AI-Driven Autoplay Agents for Prelaunch Game Tuning

Igor Borovikov

1 Introduction

Obtaining quantitative gameplay evaluation before the game is ready for open playtests is a common challenge. Simulating human players with AI-driven autoplay agents (or “agents,” for short), however, can provide a practical tool to inform design decisions, including evaluating player progression and in-game economy, discovering exploits and locked states, balancing combat, all before the game is ready to play. Agents also provide a natural way to automate tuning iterations.

At EADP AI Applications (an R&D group at Electronic Arts), we collaborated with game teams to implement agents across various genres. While cutting-edge AI tools like Reinforcement Learning (RL) can contribute to agent-based game tuning, we found that even simple techniques can often drive critical decisions. Starting game design with the agents rather than retrofitting them later speeds up design iterations and result in a better game. This chapter overviews several agent-based methodologies we applied to the gameplay design process and some case studies that illustrate our prelaunch game tuning techniques. We advocate that the agent simulation should be the primary tool for designers and the starting point of any game development.

Though we are focusing on mobile games here, interested readers may also wish to review our previous discussion of open-world games (Borovikov 20, 19a, 19b, 18), which follow a very similar approach.

2 Methodology and the chapter overview

Artificial agents are not a new tool to gaming. Often used to advance science through optimizing game-playing performance, with famous examples including the AI behind *AlphaGo* (Go) and *Deep Blue* (Chess). Exploitative bots, on the other hand, are scripted agents that automate tedious tasks or exploit aspects of a game or gaming ecosystem to the user’s advantage, such as automated harvesting of resources in mobile games. The AI involved in such examples varies from virtually non-existent, as in simple scripted bots, to cutting-edge deep neural nets trained on specialized hardware.

In both cases, gameplay agents rely on the same game interface that is available for humans. The algorithms’ input uses the render buffer, potentially enhanced with object ids, depth information, and other meta-data, but is otherwise visually the same or similar to what a player sees. The game’s input could be a direct emulation of joystick moves, button clicks, and screen taps, often relying on the actual location and behavior of the UI elements. Such low-level interfaces make learning more difficult because of the need to first extract the required information from the imagery before becoming inputs to a policy (state-action mapping). In contrast, when we introduce agents directly into the game-development pipeline, they allow us to expose different methods for agents to communicate with the game and can serve very different goals.

During development, automated game playing is primarily used to evaluate and improve gameplay mechanics and tuning. One of the oldest, non-trivial examples of this in games dates more than two decades back to the PC title *Sid Meier's Alpha Centauri* (SMAC 99), a "4X" game in which players build an empire alongside other human or AI players. In SMAC, developers discovered by accident that the computer players could play each other, thus removing the human player from the equation and fully automating the play process. The result was a valuable source of feedback that could be collected in hours compared to the multiple days it might take a human to play the full game. Today many more tools exist to enhance the utility and efficiency of autonomous play for game development.

From a design perspective, agents can provide quantitative metrics to help assess and balance the play experience. Agents have the benefit of operating at a scale and depth inaccessible to the relatively few (and frequently biased) developers playing the game early in its development. Simulating player personalities with different agents' policies can give insight into the expected player's behavior after the game's release.

For example, a simple greedy agent might measure some of the metrics that characterize less engaged or more casual players. Often, such agents represent relatively shallow activity-driven gameplay and thus are not difficult to model. At the simplest end of the intelligence spectrum, entirely random autoplay backed up by critical observations can be valuable, too. While an optimized agent may be a good representation of a highly motivated player, an activity-driven agent may correspond to a large chunk of the player population and provide reasonable estimates of average progression measures. Probabilistic models of the agents following different policies can directly translate into the corresponding multi-agent simulations. Besides measuring progression speed and comparing various progression paths in the game, agents can also provide a strength comparison for factions (clans, alliances, et cetera) to help estimate the value of combat units both in isolation and in the context of the in-game economy. Agents can substantially facilitate in-game economy analysis, identify limiting resources, and help avoid undesirable tuning extremes.

On the more extreme end, with proper optimization objectives and policies, agents can even simulate a performance-driven player capable of discovering loopholes in the game's design or implementation. Finding the global optimum (or at least a reasonable local one), however, is a non-trivial learning task.

Agents can access a complete game state, the tuning data, and embed hooks into the code, which gives them a distinct technical advantage. Such complete access dramatically facilitates information and flow of control by enabling a high level of abstraction. In the RL context, agents can directly and fully access the Markov Decision Process (MDP), which models the game. Such access simplifies and speeds up the learning of various policies, applying different learning methods, and optimizing diverse objectives.

In this chapter, we start with an investigation of the more fundamental engineering aspects then transition toward a more conceptual design-centric view. By first creating a game model we can simulate various agents' behaviors, which can then be used to quantify and validate the gameplay mechanics before moving on to later development phases. In many respects, such an approach is, in essence, test-driven development (TDD), but at a higher level of abstraction with the designers driving change-validate iterations. Finally, we will explore several case studies to illustrate our agent-based approach as a powerful tool in game design.

3 Instrumented game clients

Instrumenting the game client is an obvious way to set it up for the autoplay agents. Implementing such instrumentation early, vs. as an afterthought, in the development cycle will unlock the benefits of agent-based testing from the very beginning.

3.1 Integrated autoplay agents

Autoplay logic can be directly integrated into the existing code base and then compiled and run in the same executable. Such direct integration is a relatively small amount of work and opens several possibilities, like facilitating the implementation of a hint generator. Experimenting with reinforcement learning directly in the game code, however, is challenging. Supporting agents as part of the game code requires compiling the client and following studio production practices, potentially slowing down experimentation. Running simulations at scale on the game target platform is also rarely practical while collecting data beyond standard telemetry could be cumbersome or even infeasible, as well.

Given the pros and cons, we opted for the integrated client approach for our brief collaboration with a mobile match-3 game. The game team implemented the core autoplay functionality allowing us to get the first practical results quickly. We ranked levels of difficulty with Dynamic Difficulty Adjustment technology in mind (Xue 2017), which allows adjusting the level of challenge depending on player performance. A somewhat unexpected discovery was that the relative difficulty levels did not notably depend on the autoplay heuristic—the *relative* difficulty measured from random autoplay was actually like that obtained from near-optimal agents.

While the integrated autoplay allowed quickly picking some low-hanging fruit, we realized that it is not readily transferable to a project of higher complexity.

