Copyright Material – Provided by Taylor & Francis

35

Ambient Interactions

Improving Believability by Leveraging Rule-Based AI

Hendrik Skubch

35.1 Introduction
35.2 From Smart Objects to Smart Locations
35.3 Expressing Interactive Acts
35.4 Script Execution
35.5 Build Chain
35.6 Extensions
35.7 Conclusion
References

35.1 Introduction

It is a hot day in the city of Lestallum. An old man rests comfortably on a bench. Shaded by the nearby palm trees, he reads the local newspaper. A passing tourist takes a break from sightseeing and joins him. Wiping sweat from his forehead, he glances at the newspaper. With a puzzled look in his face, he poses a question to the old man. A brief conversation ensues.

This small scene is one of many that contributes to the realism and vibrancy of a city scenario. Although it is irrelevant for the main plot of a game or for its game play, we value the added immersion. *FINAL FANTASY XV* emphasizes the idea of a journey through a diverse landscape featuring different cultures. In order to bring that world to life and convey different cultures, it is not enough to just place NPCs in the environment. Instead, these NPCs need to interact with the environment and with each other.

This goal leads to two requirements; first, a way to mark up the environment with sufficient information to motivate AI decision-making, and second, a way to express how multiple characters can react to this information. Classical AI scripting methodologies in games such as state machines or behavior trees focus on the actions of individuals. Expressing causal and temporal relationships between the actions of different actors is difficult at best.

Copyright Material – Provided by Taylor & Francis

We present an interaction-centric approach in which interactions are described using STRIPS-like rules. A type of smart object, called *Smart Locations*, use the resulting scripts to control the actions of multiple NPCs. In contrast to planning, symbolic effects are not used to reason about future world states, but instead are used to coordinate the actions of multiple participants by updating a common knowledge representation in the form of a blackboard and reactively applying rules based on that structure. Thereby coordination becomes the focus of the language and is expressible in a straightforward manner. Moreover, the resulting rule-based scripts are highly adaptive to different situations. For example, in the city scene mentioned above, if there was no old man, the tourist might still sit down to rest. Alternatively, if the old man was joined by a friend, the tourist might have walked by due to lack of space.

35.2 From Smart Objects to Smart Locations

The concept of smart objects was originally conceived for The Sims (Forbus 2002). Smart objects are inanimate objects in a scene that carry information about how they can be used by an agent. For example, a chair may carry the information that it can be used to sit on. It even provides the necessary animation data for doing so. In a sense, by emphasizing objects instead of agents, smart objects reverse the idea of traditional agent-based AI, thereby decoupling the AI from the data necessary to interact with an asset. This allows for new objects to be added to a scene and become usable by the AI without modifications to the AI itself.

More recently, the concept of smart objects has evolved into that of smart zones by de Sevin et al. (2015). Smart zones abstract away from concrete objects and add the idea of roles that NPCs can fulfill, thereby facilitating multiple NPCs interacting with the environment in order to create a richer scene.

In a similar way, smart locations abstract away from concrete objects, as shown in Figure 35.1. They are invisible objects that refer to multiple concrete objects. For example, a single smart location may refer to two chairs and a table. This allows it not only to inform agents about the existence and usability of individual objects, but also to capture relationships between them, such as furniture grouping. Although smart zones use timeline-based scripts with synchronization points to drive the behavior of NPCs, we use a more expressive scripting language based on declarative scripting rules. Furthermore, we add an additional abstraction layer between the location object embedded in the world and the static script object. These script objects may contain additional data necessary for its execution, such as animations. The resulting decoupling allows for scripts to be exchanged, whereas the smart locations stay in place. But smart locations do not just contain information; they essentially govern the usage of the objects they refer to. To that end, they emit signals to agents in the vicinity. These signals can range from mere notifications to commands, depending on the type of emitter. In *FINAL FANTASY XV*, we use four different kinds of emitters:

- *Notification emitter*: These emitters merely inform agents about the existence of the smart location, the objects it knows about, and a set of ontologically grounded tags. Notification emitters are mainly used to inform the more autonomous characters, such as the player's buddies.

