8

Production Rules Implementation in *1849*

Robert Zubek

8.1 Introduction

This chapter presents implementation details of the production rule system used in the game *1849*. The system's main design goals were enabling quick iteration via a data-driven approach and good performance on a variety of hardware, down to significantly underpowered tablet devices.

First, we discuss the details bottom up, from the world model, through rule implementation, up to the overall rule system that manipulates them. Then, in the second half, we examine the performance consequences of these design choices, as well as lessons learned in the process of implementing the system.

8.2 Game Mechanics and Production Rules

1849 is a city building and management game for tablets, desktops, and the web. The fiction of the game is that gold has just been discovered in California, and player's task is to build gold mining towns and make money in the gold rush. The following is an overview of the game mechanics and simulation.

8.2.1 Game Mechanics

In terms of game mechanics, the game is a classic city builder, along the lines of early Impressions Games such as *Caesar* or *Zeus*. The main units of gameplay are as follows:

- *Buildings*, which the player places in town; they can be houses for residents or workplaces that produce resources or city benefits (such as fire prevention).
- *Resources* are created by buildings, either from nothing (such as farms producing wheat) or by consuming other resources (such as bakery consuming wheat and producing bread).
- *Workers* are the fuel powering all these buildings; they cannot be directly controlled by the player, but they can be influenced by providing them the resources they want.

The main feedback loops are set up such that workers power all buildings, but they are fickle and sensitive to what resources are available. As the town grows, more and more workers arrive looking for work, but they also demand more complex resources, and if they don't get what they want, they vote with their feet and leave, causing workplaces to shut down. Initially, their demands are simple, just food and drink, but soon, they start demanding increasingly processed resources, such as shoes, clothes, or newspapers. The player can either try to import those processed resources at a high cost or build out resource conversion buildings and manage their logistics. Much of the fun and difficulty of the game comes from the "spinning plates" feeling, of setting up these increasingly complicated resource production and conversion chains and then maintaining them and making sure that they are all running smoothly, that workers remain happy, and that the town's overall budget is trending in the right direction.

8.2.2 Game Simulation

The game simulation is implemented using a production rule system: all buildings run a collection of stand-alone rules that simulate the town's economy.

We can discuss them as a hierarchy of abstractions:

- Each building is a stand-alone rule executor for a set of rules.
- Each rule has some conditions that can match and produce some actions.
- Conditions typically involve queries about resources or the world, and actions typically involve resource modification and issuing side effects.
- Resource queries and modification bottom out in a data model optimized for specific types of context-sensitive access.

8.3 Rule System

Having introduced the layers of the system, let's discuss them bottom up.

8.3.1 Resources

The basic atomic unit of game economy is a *resource*. This is an `<id, amount>` tuple, such as "50 dollars" or "10 units of gold" or "50 units of stone." Everything that can be produced or consumed is a resource.

Most resources are concrete, like gold or stone or food. But there are also abstract resources such as people and map effects such as crime level.

When you build a house, and people move in, that's represented as the house gaining a "1 resident" resource—and later, when that resident gets a job, the workplace gains a "1 worker" resource as well (and the reverse happens when the resident moves out).

Map effects are things like crime, boredom, or fire risk. For example, every house creates a tiny amount of fire risk, say, "0.01 fire risk" per day, and this resource collects up in the world, and later causes fires, as we'll describe in a moment.

8.3.2 Resource Bins

Resources are not loose objects, rather, they're stored in bins. A resource bin contains a whole bag of resources and their amounts. There are three types of bins in the game.

First, *player's inventory* during a game session is stored in a bin. For example, the facts that I have $1000 in cash and 10 units of gold ready for trade are just two resource entries in my bin.

Second, each *board unit* (such as a building) has its own bin, which is its own inventory. For example, when a wheat farm grows wheat, it inserts a bunch of wheat units in its own bin. But those units are not yet usable by the player. There's a separate delivery step that has to happen, to deliver this wheat from the building to the player's inventory.

Finally, each *map tile* has a resource bin that's separate from any building that might sit on top of it. For one example, gold underground is represented as a gold resource inside that tile's bin, and it needs to be mined out of the ground and into the building's bin. For another example, fire hazard is a resource, conjured up and inserted into the world by wooden buildings.