3.2 Instrumenting for an external driver

A more robust way to implement agents for a game is to expose controls to an external application. To be able to simulate human players' behavior, the controls should be similar to those available to the player but exposed programmatically through some API. Figure 1 shows a high-level diagram of the agents interacting with an instrumented game client. Such instrumentation can support multiple AI applications, but most closely, it follows a typical training setup for Reinforcement Learning (RL). The game client implements the Markov Decision Process (MDP), and the agent acts using the MDP in lieu of a human player. While the observation-action loop is universally present in such simulations, the agents' objective function will depend on the simulation goals. The instrumentation strategy details can vary depending on the game implementation's specifics, but it can be as simple as the pseudo-code on Listing 1.

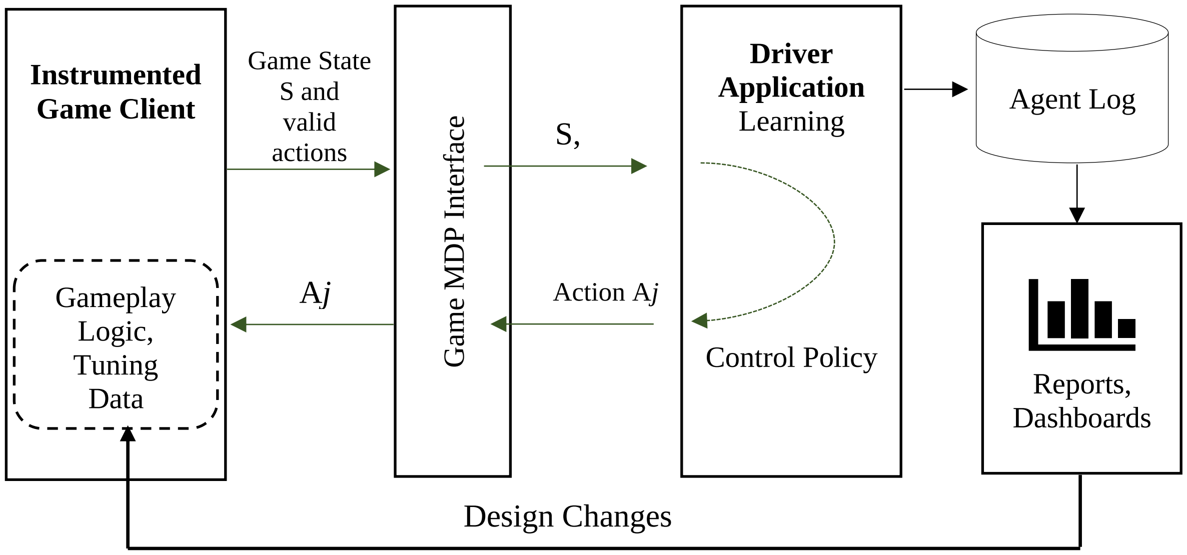

Figure 1 Information and control flow for an instrumented game client.

Listing 1 An instrumented game client update function.

```
void Game::update(…) // called each frame
{
    // Regular handling of game events, user input, etc.
    …
#ifdef AGENTS
    // An RPC-like call to the external driver app and
    // marshalling of the response:
    const Action& action = RequestAction();
    switch (action.id)
    {
        // Handle actions (or no action could be the case)
        // by calling directly into the game API or
        // posting events into the internal events queue
    }
#endif // AGENTS
}
```

To connect Listing 1 and Figure 1, note that the pseudo-code passes a single selected action from the external driver application to the game client via `RequestAction()` with no arguments. This is intentional as providing an instance of the game object to `RequestAction(`*`this`*`)` call creates an additional dependency on the game class interface. Instead, in Figure 1, we have information flow that delivers the agent's game state to the control policy before it computes the next action. The specifics of transmitting state data may vary widely and depend on the concrete features of the game.

One straightforward strategy is to pass state updates to the external agents' application at the same time the client sends telemetry since state changes are much smaller and are easier to communicate compared to a complete state snapshot. That means that the agent must maintain a full description of the game's initial state and can properly update it by reproducing game logic. This is unlikely to become a problem if the initial state is a well-defined, relatively small piece of data and most of the updates have basic logic (e.g., "decrease health by 1"). While the agent can listen directly to the telemetry, we nonetheless found that custom messaging was more effective as we were able to combine a partial state snapshot with an update to validate the agents' execution to simplify reporting. Also, custom messages can be fed directly into database tables for external aggregation and analysis without additional post-processing.

Compiling and passing to the agent a list of available actions given the current game state can require additional work, depending on the game, because not all conceivable actions may be valid at an arbitrary state. It is particularly hard if the actions are not well decoupled from the UI and its state. One practical technique is to infer actions entirely or partially from the tuning data. These data are almost universally available in the game assets as parsable files, usually in JSON or XML format. While the semantics may depend on the gameplay logic, it is rarely ambiguous. The client application could also refuse to take actions not allowed in the current state or other fail-safe functionality. Sending feedback on failure to the driver can also help in debugging and learning valid actions.

Note that in Listing 1, the proposed instrumentation plugs the agents' related logic into the original event handling loop as an optional non-destructive addition. It preserves all of the original event-handling functionality, including capturing and processing user input. That ensures that the instrumented game updates closely correspond to the normal ones and their execution logic remains as unaffected as possible. Preserving user input handling helps with debugging and enables additional experimentation with the agent-driven game client. One application of maintaining user input in place is to play in an automated manner until a particular point and then pass control to a human player. This way, the designers can start the game from a later point without spending valuable time on early gameplay. Including idle action or throttling calls to `RequestAction` may be necessary to ensure correct update logic and proper pumping of the events. Making agents conditionally compiled combines the instrumentation with the base game code while stripping it from the production version as needed.

A convenient way to implement communication between the agent and the game client is via sockets. Separation of the agents' functionality from the base game by instrumenting and exposing RPC-like API allows implementing the external driver and the agents' logic in your favorite programming language. We found that Python-based external driver hosting agents' logic provides a reasonable trade-off between performance, ease of development, and simplicity of integrating agents' code with databases for reporting and learning.

It's worth noting that many practical RL algorithms are easy to implement and need not be cutting-edge or unnecessarily complex to enable a valuable analysis of the gameplay, as we discuss next.

4 Case study 1: Progression paths in The Sims Mobile

In this section, we discuss an example of applying the outlined methodology to a game with discrete states, similar to a board game. Such games have well-defined discrete states with a complete enumeration of valid actions. A discrete setup contrasts with open-world titles where the state-action space has continuous components, and its dimensionality can be vastly higher.

In our case, we chose a very early version of The Sims Mobile (TSM). The gameplay consisted of atomic interactions that approximately corresponded with a player's screen taps. Though modest, the selection of concrete interactions in various contexts required some skill. With that in mind, gameplay difficulty could be reasonably estimated via the number of times a player taps the screen.