Copyright Material – Provided by Taylor & Francis

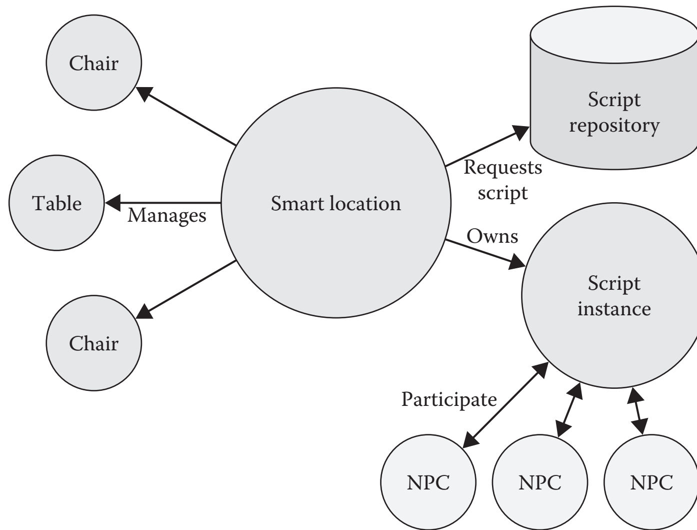

Figure 35.1

Smart locations manage static props and dynamic script instances.

- *Script emitter*: This is the most frequently used emitter type. Based on the number and type of NPCs in the vicinity, this emitter performs a role allocation and initiates the execution of a script. We will elaborate on this process in the following sections.
- *Spawn emitter*: Scripts that require three or more specific characters occur rarely when using just NPCs that pass by the location. This has already been noted by Blondeau during the development of *Assassin's Creed Unity* (Blondeau 2015). Spawn emitters fill this gap by spawning a set of NPCs at points specified by designers. This is also used to place NPCs at otherwise unreachable locations, such as balconies.
- *Player emitter*: Finally, the player emitter communicates with the player by means of interaction icons, which, when reacted to, will initiate a script. The player themselves joins this script as an NPC. Player input is fed to the script's blackboard, allowing designers to script short interactive scenes in a dynamic fashion without changing methodology.

35.3 Expressing Interactive Acts

We express interactions in the form of rule-based scripts. Rules are drawn from STRIPS (Fikes and Nilsson 1971), but instead of being used for planning, they form a declarative scripting language, where rules operate over a special kind of blackboard, a tuple space. Tuple spaces were originally devised as a way to coordinate data access in distributed systems (Carriero et al. 1994). The combination of both methodologies allows agents to exchange information by executing rules on the same tuple space. In the following, we briefly introduce STRIPS and tuple spaces as the two foundations of interaction scripts.

Copyright Material – Provided by Taylor & Francis

35.3.1 Foundation: STRIPS

Stanford Research Institute Problem Solver, or STRIPS, was one of the first automated planning systems. Here, we refer to the language of that system. In STRIPS, an action is described by a precondition that must be true in order for the action to be executable, and the positive and negative effects it has on the world. Positive effects add facts to the world, whereas negative effects remove them. Note that, although the original formulation of STRIPS is based on pure propositional logic, we use a first-order notation here and in the remainder of this chapter. Since we only consider finite domains under closed-world assumption, this is nothing more than syntactic sugar, as all formulas can be compiled into propositional logic.

Consider a classic example from academia, called the Blocks-world. In this world, a robot is tasked with stacking blocks on a table on top of each other. One of the actions the robot has is PickUp(B)—with the intuitive meaning that the robot will pick up a block B with its actuator. The precondition for this action is $\text{onTable}(B) \wedge \neg\text{on}(A, B) \wedge \neg\text{holding}(K)$, meaning that B should be on the table, nothing should be on top of it, and the robot should not already be holding anything else in its actuator. The positive effect of this action is holding(B)—the robot is now holding block B. The negative effect is onTable(B)—the block is no longer on the table. With action descriptions like this, and state descriptions for a goal and an initial state, planning algorithms can chain actions together in order to find a sequence of actions leading from an initial state to a goal state.

