, and these stereotyped routines drive their behavior.

Once we realized we were modeling behavior at the wrong level of abstraction, the solution was clear: we decided to abandon planning altogether, and reimplement NPC behavior as libraries of *stereotyped scripts*, which were descriptions of routine activities such as going to the office, going to a sit-down restaurant and sitting down to eat, processing a repair request from a tenant, and so on. Scripts would then be bundled together into various *daily schedules*, with very simple logic for picking the right script based on current conditions, such as the time of day and the contents of a simple "working memory" (e.g., info on where the NPC wants to go, where its current job is, where its current home is, and so on). Below is an example definition of a daily script, for someone who works long hours at the office:

```
name "schedule-office.7"
blocks [
    { from 8 to 20 tasks [ go-work-at-workstation ] }
    { from 20 to 8 tasks [ go-stay-offsite ] }
]
oneshots [
    { at 8 prob 1 tasks [ go-get-coffee ] }
    { at 12 prob 1 tasks [ go-get-lunch ] }
    { at 15 prob 0.5 tasks [ go-get-coffee ] }
    { at 17.5 prob 0.25 tasks [ go-visit-retail ] }
    { at 20 prob 0.5 tasks [ go-get-dinner ] }
    { at 20 prob 0.25 tasks [ go-get-drink ] }
]
```

This definition is split into two sections. In the `blocks` section, we see that they work from 8 am to 8 pm at their assigned work station (e.g., their desk), and otherwise spend time at home. Those *continuous* scripts such as `go-work-at-workstation` are performed as simple looping activity, repetitive but with tunable variations. Then the `oneshots` section specifies individual *one-shot* scripts that might or might not take place, depending on the probability modifier prob, and each script itself will have additional logic to decide what to do (e.g., `go-get-coffee` might start up and cause the NPC to go buy a cup of coffee, thus spending money in your building, but if there are no cafes in the building it will abort and cause the NPC to complain). Finally, all of these scripts bottom out in sequences of individual actions, as described in the next section.

This knowledge representation is simple compared to our previous planning approach, but it was a positive trade-off. Interestingly enough, early in preproduction we had also attempted a more complex internal personality models for NPCs, which included physiological state such as hunger or tiredness, but over time we removed all of this detail. The reasons were two-fold: (1) internal state acted as "hidden information" that made it difficult for both the designer and the player to understand why an individual is behaving in a certain way and (2) when multiplied by dozens or hundreds of NPCs, this made for many frustrating moments of trying to understand when entire populations behaved unexpectedly.

Our main take-away was that *the utility of detailed NPC representation is inversely proportional to the number of NPCs the player has to manage*. When the number of simulated

people is small, players appreciate them being complex. However, as the number gets larger, this does not scale. Having to understand and manage them in detail becomes a burden for both the player and the designer, so it is better to increase the level of abstraction as the number of NPCs increases, and limit the complexity that the player has to deal with.

34.3 Action Performance

Both of our action selection systems—the planner, and the script scheduler—produced sequences of actions that needed to be performed by the NPC. In this section we will look at the flip side of this coin: action performance. We will also talk about two simplifications that enabled efficient implementation: open-loop action performance, and domain-specific representation for pathfinding.

34.3.1 Action Queues and Open-Loop Action Performance

Many NPC AI systems are *closed-loop feedback systems*—they monitor the world while actions are performed, and adjust behavior appropriately, primarily so that they can handle failures intelligently. This comes at a price, however: checking the world has a nonzero computational cost (based on the frequency of updates, the fidelity of the sensory model, etc.), as does deciding whether to act on this new information. Some architectures like *subsumption* (Brooks 1986) or *teleoreactive trees* (Nilsson 1994) accept constant resensing and recomputation as the cost of doing business—while various *behavior tree* implementations, for example, differ greatly in whether the individual nodes revalidate themselves in teleoreactive fashion or cache their activation for extended periods of time.

In our system we take this to a stark extreme: we run action performance almost entirely *open-loop,* without trying to monitor and fix up our behavior based on changes in the world. The main AI setup looks something like this:

1. Action selection picks a script (e.g., *go to work*), and combines it with the NPC's working memory (e.g., *I work at office #243*) to produce a sequence of simple actions: *go into the lobby, wait for elevator, take elevator, walk into office #243, sit down at my desk, etc.*
2. Actions get dropped into an *action queue* and executed in linear order. This is detailed in (Zubek 2010), but anyone who has played The Sims or classic base-building real-time strategy games will be immediately familiar with how this works at runtime.
3. Each action can optionally monitor for custom failure conditions. For example, a navigation action will fail if a path to the destination cannot be found.
4. If a failure is detected, the queue is flushed immediately, and optionally a fallback script may be queued up instead (e.g., *turn to the camera, play displeased animation, and complain about the conditions in this building*).
5. Once the queue is empty, the system runs action selection all over again, which picks the next set of actions and refills the queue.