Among other activities, at different points in the game, a player must choose one of the available progression paths, build up their sims, upgrade the home lot, collect various items, and work on producing in-game resources. One example of a choice in TSM is selecting a relationship track, each of which has a corresponding strategy to progress, and serves.as a source of player experience (XP) and other rewards.

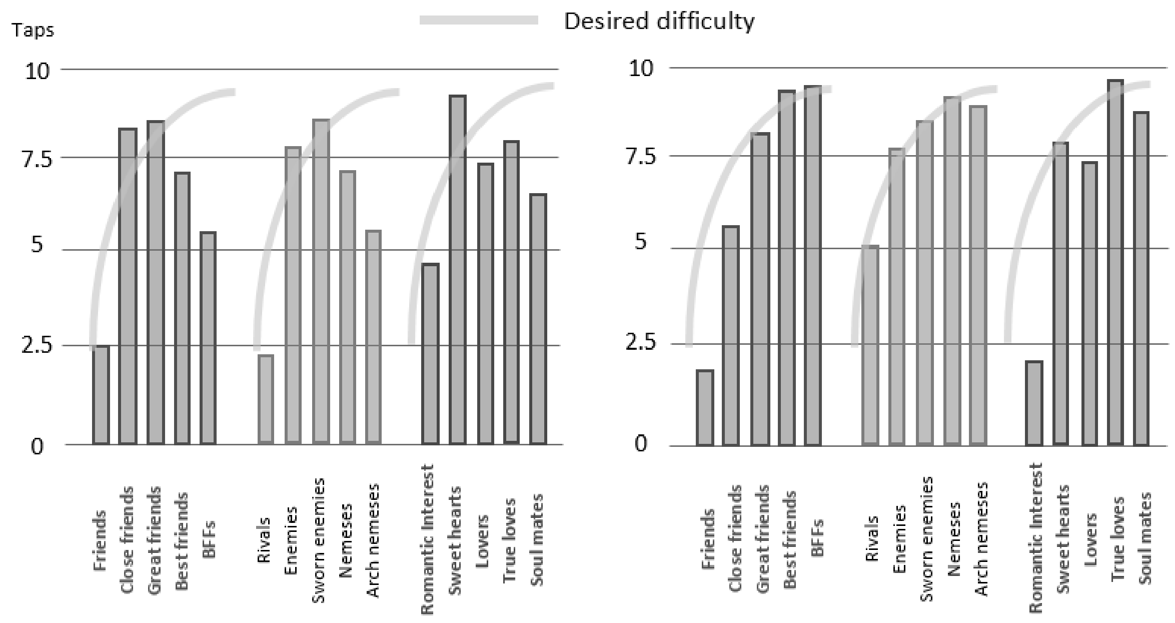

Figure 2 Relative difficulty of the relationship tracks before (left) and after (right) the autoplay agents' feedback. The optimized policy estimated the absolute difficulty (shown), which had the same relative nature as a near-random policy.

As initially designed, the relationship progression had five distinct stages along each of the three tracks (see Figure 2). In an early version of the game, players had to complete a corresponding “event” to progress to the next stage. Events consisted of a series of actions that generated experience points specific to the selected relationship track. While players could attempt events multiple times, a failed attempt resulted in lost progress and wasted resources. An event started with insufficient resources would “fail”, not delivering the full experience and preventing the player from advancing to the next stage. Thus, successful completion of events requires players to devise some strategy for entering them.

While iterating on the gameplay mechanics, a question among designers often arose: Do different relationship tracks provide similar benefits per tap? If not, players would quickly discover any imbalance and concentrate their efforts on the path with greater rewards. This skewed distribution of players would result in a less rich experience for the player as well as less efficient use of design, animation, and tuning efforts spent on the unpopular tracks.

To answer this question, we simulated players’ progression using the instrumented game client described in the previous section. In doing so, one of the most insightful observations was that near-random gameplay with minimal heuristics revealed several new-to-us and vital metrics, even though in relative terms. While the absolute number of taps we obtained was far from what was observed from human gameplay, the relative rate of progression did not notably change after we applied RL and learned more sophisticated gameplay policies. This meant that implementing RL across difficulty levels was not strictly necessary for our purposes and was in line with our observation that the relative difficulty of the match-3 game discussed earlier was invariant to the auto-play heuristic. Figure 2 shows progression speed and difficulty measured for the three relationship tracks we explored. The absolute values on the charts correspond to utility-based learned policy results, discussed next. The near-random gameplay generated similar graphs, albeit with a different scale on the Y-axis, showing taps.

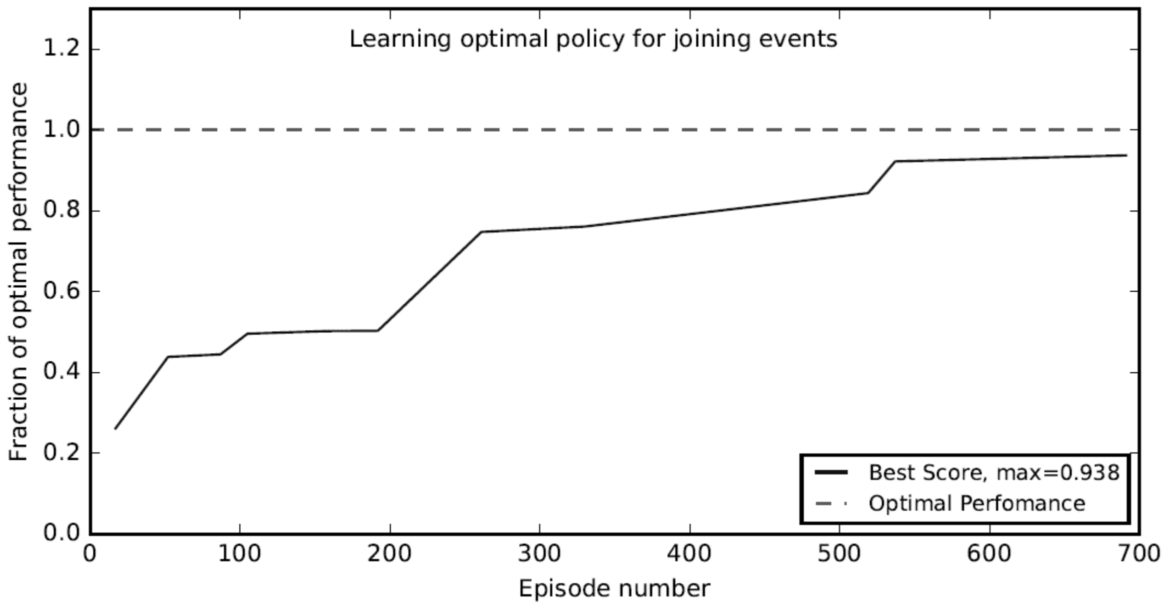

Figure 3 Learning optimal policy for joining events to evaluate the difficulty of relationship tracks. A black box stochastic optimization allowed us to learn weights for the utility function using parallel execution on several cloud nodes in 700 episodes. The step-wise shape of the plot shows the performance of the best learned so far policy.