Note that the precondition acts as a selector over which block B will be picked up. In Section 35.4.3, we will discuss how both the truth value of the precondition and an assignment for the variables in the rule, such as which block is chosen for variable B, can be determined at the same time by means of a backtracking algorithm.

35.3.2 Foundation: Tuple Space

Tuple spaces are a way to coordinate distributed systems. Since we do not need to operate a truly distributed system, that is, there are no synchronization issues or accidental data loss, we use a simplified version here. The tuple space provides facilities to store, query, and modify facts used in STRIPS rules, such as onTable(B) from above. Basically, a tuple space can be thought of as a multimap, where the name and the arity (arguments) of a predicate (e.g., onTable/1) are used as keys to map onto all ground facts currently considered being true. Thus, in this form, it only represents positive knowledge under the closed-world assumption, matching STRIPS and enabling negation as failure reasoning.

35.3.3 The Interaction Language Model

A basic interaction script consists of a set of roles and a set of rules. Intuitively, roles define which kind of actors can participate in a script and how many, whereas rules indicate what the actors are supposed to do.

Whenever an actor is participating in a script, it is assigned exactly one role. Multiple actors can be assigned to the same role. An actor may be able to take on different roles.

Copyright Material – Provided by Taylor & Francis

For example, the old man on the bench may be able to take on the roles citizen, male, elder, and human, but not waiter or tourist (but in the context of a single script instance, he will take on only one role). More precisely, a role consists of the following:

- *Name*: The unique name of the role, such as tourist, waiter, or citizen.
- *Cardinality*: The minimum and maximum number of actors that can take on this role. Role allocation will try to satisfy all cardinalities of the script. If that is not possible, the script will not start. Should any cardinality be violated while the script is running, for example because an actor left, the script will terminate.
- *Flags*: An extensible set of flags that further describes how a role is to be treated at runtime. Most importantly these flags govern whether or not an actor can dynamically take on a role and join the script while it is running, or if any special actions are necessary when an actor is leaving the script, such as sending them a command to move away from the smart location.

We may describe the initial scene between the old man and the tourist with these roles:

- *Elder*: 0..1 dynamicJoin = true
- *Tourist*: 0..2 dynamicJoin = true

Thereby, the script can run with any combination of up to two tourists and one elder. Any role initially unfulfilled can be assigned later dynamically. This permits a variety of different sequences of events; for example, the old man may join the tourist on the bench.

In contrast to roles, rules entail what actors should do once they participate in a script. In a minimal version of this scripting language, rules must consist of at least:

- *Precondition*: The condition that has to hold in order for the rule to fire.
- *Action*: The action an actor should undertake. Note that actions are optional; rules without actions simply apply their effects and can be used to chain more complex effects. In our case, each action identifies a state machine or behavior tree on the lower individual AI level. The degree of abstraction is arbitrary, but we think a reasonable degree is achieved by actions such as “sit down,” “go to,” or “talk to.” Of course, this depends on the concrete game considered.
- *Action parameters*: A list of key-value pairs that are passed to the lower layer that executes the action. For example, movement speed or animation variations often occur as parameters.
- *Addition* (δ^+) *and Deletion* (δ^-): The list of positive and negative effects a rule has. They are applied immediately when the rule fires.
- *Deferred Addition and Deferred Deletion*: Similar to Addition and Deletion, the deferred versions modify the tuple space. However, they are applied only after the action has successfully terminated. If there is no associated action, deferred effects are applied immediately.
- *Termination type*: A rule can be used to terminate the whole script or cause an individual actor to leave the script. The termination type is used to indicate this.