8.3.3 Conditions and Actions

Since almost everything in the simulation is a resource, a lot of the game is based on resource conversions. Some simplified examples from our data definition files:

Ranch produces meat and leather, and shows an animated NPC at work:

```
"doWork":
  "outputs": ["unit 6 meat", "unit 6 leather"]
  "success": [
    "_ a-spawn-worker npc npc-farmer action ranch days 7"
  ]
```

Ranch delivers meat into storage, 20 units at a time:

```
"deliverWork":
  "frequency": "every 5 days",
  "checks": ["unit workers > 0"],
  "inputs": ["unit 20 leather"],
  "outputs": ["player 20 leather"],
  "success": [
    "_ a-spawn-walker npc npc-delivery
       to bldg-trade-store then return"
  ]
```

Cobbler brings leather from storage if it doesn't have any, consumes it, and produces shoes:

```
"bringMaterials":
  "checks": ["unit workers > 0", "unit leather < 2"],
  "inputs": ["player 8 leather"],
  "outputs": ["unit 8 leather"]
"doWork":
  "inputs": ["unit 2 leather"],
  "outputs": ["unit 3 shoes"]
```

But conversion rules don't have to be limited to just buildings bins—they also frequently interact with map tiles underneath and around:

Gold mine consumes gold from the map tiles underneath and produces gold in its own inventory, until all ground gold has been exhausted:

```
"doWork":
  "inputs": ["map 5 gold"],
  "outputs": ["unit 5 gold"]
```

Every wooden house produces a little bit of fire risk in the map tile underneath:

```
"produceFireHazard":
  "frequency": "every 7 days",
  "checks": ["map fire-hazard < 1 max"],
  "outputs": ["map 0.04 fire-hazard"]
```

Fire brigade consumes all fire risks from the map, within a given radius, using a special action:

```
"consumeMapResource":
  "frequency": "every 7 days",
  "checks": ["unit workers > 0"]
  "success": ["_ a-change-resource-in-area
               radius 5 res fire-hazard amount -1"]
```

As you can see, the fact that gold comes from underground, while food and other things are made in buildings, is not actually hard-coded anywhere in the engine. Right now, it's just a matter of convention. This means that you could rewrite the rules such that, for example, the cobbler makes shoes and inserts them underground inside the tile. You probably wouldn't want to, because nobody would be able to get at those shoes if they wanted them, but it's a possibility.

8.3.4 Rule Execution

As you can see from the previous examples, each rule consists of several elements. Here is the complete list:

- "Frequency": how often we check.
- "Checks": all of these have to be satisfied.
- "Inputs": if checks are satisfied, we check if desired inputs exist, and if so, they will be consumed.

Listing 8.1. Pseudocode for rule matching algorithm.

```
for each rule that should run at this point in time
     if all checks are satisfied
          if all inputs exist
               consume inputs
               produce outputs
               run success actions
          else
               run failedInputs actions
     else
          run failedChecks actions
```

- "Outputs": if inputs were consumed successfully, these will be produced.
- "Success": actions to run if this rule was applied successfully (neither checks nor inputs have failed).
- "FailedInputs": fallback actions to run if inputs were insufficient.
- "FailedChecks": fallback actions to run if checks failed.

The algorithm in pseudocode is listed in Listing 8.1. As we can see, "frequency" and "checks" both denote *conditions* in which the rule runs, "inputs" defines both *conditions* to be checked and related *actions* (consume inputs), while "outputs" and other fields define *actions* only. Frequency is pulled out separately as an optimization step (see next section).

8.4 Performance

Our production system is very efficient—in a town with many hundreds of entities, the rule engine's CPU consumption is barely noticeable in the profiler, even when running on rather underpowered tablets.

Most of the processing power is spent, predictably, on checking conditions. One of the design goals for this system was to make sure conditions can be checked quickly, ideally in constant or near-constant time, to help with performance.

We have three optimizations in place to help with this: flexible frequency of rule execution, a drastically simplified language for conditions and actions, and an efficient world model that is inexpensive to query.

8.4.1 Condition Checking Frequency

Production systems vary in how often the rules should be run. For example, we could run rules whenever something changes in the world, which in a game could be every frame, or maybe on a fixed schedule, such as 10 Hz, or on every game "turn" in a turn-based game.

In *1849*, the game's simulation is triggered off of game clock days (e.g., a farm produces wheat every 7 days), so we felt no need to run the rules too often. Our default frequency is once per day, and we made it easy to raise or lower the frequency as needed on a per-rule basis.

Here is an example of how frequency is specified—it's pulled out of the conditions definition into its own data field:

```
"produceFireHazard":
  "frequency": "every 7 days",
  "checks": ["map fire-hazard < 1 max"],
```

Finally, we implemented a very simple scheduler that keeps track of which rules are supposed to run when, so that they don't get accessed until their prescribed time.