Our initial learnings exposed and quantified two otherwise subjective problems experienced in manual play. In particular, the event difficulty for relationships did not follow the desired pattern of gradual increase shown by grey curves in Figure 2. The second stage (“sweethearts”) is substantially more challenging than desired and more challenging than the second stage on other tracks, peaking the difficulty too early in the relationship stages’ progression. As a result of this feedback, we were able to tune the game to take care of most of the discovered issues by the next iteration and match the target curves better at a qualitative level. The match we aimed at was about the progressive difficulty and comparable difficulty across the tracks.

In our first implementation, the changes in tuning were done manually by designers. With manual tuning updates, however, achieving a numerical match to the target curves is very difficult. Returning for a minute to Figure 1, note the arrow going from the agents’ reports and dashboards back to the tuning data and gameplay logic that indicates tuning parameters. Nothing prevents automating this feedback loop. The collected metrics can drive the automatic optimization of specific objectives provided by the designers. In the example of the relationship track, we could aim to fit the design’s target difficulty (i.e., desired number of taps). The iteration time would be much faster than manual tuning changes and enable a tighter fit to the objectives. Though we did not automate feedback for TSM, we did do so for a different game, discussed later.

While the random policy was qualitatively useful, we still needed more accurate quantitative metrics that were similar to human players. That became possible by introducing a simple version of RL with the goal to discover the optimal strategy of completing relationship events. As mentioned, players ideally need to accumulate resources through completing less rewarding activities, a version of “delayed gratification”. In doing so, the player would initiate the event, and given sufficient resources would have an opportunity to finish it with maximum rewards.

While the game state space was relatively small and could be tracked using table-based learning, tabular methods may not scale with future design iterations. And the tabular definition of an action-value function is not an “organic” model of the logic behind human players' decisions. Arguably, humans develop an intuitive feel for the action values which is similar to a contextual utility function. To model that, we used a utility function approximating value U of an action a depending on the current game state s, expressed via rewards *R* and costs *C* as follows:

$$U(s, a) = R(a) * v(s) + C(a) * w(s)$$

For each action a, the game tuning explicitly specified reward *R* and cost *C* in terms of the resources gains and drains, experience points earned, and events triggered. The game state’s dependence comes as the learned weight functions *v(s)* and *w(s)* modifying actions’ utility. The state *s* included several commodities like energy, hunger, and an event indicator (0 for outside of the event and 1 otherwise) wrapped into a vector. We selected both *v(s)* and *w(s)* to be linear functions with coefficients p and q that we had to learn:

$$v = p * s;\ w = q * s$$

To pick the next action, we used the SoftMax algorithm, a probabilistic decision rule with the probability of selecting an action a_i being proportional to the exponent of its utility:

$$\sigma(a_i, s) = \frac{\exp(U(a_i, s) * \tau)}{\sum_j \exp\left(U\left(a_j, s\right) * \tau\right)}$$

Here $\tau = 1/T$ is the inverse of the “temperature” *T* parameter. A higher temperature results in the distribution of probabilities over actions close to the uniform, like in a random policy. Correspondingly, a lower temperature increases preference for the action with higher utility. Thus, the temperature may be interpreted as player skill.

The optimal policy’s objective was to complete all events and reach the highest available relationship level in the least number of taps. Capturing that objective required rewards engineering. Experimentally, we discovered that the following expression for the total reward $\mathcal{R}$ per episode worked well as the optimization objective:

$$\mathcal{R} = \frac{r(r + \epsilon)}{(a + \epsilon)} \rightarrow max$$

Here *r* is the number of rewarded events, a is the number of all attempted events during the episode, and ε is a small number (less than 1) to eliminate division by zero when the policy did not try to start any event. Intuitively, the total reward expression penalizes a lack of attempts to initiate an event (denominator) and quadratically rewards the number of completed events (numerator). Other reward functions would work provided they encourage starting events, penalize unsuccessful ones, and reward better paths to completion.

Due to the discrete nature of our learning problem, optimizing $\mathcal{R}$ using gradient methods would fail, especially at lower temperatures. Finding gradient direction requires randomization of the weights we are trying to learn, whereas the graph of our function contains “flat” terraces that would interfere with this process. It turned out that black-box optimization with artificial noise added to the parameters, now popularized as Evolutionary Strategies (ES) (Salimans 17), worked very well. One of the advantages of ES optimization is the ease of using several cloud nodes to perform optimization. To ensure convergence, we had to tune meta-parameters, the initial values for the temperature, and the initial radius for the Gaussian sampler, as well as their decrease rate. Finding a practical combination of these parameters was straightforward, thanks to running simulations on several cloud nodes. In only 700 episodes the utility-based approach successfully learned an efficient policy for joining events (Figure 3). The results in Figure 2 are the metrics obtained from the optimized agent and were qualitatively in line with the results from the game-balancing specialists.

We found that utility-based policy performance was close to the best theoretical limit directly estimated from the game tuning data. Unfortunately, this was not practically achievable given our approach. Still, finding global optimum is critical in evaluating a game for exploits and search-based methods can result in globally optimal policies that point to design loopholes, at least in principle. With this in mind, we revisit TSM with A-star as the primary search method for computing optimal gameplay in section 6.2.

The following section describes a more practical alternative and looks at why agents-based simulation should come first in the development process.

5 Case study 2: Modular agents, rock-paper-scissors, and combat evaluation

This section looks at an application of agents leveraging natural modularization engineered from the beginning of the game development. Our case study is based on mobile a 4X MMO. Unlike an activity-based game like TSM, the 4X genre assumes highly competitive gameplay. We look at the problem of balancing the classic rock-paper-scissors (RPS) dynamics with the game economy in mind.

The game implemented a standard client-server architecture, with the local client facilitating the player’s interaction with the game and any game logic is executed on the server. There is no simulation happening on the mobile device. As is typical for the 4X genre, players send armies consisting of units created on their bases to fight other players who belong to an opposing faction or clan. Since the combat logic is naturally separable from the rest of the gameplay, the game team implemented a wrapper to allow combat simulation in a standalone application while sharing the server’s codebase. Our goal was to facilitate tuning of combat units and general player progression.