Copyright Material – Provided by Taylor & Francis

Additionally, we added the following syntactic sugar to simplify common cases:

- *Target*: Most actions have a target, as is implied by the proposition in their name such as goto or lookat. Because these are used so frequently, we treat the target separately instead of using action parameters.
- *Role*: Acting as an additional precondition, the role property of a rule limits the NPCs that can execute a rule to those of the corresponding role.
- *State*: A state also acts as an additional precondition. Only actors currently inhabiting this state can execute this rule. Thereby we overlay the rule-based execution with a simple state machine. It is trivial to formulate state machines using preconditions and action effects; however, a simple state machine greatly reduces the perceived complexity and length of preconditions.
- *Next state*: After successfully executing a rule, an NPC will transition to this state.
- *OnError*: In order to simplify handling of errors stemming from failed actions, a list of facts can be provided to be added if a rule fails.

Treating these notions explicitly simplifies the task of writing scripts, and, in many cases, allows optimizing script execution further. For example, by rearranging rules so that rules of the same state are consecutive in memory, only rules of the current state of an actor have to be evaluated.

Let us consider some simple example rules useful for the bench scenario. First, we need a rule for sitting down:

- **Rule 1**:
  - **Action:** $\text{sit}(X)$
  - **Precondition:** $\text{seat}(X) \wedge \neg\text{reserved}(X,Y)$
  - **δ^+:** $\text{reserved}(X,.me)$
  - **deferred δ^+:** $\text{sitting}(.me) \wedge \text{timer}(.me,.now + randf(2,5)^* .minute)$

The action sit has a target, namely the object to sit on, denoted by the variable X, which is sent as a game object to the individual AI. The implementation of sit may be arbitrarily complex, in this case, it will query the animation system for where to be in relation to X when triggering the sit-down animation, path plan to that point, move, and finally trigger the animation.

The specific game objects that X can possibly refer to at runtime are known by the smart location. Before executing the script, the smart location dumps this information into the tuple space using the predicate seat. The second predicate, reserved, is used to formulate a reservation system inside the tuple space. Note that the keyword reserved does not have any meaning outside of the rule for sitting. The definition of the rule gives meaning to the symbol. The same holds true for the predicates sitting and timer which are used to inform other NPCs that the sitting action has now been completed and to store a timer for future use, respectively. The difference between addition and deferred addition allows us to reserve the seat the moment the action is committed to, while informing other NPCs of the action completion several seconds later.

However, not everything can be solved purely by means of the tuple space, sometimes we need to call into other systems or refer to special purpose symbols. These symbols are

Copyright Material – Provided by Taylor & Francis

prefixed with a dot "." as in $.me$, which refers to the NPC that is currently evaluating the rule. Therefore, the positive effect reserved$(X, .me)$ will substitute X with the game object that represents the seat and $.me$ with the NPC in question before inserting the resulting tuple into the blackboard. Once the NPCs are sitting, we can drive a simple randomized sequence of talking and listening actions using the following three rules:

- **Rule 2:**
  - **Precondition:** $\neg\text{talker}(X) \wedge .any(Y) \wedge \text{sitting}(Y)$
  - **δ^+:** $\text{talker}(Y)$

- **Rule 3:**
  - **Action:** $\text{talk}(X)$
  - **Precondition:** $\text{talker}(.me) \wedge .any(X) \wedge X \neq .me \wedge \text{sitting}(X)$
  - **Deferred δ^-:** $\text{talker}(.me)$

- **Rule 4:**
  - **Action:** $\text{listen}(X)$
  - **Precondition:** $\text{talker}(X) \wedge X \neq .me$

Rule 2 does not cause an action; it is merely used to designate a single sitting NPC as the currently talking one by means of the predicate talker. The built-in predicate $.any$ unifies its argument with a random participating NPC. The following predicate sitting limits this to a random sitting NPC. The backtracking algorithm achieving this behavior will be discussed in Section 35.4.3.

Rule 3 triggers a talk-action for the designated NPC. Supplying another random sitting NPC as target X, allows for LookAt-IK and other directed animations to take place. After the talk-action finishes, talker is removed, in turn triggering the first rule again. The fourth rule lets us model a listening reaction to the talking action, such as a nod.