8.4.2 Condition Definition Language

Many rule systems express conditions and actions in an expressive language such as predicate logic, so that the developer can make queries and assertions about entities as a class without committing to specific instances, and let the computer figure out to which entities those rules can be applied.

Here is a made-up example in a made-up predicate language:

```
If is-a(X,gold-mine) and is-a(T,map-tile) and
is-under(T,X) and contains(T,R,5) and is-a(R,gold)
=> Then increment(X,R,5) and increment(T,R,-5)
```

This kind of a rule would be very expressive and general. However, finding entities in the world that match this query can get expensive quickly: it's essentially a search problem. While numerous optimizations for inference systems are well known (e.g., the Rete algorithm [Forgy 82]), they're still not enough, given our desire to make conditions execute in constant or near-constant time.

Conditions and actions we use in our engine are not so generic. Instead, they are more contextual, which lets them be simpler. Once again, here is our gold mine example:

```
"doWork":
  "inputs": ["map 5 gold"],
  "outputs": ["unit 5 gold"]
```

Here, "map" and "unit" are like variables, in that they're not specific entities like "gold mine #52"—but they're also not free variables like `X` was in the previous example. Instead, they're contextually bound indexicals: "unit" refers to the entity that's currently executing this rule, "map" refers to all tiles underneath the unit, and "player" refers to the singleton entity that keeps player's city inventory.

In other words, instead of using objective representation and predicate logic, we use deictic representation [Agre 87], with variables that are already contextually bound at query time to particular entities. Game units typically only care about themselves and their immediate surroundings, so deictic representation is a perfect match.

This choice constrains our system's expressiveness, compared to a language with completely free variables and unification, but it drastically eliminates a huge search problem and associated costs.

8.4.3 Data Model

Most conditions are resource queries, and most actions are resource modifications. For example: check if there is gold underground, and if so, consume it and produce gold in my inventory; or check if I have any workers working here, and if there is gold in my inventory, and if so, deliver gold over to player's inventory, and so on.

As we described before, we store resources in resource bins, and those bins are attached to units, map tiles, and the player's data object. Each resource bin is implemented as a vector of 64 floating-point numbers, indexed by resource (because there are currently 64 resource types).

A resource query such as "unit gold > 5" then works as follows: first, we get a reference to the unit's own resource bin (via a simple switch statement), then look up resource value (an array lookup), and finally do the appropriate comparison against the right-hand side value (another simple switch statement). All this adds up to a constant-time operation. Similar process happens for update instead of a query.

A query such as "map gold > 5" is marginally more expensive, because it means "add up gold stored in all tiles under the unit and check if > 5". Fortunately, units are not arbitrarily large—the largest one is 2 × 2 map tiles—which means we execute at most four tile lookups, making it still a constant-time operation.

And as a fallback, we allow ourselves to cheat if necessary: both conditions and actions can also refer to a library of named built-in functions, and those can do arbitrary computation. For example, the fire brigade has a built-in action `a-change-resource-in-area` that consumes a pre-tuned amount of fire risk resource within its area of effect, but this operation is actually linear in map size. We use such actions rarely.

8.5 Lessons from *1849*

With the system overview behind us, we'll quickly go over what worked well in the process of building our game using this engine, and what, with the benefit of hindsight, we wish we had done differently.

8.5.1 Benefits

Performance was clearly a high point of the system, which can run cities with hundreds of active entities without breaking a sweat, even on comparatively underpowered tablet devices. We could probably push production rules even further, if the rendering subsystem had not claimed all available processor cycles already.

Also, as you can see from our examples, the rules themselves are specified in a kind of a domain-specific language, based primarily on JSON, with condition and action bodies expressed as strings with a specific syntax. They get deserialized at load time into class instances, following simple command pattern.

Exposing game rules as a DSL that can be loaded up with a simple restart, without rebuilding the game, had the well-known benefits of data-driven systems: decoupling configuration from code, increasing iteration speed, and ultimately empowering design.

8.5.2 Lessons

At the same time, we ran into two problems: one with how our particular DSL evolved over time, and one with production systems and how they matched the game's design.

The DSL was initially developed to support only queries such as "`<bin> <resource> <comparison> <value>`" or actions such as "`<bin> <resource> <delta>`".

These were appropriate for most cases, but we quickly found ourselves wanting to do more than just resource manipulation. For example, we wanted to start spawning workers to go dig up gold or carry it in wheelbarrows to the storage building—or even more mundane things, like playing sound effects or setting or clearing notification bubbles if a building is understaffed or can't get road access.