The agents in our exploration communicated with the game using two different methods. One way to interact is using the server API, which already had exposed web sockets and REST-ful calls. That required implementing headless client incorporating general agents' logic, corresponding to the "driver" in Figure 1. Such agents conceptually would not differ from our experience with the instrumented game discussed in the previous sections, except we did not have to instrument the server code. The other kind of interaction was via combat simulation, which we still had to instrument like in Figure 1. In this section, we discuss combat simulation as an example of a modular agent-driven approach to design. Breaking a monolith black-box simulation of an entire game into its sub-systems provided a much more efficient way to answer combat-specific questions related to combat units' balancing.

5.1 Combat mechanics

Game combat involves two fundamental types of mechanics: The Lanchester Laws, and the Rock-Paper-Scissors (RPS) power cycle. Seemingly, tuning same-type units is trivial but it goes against the intuition as soon as units have different powers due to the nature of the Lanchester Law. Moving from same-style units to RPS brings in even more challenges.

To illustrate, we start with an informal introduction to the Lanchester Law of combat attrition. The main parameter of Lanchester Law is power, which is discriminating between "ancient" and "modern" combat. In an ancient battle, units interact on a one-on-one basis, while in modern warfare, all units fire at and can receive damage from all participants of the opposing side. The first case results in linear power in Lanchester equations of attrition, the other is quadratic.

The linear law is straightforward. In ancient combat, an army's power grows linearly with its size assuming equal strength of units. For the quadratic case, however, we must deal with the size/power multiplier squared. As an illustration, consider one unit with quadrupled power. Somewhat unexpectedly, it requires only two regular power units to achieve a tie, not four. It is easy to see that the stronger unit receives fire from two units, making the damage multiplier equal to 2. But the fire it returns also spreads over two units, hence the opponent damage multiplier is actually ½. We get a quadrupled attrition rate for the stronger unit by combining those, compared to the opponent units. With the law's quadratic version, it is easy to misjudge the game balance by implying intuitively "obvious" (but incorrect) multiplier 2 in the quadratic case. With a concrete game fiction Lore, the Lanchester law may become very different from the classical ones, which designers would have no means to analyze directly. Analysis of attrition with agent-based simulations could get meaningful insights into the balance of units' powers.

The second critical component in tuning combat is an RPS-style specialization of units. The RPS mechanics is a venerable tool in combat units' design, though it can come in many flavors and is usually multi-tiered. The RPS tuning objective is to assure that all combat units (or, at least, types of units) are essential for successful combat when the opposing army's composition is unknown in advance. In traditional RPS gameplay, pure strategies (i.e., homogeneous armies) are not the best choices for the player, as the opponent can easily counteract them after their discovery. As such, we need to ensure that the agent-based simulation generates "balanced" armies. One of the aspects of such verification is proof that there are no over- or under-powered units. Underpowered units would be impractical to build, while overpowered units would be preferred, thus reducing the gameplay richness. The next subsection explores the problem of unit balance.

5.2 RPS balance evaluation

A starting point for evaluating RPS mechanics is computing units' relative combat power via their matchups. A matchup of two unit types from two opposing factions requires constructing homogeneous armies for each unit type and determining which one wins in a battle against each other. The ratio of unit counts resulting in a tie gives the relative dominance (power) of units, which we will call the simple power ratio. Besides the simple power ratio, we also need to account for the limited capacity of the armies. If army size is the limiting factor, we need to compute the power ratio per unit size instead of the naïve per-count value.

Additionally, we want to capture the connection of RPS tuning to the in-game economy. In short, we want to ensure the absence of both dominant and inferior units from the production cost (or production time) perspective. Note that the construction time usually equates to some in-game currency costs via rush interaction. Thus, there are at least three metrics for RPS evaluation: (1) by count, (2) by unit size, and (3) by construction costs.

Computing each of these metrics is a straightforward task. In the context of agents, the combat simulation represents an isolated sub-process of the game MDP. The agents' policies limit them to the construction of homogenous armies only. With a fixed homogenous army of an opponent, the agent's objective is to find a minimal homogenous army of the selected unit type that ties (or barely wins) in the matchup. The results of such matchups come in the form of three $N \times N$ "dominance matrixes" for the ratios of the count, size, and costs of units, correspondingly. Here N is the total number of unit types, usually the same for each faction or clan.

Next, we interpreted these matrices as a complete bipartite graph, which we call a "dominance graph" with the nodes representing units. Naturally, the units of opposing factions belong to the two opposite parts of the graph. The weight of the directed edges is equal to the power ratio of the connected units. The edges point from a more powerful unit to the less powerful one (an arbitrary choice that can be inverted). With this, we can use graph-based algorithms to characterize the RPS relationship.

The first general observation is that a cycle of a bipartite graph is always of even length (Skiena 90) and represents one of the RPS-style cycles: each unit in the RPS cycle is stronger than the next one in the order of the cycle traversal. If all nodes in a dominance graph belong to some cycle, we call it a complete RPS graph. For the analysis, we used the NetworkX Python package implementing Johnson's algorithm for discovering simple cycles. By definition, a simple cycle does not visit the same node twice other than the starting and ending node.

Our case-study game relied on multi-tiered units of three distinct power types, like in classical RPS. We found that the original dominance relationship is a complete RPS for the two most important metrics, unit size, and unit cost. This suggested that even lower-tier units could compete with the higher-tier units in combat when the limiting factors are production costs or the army size. It does not guarantee that a beginner could challenge the more advanced player in the game, though beginners' alliances could get a good run for their efforts even facing a higher-level player in combat. It is a useful feature ensuring rewarding experiences for the latecomers' joining a mature MMO with a population of established older, more powerful players.

Measuring units' dominance is only a small part of the potential contribution that agents can offer to combat tuning. Using a simple power ratio, we discovered non-RPS units that did not belong to any simple cycle in the dominance graph. Forcing them into some RPS cycle with manual tuning changes could be extremely challenging. Automating the loop of changing-testing for improved metrics, however, is precisely where the AI-driven approach can provide substantial benefits.

For example, we can interpret the units' dominance relationship as a "potential function" *U* on the dominance graph. If we traverse an RPS cycle, the potential *U* will always monotonically change until we reach the node where we started. Yet for a non-RPS unit, by definition, we *cannot* construct a cycle with such a property. Using potentials, though, we can instead measure the "RPS defect" for non-RPS unit *U* and build depth-first search (DFS) trees up and down the dominance relationship. We can then combine these trees into the single dominance tree rooted at the node *n* in question.