Finally, in order to avoid an endless discussion between the NPCs, we add a termination rule that refers to the timestamp introduced in Rule 1, which causes the NPCs to get up and terminate its participation in the script:

- **Rule 5:**
  - **Action:** getup
  - **Precondition:** $\text{timer}(.me, T) \wedge T < .\text{now}$
  - **Terminate = true**
  - **δ^-:** $\text{timer}(.me, T)\ \text{sitting}(.me)$
  - **Deferred δ^-:** $\text{reserved}(.me, X)$

Note the free variable X in the deferred deletion causes all matching tuples to be removed from the tuple space.

Formally, our query language is limited to conjunctions of literals. A literal is a possibly negated predicate such as reserved(X, Y) or $\neg$talker(Z). Thus, we exclude disjunctions and negations over the scope of multiple literals from the language. Furthermore, the query language also relates to DataLog (Ceri et al. 1989), a function-free declarative language.

Copyright Material – Provided by Taylor & Francis

That means function symbols are not allowed to occur, thereby greatly reducing the complexity of evaluation compared to other declarative languages, such as Prolog. We can still tolerate function symbols to occur by simply enforcing them to be evaluated immediately when they are encountered, circumventing any costly term manipulation. This allows us to express functional terms such as $.now + randf(2,5) * .minute$ in Rule 1, or distance$(.me, Someone)$ without incurring the cost of term manipulation in a full first-order language. A tool side verification step ensures that this treatment of function symbols is safe, for example, that *Someone* will always be instantiated to an NPC when evaluating distance$(.me, Someone)$.

35.4 Script Execution

A smart location equipped with a script emitter will periodically query a spatial database for NPCs in the vicinity in order to start an interaction scene. Based on the smart location's preference for different scripts, the NPCs found, and the currently loaded scripts, a script is chosen to be executed. From there on, the script is updated regularly in the following way:

- Shuffle all participants. This counters any unintentional bias in the rule set toward the first or last NPC. Any predicate that accesses participants effectively accesses the shuffled list.
- Apply the deferred effect of any action that finished and the OnError list of any failed action.
- For each participant not currently executing an action, find the first matching rule in its state. If found, trigger the action and apply the effect.
- Remove all NPCs from the script that encountered a termination flag.
- Terminate the script if not enough NPCs remain.

Since scripts operate on a local blackboard, each NPC only participates in at most one script at a time, and smart locations have exclusive ownership over their props, multiple script instances can be executed concurrently without the need for thread synchronization.

35.4.1 Role Allocation

Role allocation is the process of assigning actors to roles such that the resulting assignment satisfies all given constraints, such as our cardinalities. In practice, we use additional constraints to reflect that some NPCs, such as families, act as groups and only join scripts together. Since each actor can satisfy multiple roles and each role may require multiple actors, the problem is NP-hard (Gerkey and Mataric 2004). However, the specific problem instances encountered in interaction scripts are typically very small and simple, meaning rarely more than three or four distinct roles and rarely more than five actors. Furthermore, we do not require the resulting allocation to be optimal with respect to some fitness function. Indeed, we can even allow the role allocation to fail occasionally. Thus we can formulate role allocation as a Monte-Carlo algorithm by randomizing its input.

After randomization, role allocation simply assigns NPCs greedily to roles until the lower bound of the respective cardinality is reached. If a role-cardinality cannot

Copyright Material – Provided by Taylor & Francis

be satisfied in this way, the allocation fails immediately. Subsequently, potentially remaining NPCs are assigned until the maximum cardinality is reached or no more NPCs can be added.

35.4.2 Joining Late

Although a smart location is running a script, it will not start a second one to avoid concurrent access to its resources. However, it will periodically query the spatial data base for NPCs in the vicinity that can join the already running script instance. Whether or not this is possible is indicated by the flag "dynamicJoin" introduced in Section 35.3.3. This behavior allows for more dynamic scenes where NPCs come and go, such as street vendors serving pedestrians walking by.