Over time, we added support for more types of actions, and a generic deserializer syntax, which supported actions such as `" _ a-spawn-worker npc npc-farmer action ranch days 7"`. This was just syntactic sugar for a definition like `{" _ type": "a-spawn-worker", "npc": "npc-farmer", "action": "ranch", "days": 7}`, and that in turn just deserialized into the class `ASpawnWorker` and filled in the appropriate fields.

In retrospect, we should have added support for custom or one-off conditions and actions from the very beginning; that would have saved us engineering time later on reworking parts of the system. Even in the most organized system design, there will *always* be a need for one-off functionality to achieve some specific effects, and all systems should support it.

Separately from this, we also discovered a representational deficiency, which came from a mismatch between one-shot and continuous processes. This is a deficiency we failed to resolve in time for shipping.

From the earliest points in the game's design, we operated under the assumption that resource manipulation is sparse and discrete, for example, every 7 days, the wheat farm produces 6 units of wheat or the bakery consumes 3 wheat and produces 5 bread. This lent itself perfectly to a rule system that triggers on a per-rule timer.

However, fairly late in the process, we realized that this kind of a discrete system was hard for our players to understand. Whether it was because we surfaced it poorly or because their expectations were trained differently by other games, our beta players had difficulty understanding the simulation and what was actually going on, because the activities were so sparse.

When we explored this further, we found that players reacted best when buildings looked like they operated continuously, for example, wheat farm producing wheat at velocity of 0.8 per day, and when its storage fills up, the surplus gets delivered.

Ultimately, we were able to produce much of the desired user effect by essentially faking it in the UI and in how we give feedback to the player. But had this happened earlier in development, we might have rewritten all of our rules to run much more frequently, to simulate continuous production, even at the cost of spending significantly more processing time on rule checks per second. Even better, we should have considered combining it with a form of parallel-reactive networks [Horswill 00], to help represent continuous processes, and hooked that up as part of the data model manipulated by the rule system.

8.6 Related Work

On the game development side, this implementation was very directly influenced by previously published implementation details of Age of Empires (AoE) [Age of Empires 97] and of the GlassBox engine used in SimCity [Willmott 12].

AoE was one of the earliest games to expose data-driven production systems. Their syntax is based on s-expressions, and rules might look something like

```
(defrule
  (can-research-with-escrow ri-hussar)
=>
  (release-escrow food)
  (release-escrow gold)
  (research ri-hussar))
```

The AoE system plays from the perspective of the player, that is, one rule engine is active per enemy player. The GlassBox rules, on the other hand, are much more granular and run from the perspective of each individual unit, for example,

```
unitRule mustardFactory
    rate 10
    global Simoleans in 1
    local YellowMustard in 6
    local EmptyBottle in 1
    local BottleOfMustard out 1
    map Pollution out 5
end
```

We were highly inspired by the design choices from GlassBox, especially the data model that organizes resources into bins, distributes those bins in the game world, and lets production rules check and manipulate them.

Finally, the representation of conditions and actions using a contextual language like "`unit gold > 5`" is related to the history of work on deictic representation, such as the implementation of game-playing AI for the game Pengi by [Agre 87] or reactive autonomous robots in [Horswill 00]. In particular, we decided against inference or queries with arbitrary free variables such as "`is(X,gold-mine) and has-workers(X)`". Instead, we replaced them with task-relevant indexicals, which made fast queries much easier to implement. The task of binding deictic variables can then be moved to a separate subsystem that can be optimized separately (in the Pengi example, it was done by simulating a visual attention system, but in our system, it's trivially easy, based on which entity executes the rule).

8.7 Conclusion

This chapter examined the implementation details of a production rule system used in the game *1849*. We started by examining the architecture of the system, followed by details of production rules and their components. As we demonstrate, a few specific simplifications enabled a very efficient implementation, suitable even for underpowered mobile devices.

References

[Age of Empires 97] Uncredited. 1997. *Age of Empires.* Developed by Ensemble Studios.

[Agre 87] Agre, P.E. and Chapman, D. 1987. Pengi: An implementation of a theory of activity. In *Proceedings of the AAAI-87.* Los Altos, CA: Morgan Kaufmann.

[Forgy 82] Forgy, C. 1982. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence* 19: 17–37.

[Horswill 00] Horswill, I.D., Zubek, R., Khoo, A., Le, C., and Nicholson, S. 2000. The cerebus project. In *Proceedings of the AAAI Fall Symposium on Parallel Cognition and Embodied Agents*, North Falmouth, MA.

[Willmott 12] Willmott, A. 2012. GlassBox: A new simulation architecture. *Game Developers Conference 2012*, San Francisco, CA.