Next, we consider each pair *(a, b)* of opposing bipartite nodes in the dominance tree and observe the potential change along the edge connecting the pair. The change of potential *P=U(a) – U(b)* will violate monotonicity along the path connecting the DFS tree nodes. We call *P*'s absolute value along the edge *(a, b)* an "RPS defect," conditioned on the node *n*. The minimum value of the defects *P* across all such edges is the "RPS defect" of the unit *n*. Intuitively it describes how close we are to include *n* into an RPS cycle by flipping dominance in the least "defective" edge *(a, b)*.

By quantifying the defect, we can use optimization to find the most conservative (by some metric) tuning changes that will eliminate defects for all non-RPS units. The optimization would change the tuning of the units involved to minimize and eventually eliminate the RPS defect. This approach can result in a practical solution that removes the painful trial-and-error loop of manual tuning of the units' parameters while chasing the complete RPS balance.

5.3 Asymmetric units tuning

So far, we have assumed that conflicting sides are symmetrical, but that is not always the case. It is unlikely, for example, that Orcs will exactly mirror Elves in combat. Though asymmetry makes achieving a complete RPS even more challenging, we can tackle this problem in a similar fashion to the RPS defects minimization just described. Also, we can extend the analysis of asymmetric tuning directly to non-homogeneous armies.

One useful trick is to simplify asymmetric tuning by correlating or grouping unit characteristics to reduce the dimensionality of the search space. Consider, for example, the relationship between army size and attack power. Naturally, to preserve balance, a faction with larger unit power per unit should have a lower cap on the allowed army size. Figure 4 demonstrates how army size must be larger if it provides more attack power than a would-be symmetric counterpart unit. In other words, it shows a conversion rate of the unit size to the unit attack power that preserves the balance of factions.

The solid line in Figure 4 is an average of multiple stochastic simulations. In each simulation, we constructed mixed armies (i.e., armies can contain all unit types available to the player) and matched them up against the opponent's armies of the same size. Since we sampled only a limited number of armies, the individual curves contain noise. But their average shows linear dependence with high confidence, which is a useful observation. Generally, the strength-size dependence does not have to be linear, of course, but discovering the conversion curve is another example where agents can close the loop for automatic tuning.

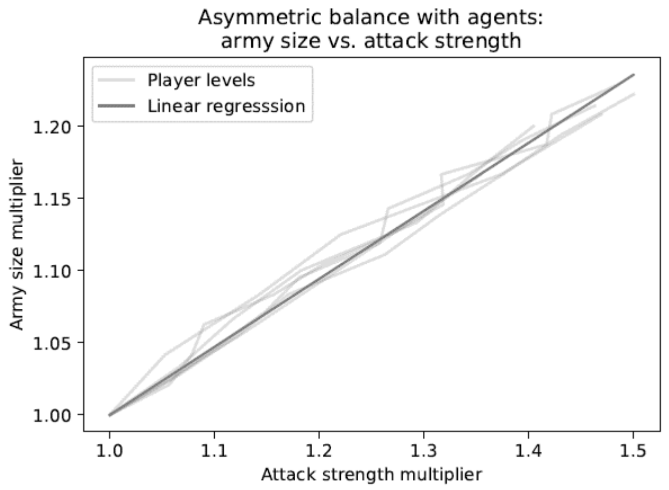

Figure 4 Asymmetric tuning with agents. The player can build more massive armies using weaker units to balance the opponent's units' stronger attack power. The grey lines correspond to player levels and show the army scale obtained from agents' simulation. The agents' objective is to achieve a tie in combat. The solid bold line is linear regression fitted to the tie data points.

5.4 Nash equilibrium with modular agents

Homogeneous armies are non-dominant in RPS scenarios given repeated interactions and assuming no prior knowledge of the opponent, forcing the player to instead explore heterogeneous composition strategies. In such a situation, the Nash equilibrium can be useful in identifying the optimal army composition for players. Still, even in simple games, the exact computation of a Nash equilibrium may be impractical due to its PPAD complexity (Nisan 07). It is worth noting that players nonetheless quickly discover the optimal ratio of unit types that constitute a Nash equilibrium via repeated trials, exposing this as a practical method to approximate a Nash equilibrium using agents and constructing "best responses."

Simulating how players discover the optimal composition of their armies with agents is straightforward. At the high level, we construct a population of armies of different compositions and observe their win-loss ratio. We then do a standard iteration through evolution by eliminating underperforming armies and randomly mutating the composition of the winning armies. With the right stimulation parameters, the iterations converge to a composition that statistically performs best against a random army of the opponent. Making this process computationally efficient requires some additional details which we outline with the algorithm in Listing 2.

Listing 2. Approximate a Nash equilibrium via evolutionary best response

```
Army size = Entry-level army size
for each faction:
     generate armies’ population with KDE sampler
     constrained by the army size
     while army size < maximum size:
          for random pairs of armies from opposing factions:
               compute combat outcomes

          eliminate defeated armies from the populations

          army size += army size increment
          for each faction:
               resample armies’ populations constrained by
               both:
                    1. The new army size
                    2. Current army powers compositions
```

The above algorithm starts with smaller armies, which imply smaller combinatorial complexity, and shorter battles. This way, we get an estimate of the equilibrium early in the simulations and we gain better precision in the later iterations because the collected metrics become more granular given the gradual increase in army size over time. We also mimic players’ progress in the game. In the internal loop, the algorithm in Listing 2 keeps the armies at equal sizes for the opposing sides, runs random battles between opponents, and eliminates losing armies. It is worth noting that the elimination of defeated armies happens at a rate that does not impact the population too radically. Overly aggressive removal of defeated armies leads to composition oscillations, similar to switching between pure strategies in the repeated play.

To maintain population size, we add new armies generated by “mutating” more successful ones, like in Genetic Algorithms. The mutation happens via sampling armies using Kernel Density Estimator (KDE) with Gaussian kernels centered around victorious armies. The count of sampled centers for KDE corresponds to the same armies’ count in the current population. The army size increase also happens by generating new samples that fit the new size precisely by rounding the powers ratio to the exact units count. For convergence, the kernel radius needs to become small enough to keep armies from spawning far from the successful “parent.” Figure 5 shows a computed approximate Nash equilibrium. The identified army compositions can guide units’ production in the in-game economy simulations.