35.4.3 Rule Evaluation

At its core, our rule-based scripting language is declarative in nature. This means that execution and evaluation are often the same. For example, successfully evaluating a precondition will result in an assignment of values to its variables. Rule evaluation is based on two main algorithms: unification and backtracking search.

Unification is an essential algorithm for term manipulation. Given two terms, unification decides whether or not they can be equal and, if so, applies an equalizing substitution. For example, the term 2*X+f(A, B) can be unified with 2*A+K by substituting X with A and K with f(A, B). For details, consult work on automated reasoning, such as done by Fitting (Fitting 1996).

Since we draw from DataLog and evaluate function symbols immediately, we can simplify unification to three cases: constant to constant, variable to constant, and variable to variable. The first case becomes a simple equality check, and is the only case that can fail. The second case is a pure assignment, and only the last case requires a more work, as the system has to keep track of which variables are unified with each other. We recommend to identify variables with indices and, for each variable, to keep track of the respective variable with the lowest index it has been unified with.

The second algorithm needed, backtracking, is a common method to search for a solution in a structured space, such as a conjunction of literals. Its origins are lost in history, the earliest known application of this algorithm was done by Ariadne to solve Minos' labyrinth (cf. Morford et al. 2013). In essence backtracking explores the state space by successively expanding options and returning to earlier states if it encounters a failure or if it exhausts all options at a particular junction.

Hence, backtracking requires the ability to reestablish a former state in the space explored. In general, this can be achieved with three different techniques: saving the previously visited states, recomputing them from the start, or undoing actions applied. We use a combination of the first two options; we save whatever was assigned to the variables of the rule in the previously visited state, but require any predicate to compute its nth solution from scratch without saving data leading to solution n–1. Alternative solutions using scratchpad stack memory are possible, but have not been explored in the context of this work. For further reading on backtracking algorithm we recommend literature discussing DPLL algorithms for SAT solving (Davis et al. 1962), such as (Nieuwenhuis et al. 2004).

Copyright Material – Provided by Taylor & Francis

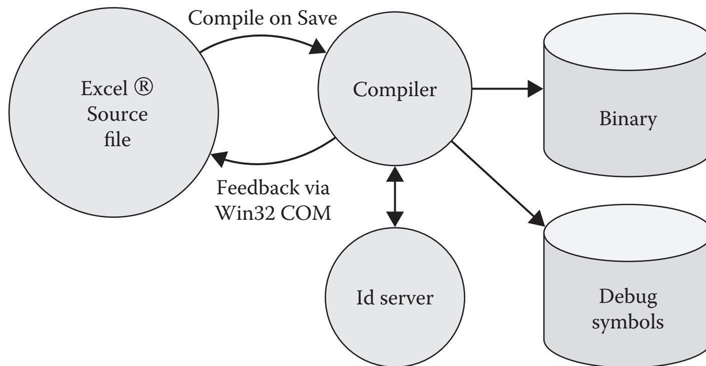

Figure 35.2

The build chain transforms textual scripts into an interpretable binary format.

35.5 Build Chain

The success of any novel scripting methodology depends on its ease of use. We designed the build chain, as shown in Figure 35.2, to provide quick iteration cycles and make use of tools designers are typically familiar with. Scripts are edited in Excel, where table oriented perspective lends itself well to rule-based scripting. Upon saving, a compiler translates the XML source to a binary loadable by the runtime. All identifiers used in the script, such as roles, predicates, and actions are translated to unique 32-bit identifiers by means of an id-server. In order to supply a debug UI with readable names again, a separate file with debug symbols is generated.

35.5.1 Validation

The mathematically grounded properties of STRIPS allow us to detect various scripting errors during compilation and supply feedback directly into the editor. Most notably, we can detect common problems such as:

- Unreachable states.
- Unexecutable rules (specialization of rules of higher precedence).
- Usage of uninstantiated variables.

However, the Turing completeness of the scripting language prevents us from detecting all problems.