In effect the system only evaluates the world when it has nothing to do, and once a course of action is decided, it runs open-loop until it either succeeds or gets interrupted.

These sequences of actions also end up being rather short—for example, a script for going to a restaurant and eating might produce a dozen individual actions, altogether taking about an hour of game time (or: less than a minute of real time) to execute.

This works only thanks to the mostly benign nature of this game world: it is usually okay to run open-loop without paying too much attention to the world. If something unexpected does happen, action selection is so inexpensive that we can just abandon the previous activity and start over. So the brief take-away is that, for game designs that allow it, inexpensive action selection enables a whole host of other simplifications, such as skipping proper failure handling in favor of just starting all over again.

34.3.2 Pathfinding over a Simplified Model

The second optimization had to do with pathfinding. The game takes place on what is essentially a 2D grid—a cut-away side view of a building, which can be, say, 100+ stories tall and several hundred grid cells wide, depending on the scenario. A naive implementation of A* pathfinding on the raw grid representation quickly turned out to be insufficient when hundreds of NPCs tried to navigate the grid at the same time.

Naturally, we reformulated pathfinding to be hierarchical to reduce search space. However, instead of using a generic clustering approach such as for example, HPA* (Botea et al. 2004), we used our domain knowledge to produce a specialized compact representation, which made it easier to support the player making ongoing changes to the path graph (as they built, altered or expanded their highrise). In short: based on the game's design, the pathable space divided up into distinct *floor plates,* which were contiguous sequences of tiles on the same floor, such that the character could do a straight-line movement inside a floor plate. Additionally, each floor plate was connected with those above or below it via stairs, escalators, or elevators, together known as *connectors*. Floor plates and connectors became nodes and edges in our high-level graph, respectively, and movement inside each floor plate became simple straight-line approach.

This search space reduction was significant: for an example of a dense building 100 stories tall by 150 tiles wide with four elevator shafts, we reduced the space from 15,000 grid cells to only 100 graph nodes with 400 edges between them. At this point, the data model was sufficiently small to keep running A*, and additional tweaks to the heuristic function prevented the open set from fanning out unnecessarily.

I should also add that we considered alternatives such as JPS and JPS+ over the raw grid, but found them to be an uneasy fit given that the player would be altering the grid space all the time. In particular, JPS (Harabor and Grastien 2012) effectively builds up a compact representation as needed, in order to simplify its search, but as the player keeps changing the game board it would have to keep redoing it over and over again—which seems less optimal than just keeping the source data model compact to begin with. Additionally, JPS+ (Rabin 2015) gains a performance advantage from preprocessing the search space, but this is an expensive step that is not intended to be reapplied repetitively while the game is running.

In the end, although we considered more complex approaches than A*, they became unnecessary once we realized how to *optimize the search space instead of optimizing the algorithm*. We used our domain knowledge to reduce the data model so drastically that the choice of algorithm no longer mattered, and it was a very positive development. Many areas of AI involve search, and model reduction is a classic technique for making it more tractable.

34.4 Conclusions

Drastic simplifications of character AI allowed us to reach our goal of 1000 NPCs at 60 FPS, while keeping development costs down. It was a good example of the power of a super-specialized AI implementation which, although not generalizable to more complex behaviors or more hostile environments, was an excellent fit to the problem at hand, and carried no extra computational (or authoring) burden beyond the minimum required.

This might be an interesting example of the benefits of tailoring one's AI implementation to fit the problem at hand, instead of relying on more general middleware. Although general solutions have their place, it is amazing what can be achieved by cutting complexity mercilessly until there is nothing left to cut.

References

Botea A., Mueller M., Schaeffer J. 2004. Near optimal hierarchical path-finding. *Journal of Game Development*, 1(1), 7–28.

Brooks, R. 1986. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation,* RA-2(1), 14–23.

Conway, C., Higley, P., Jacopin, E. 2015. Goal-oriented action planning: Ten years old and no fear! *Game Developers Conference 2015,* San Francisco, CA.

Fikes, R. E., Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3–4), 189–208.

Jacopin, E. 2015. Optimizing practical planning of game AI. In S. Rabin (ed.), *Game AI Pro 2,* CRC Press, Boca Raton, FL.

Harabor, D., Grastien A. 2012. The JPS pathfinding system. In *Proceedings of the Annual Symposium on Combinatorial Search (SoCS)*, Niagara Falls, Ontario, Canada.

Nilsson, N. 1994. Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, 1, 139–158.

Rabin, S. 2015. JPS+: Over 100x faster than A*. *Game Developers Conference 2015,* San Francisco, CA.

Schank, R., Abelson, R. 1977. *Scripts, Plans, Goals, and Understanding.* Lawrence Erlbaum Associates, Hillsdale, NJ.

Suchman, L. 1987. *Plans and Situated Actions.* Cambridge University Press, Cambridge.

Zubek, R. 2010. Needs-based AI. In A. Lake (ed.), *Game Programming Gems 8,* Cengage Learning, Florence, KY.