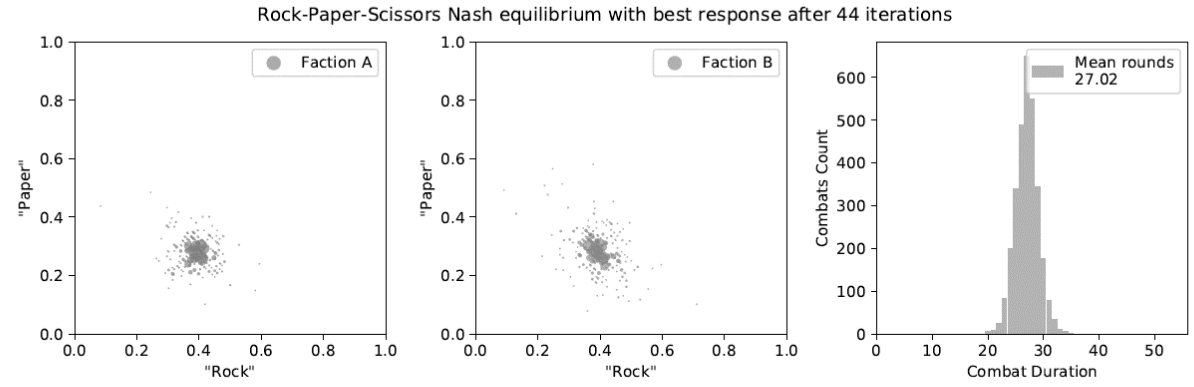

Figure 5 Approximate Nash equilibrium found with the best response evolutionary algorithm (only two axes are shown). Factions have symmetric units with unequal RPS powers, leading to an approximately 4:3:3 ratio of the optimal army types. Combat duration (measured in “atomic” interactions required to complete the combat) predicts the server load for the anticipated optimal players’ behavior.

To conclude this section, we emphasize that AI agents can help solve tuning tasks in a modular game architecture very efficiently. We illustrated that with combat simulation and unit tuning in the RPS game mechanics. With modularity enforced in the gameplay logic, we can obtain extra mileage for agent-driven analysis. Modules expose better-defined API and are better suitable for the agents’ instrumentation than monolithic gameplay implementation. The results from the “modular” agents can be aggregated and used as probabilistic models in the complete game instead of running full live simulations. Such a bottom-up approach can reduce the complexity of the entire game simulation. Closing and automating the tuning loop is another goal, which is easier to achieve with a modular design.

6 Simulating gameplay outside of the game

While an instrumented game client helps to ensure the correct logic of agents’ interaction with the game, it has some drawbacks. The main disadvantage is relatively low performance and a high load on CPU and graphics. A headless client, such as the combat simulator from the previous section, can alleviate this issue. There the simulator encapsulates an essential part of the gameplay logic in a compact, high-performance application. In the absence of the headless client, however, we need more capable (hence, more expensive) cloud nodes for massive agents’ simulations.

In this context, an issue more critical than CPU and GPU cycles is how the game manages simulation time. Many games can support a simulation speed multiplier, but this has limitations due to the main update loop requirements. Executing simulations at only a low multiple speed of the average human will significantly reduce the practicality of such simulations, and may create syncing issues, such as with physics simulation. If the algorithm cannot accurately approximate changes over larger time increments, it will generate incorrect results that potentially affect gameplay. That is true for any system that makes an implicit assumption of sufficiently small fixed-size time increments per update. Finally, severe limitations arise from the necessity to load and save game states if we want to perform a look-ahead. In short, the game client carries excess baggage in the form of constraints hindering agents' optimal application.

We can address most of these issues if we can implement the gameplay logic outside of the game application. Such an implementation must not require graphics and must implement state switching without disk operations, i.e., can store and switch look-ahead states in memory. Finally, it must also be able to perform events-based simulations without explicitly simulating time increments.

Given this, sometimes the work to instrument a game client not designed for agents can outweigh the work to extract the necessary game logic directly from the game by following the design documents as well as the practical experience of playing the game. This latter approach is similar to the rough simulations that designers run in their spreadsheets. Additionally, the instrumentation of the client is affected by continuous changes to the game code and can require considerable effort to stay properly integrated with the agents. We learned that the cost of ongoing maintenance for the external application was less work, even if we factor in the additional scrutiny necessary for logic validation.

Games with more discrete decision spaces are much easier to simulate in the described manner. The meta-game (i.e., the gameplay outside of real-time shooting and open-world exploration) of the open-world games can be a valid candidate for such an approach, too. This section describes our experience of simulating gameplay in a dedicated simulator created from the tuning data and gameplay description rather than from an integrated instrumented game client.

6.1 Agents for 4X genre

One of the design objectives in tuning our 4X game is to achieve the desired progression speed via adjusting resource generation and spending by player level. The early game should provide sufficient rewards and fast progression to keep the new players engaged. The late game is usually (exponentially) more difficult and slower in leveling up. The ideal difficulty curve keeps highly committed players in the game without losing them prematurely after reaching the "end game."

A conventional approach to evaluating progression and in-game economy is authoring all the tuning data in spreadsheets and computing simple formulas for progression speed. Such equations usually cover a single scenario of a highly engaged player with maximized progression and resource gains and assume a unique strategy of gameplaying. For less engaged players, the metrics are simply scaled-down versions of the highly engaged players. This can give reasonable estimates but is limited in the ability to explore different progression paths, find exploits, or consider player personalities or play styles. To address this, it is reasonably straightforward to convert such spreadsheets into AI-driven simulations. This naturally introduces player-to-player interaction, different policies, and the opportunity to apply RL, including search-based techniques, to discover optimal or near-optimal paths. Additionally, we more easily simulate various distributions of players, such as by level.