35.6 Extensions

The concepts presented here were used heavily during production of *FINAL FANTASY XV*. Naturally, we made adjustments to the original system to accommodate unforeseen needs and situations.

Copyright Material – Provided by Taylor & Francis

- *Beyond agents*: Although actions of NPCs are easily representable in the language presented, achieving other effects, such as opening a shop UI or reacting to the player clicking an icon was not. Most of these issues relate to the communication with other game systems. We addressed this by wrapping these systems into proxy objects that participate in a script as if they were NPCs. Thus the shop itself becomes an NPC with available actions such as opening and closing specific shop pages. Moreover, these proxy objects push information about ongoing events in their domain into the blackboard. For example, the shop informs the script of what the player is buying or selling. The shopkeeper's reaction can then simply be driven using the blackboard.
- *Templating*: During development, we discovered sets of highly similar scripts being used throughout the game, such as scripts controlling different shopkeepers. The logic within the scripts was almost identical, but various parameters such as motion-sets and shop configurations were changed. We countered this effect by allowing for one script to include another as a base and introduced meta-parameters that the script compiler would replace before creating the binary. Thus we introduced a string templating system in the tool chain. The runtime was completely agnostic about this since it only sees the resulting scripts.

35.7 Conclusion

In this chapter, we presented a novel way of scripting interactions between ambient NPCs using STRIPS rules that modify a blackboard shared by participating NPCs. We described how the necessary information to ground a script in its environment can be supplied by situated objects, namely smart locations, which govern the usage of multiple props. Furthermore, we presented the relevant algorithms for evaluation and role allocation. This approach shifts the modeling focus from the actions of the individual to the interaction between multiple agents and thus significantly simplifies the representation of multiagent scenes encountered in living breathing city scenarios.

The declarative nature of the rule-based approach caused an initially steep learning curve for our designers, but was adopted after a brief transition period. Compared to other scripting methodologies, three advantages became apparent during the development cycle:

- *Locality*: Problems are contained within a single script and thus easier to find.
- *Adaptability*: Scripts adapt themselves naturally to a wide variety of situations.
- *Validation*: The ability of the script compiler to find common mistakes early on greatly reduced iteration time.

References

Blondeau, C. Postmortem: Developing systemic crowd events on Assassin's creed unity, GDC 2015.

Carriero, N. J., D. Gelernter, T. G. Mattson, and A. H. Sherman. The Linda alternative to message-passing systems. *Parallel Computing*, 20(4): 633–655, 1994.

Copyright Material – Provided by Taylor & Francis

Ceri, S., G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge & Data Engineering*, 1(1): 146–166, 1989.

Davis, M., G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7): 394–397, 1962.

de Sevin, E., C. Chopinaud, and C. Mars. Smart zones to create the ambience of life. In *Game AI Pro 2*, ed. S. Rabin. Boca Raton, FL: CRC Press, pp. 89–100, 2015.

Fikes, R. E. and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. Technical report, AI Center, SRI International, Menlo Park, CA, May 1971.

Fitting, M. *First-order Logic and Automated Theorem Proving* (2nd Ed.). Springer-Verlag New York, Inc., Secaucus, NJ, 1996.

Forbus, K. Simulation and modeling: Under the hood of The Sims, 2002. http://www.cs.northwestern.edu/~forbus/c95-gd/lectures/The_Sims_Under_the_Hood_files/v3_document.htm (accessed July 5, 2016).

Gerkey, B. P, and M. J. Mataric. A formal analysis and taxonomy of task allocation in multi-robot systems. *The International Journal of Robotic Research*, 23(9): 939–954, 2004.

Morford, M., R. J. Lenardon, and M. Sam. *Classical Mythology* (10th Ed.). Oxford University Press, New York, 2013.

Nieuwenhuis, R., A. Oliveras, and C. Tinelly. Abstract DPLL and abstract DPLL modulo theories, *Proceedings of the International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*. Montevideo, Uruguay: LPAR, pp. 36–50, 2004.