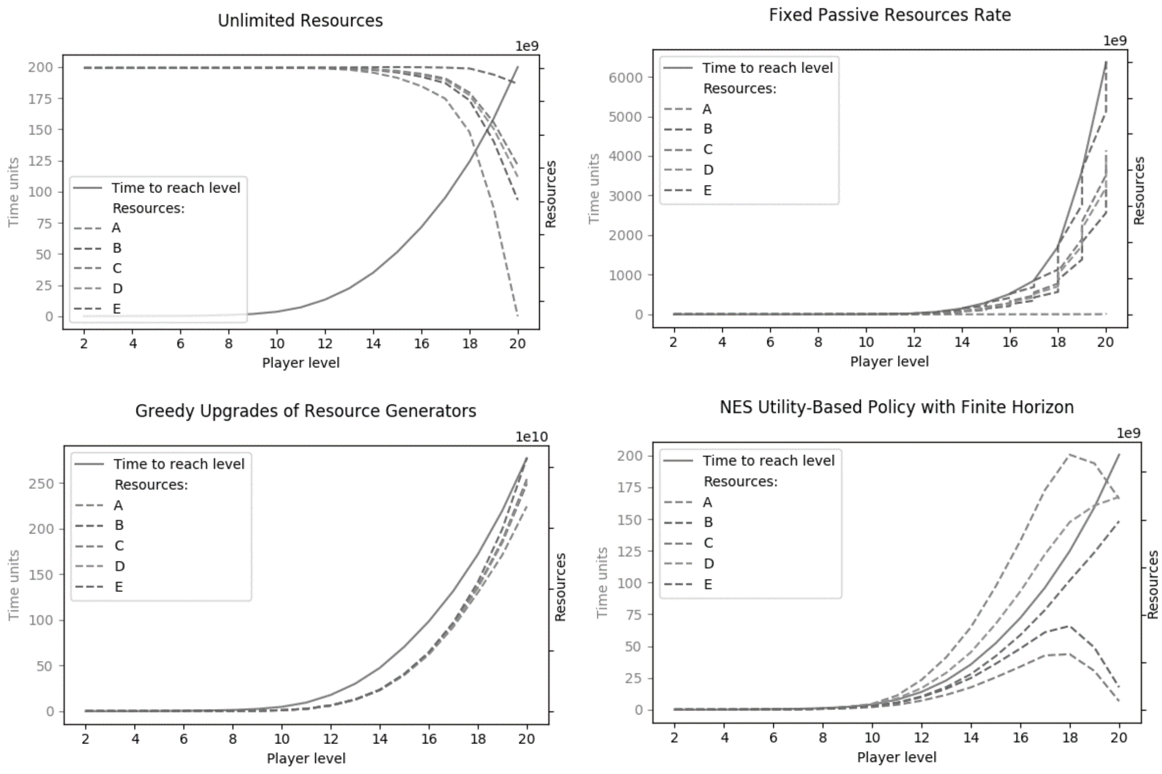

Figure 6 A qualitative comparison of layer progression with a standalone economy simulator. Top-left: unlimited resources scenario sets the fastest progression limit. At the late game, the resources balance goes down, potentially into the negative, which is allowed by the unlimited resource assumption. Meaning, the passive generation without upgrades is not covering the resources use. Top-right: passive resources generation with no upgrades for resource generators. It is an unlikely scenario that shows the slowest possible progression. Bottom-left: greedy policy for upgrading resources generators is only 25% slower than the fastest progression. Bottom-right: finite horizon allows an early stop for upgrades. The policy discovered with NES utility-based agents is very close to the maximum possible progression speed. Resource A is the most limiting resource in these simulations.

Consider the utility model for upgrading base buildings in a 4X game. Updating and adding new buildings to the player base is an essential part of the player activity, and strategically choosing which building to upgrade can increase progression speed considerably. Alternatively, buildings that generate less limited resources can stop on the upgrade path early if the player has a finite horizon within the game, such as victory conditions that do not depend on a particular resource. To illustrate this relationship, we consider a naïve greedy utility-based policy where buildings’ utility depends on the scarcity of the resources they produce. The plots in Figure 6 show the player level progression with (1) unlimited resources, (2) passive resource generation while using naïve upgrade strategy, (3) greedy strategy, and (4) optimized utility-based strategy.

For our utility-based optimization, we computed building utility for the optimal policy using the current building level and player level. Since the upgrade loop dependencies do not change from level to level, we assumed that utility depends only on the difference between the player level and the building level. Learning weight for these differences turned out to be straightforward with ES (Evolutionary Strategies, discussed previously) and revealed a superior policy to the naïve greedy one. In one of the experiments, shown in the bottom right in Figure 6, we made some resources less limiting for the player and assumed a finite horizon for the player. The result was an early stop for the upgrades of some buildings for the corresponding resource-generators. Of course, such a scenario is not desirable from a design point of view. Identifying such a tuning problem using RL helps to eliminate the resources imbalance.

6.2 Search based method in The Sims Mobile

Running simulations entirely outside of the game client also opens an opportunity to apply search-based algorithms, like A-star. We implemented A-star to discover various optimal policies for an early version of The Sims Mobile (TSM) discussed in section 3 (Silva 18). Given an admissible heuristic and a proper objective, e.g., progression path "length" or "cost," we can discover the optimal strategy that is more difficult and expensive to uncover with the mainstream RL approaches like Q-learning. One of the advantages of having a globally optimal policy is that comparing the progress for different activity-based game paths is more representative than an evaluation based on other, non-global gameplaying policies. It is also guaranteed to discover exploits as they must be part of the globally optimal solution.

Since A-star is deterministic, we can run a single simulation to describe optimal progression, unlike in soft-max MCMC or other iterative algorithms; such stochastic algorithms require many simulations to provide statistically significant results. While being generally more efficient, search-based simulation can slow down significantly due to the high level of degeneration of the actions available to a player. Specifically, multiple actions might differ only visually but would take the same amount of time, cost the same, and reward equally. Such multiplicity would inflate A-star's priority queue as such action's contribution to the costs would be equal. Hand-tuning of the heuristic (e.g., adding small preferences to break the tie between actions) could alleviate this situation but may also skew results. We decided to pay the higher computational costs and achieved a practical trade-off by limiting the allowed size of the priority queue in some simulations. This reduced our memory footprint but violated the theoretical guarantees of optimality. The result was a near-optimal policy, which was still competitive or better than ES learning on utility functions. A detailed account of this exploration appeared in (Silva 18).

To conclude this short section, we emphasize that simulating logic outside of the game offers many advantages. It is especially true when we rely on advanced simulation techniques beyond simple spreadsheet computations. Besides just inherently faster simulation, we gain the ability to save and load game states efficiently, enabling search-based algorithms like A-star. Search can discover globally optimal policies and has a higher power to detect exploits or tuning oversights. The downside is the necessity to reproduce the gameplay logic and ensure the new implementation's correctness. We can alleviate that problem by creating the gameplay model first before its full-scale development.

7 Leveraging agents as a primary design tool

Our approach to agents was driven by technical feasibility and the considerations of a game already under development. We also looked for more efficient ways to introduce agents and increase their feasibility as a designer's primary tool. The integrated, instrumented game client was a big step towards better practices, while moving to a modularized representation of the gameplay, providing the additional advantage of testing the design balance by sub-systems. The final step was performing gameplay simulation entirely outside of the game itself. Doing so proved to be the most promising, flexible, and robust method to apply agents. We also explored several applications of the AI-driven agents in the game development pipeline that we're able to deliver benefits for gameplay modeling and balance even when introduced late in the development process.

Ultimately, we found that if game design starts with the simulator and the agents, the designers will have more powerful tools from the beginning and throughout the development process. Given the advantages of the agent-enabled design, we advocate a systematic bottom-up approach to a game development process that starts a game project by developing the simulation agents' framework encapsulating the central gameplay concepts. It is especially valuable when this is completed before the actual game client or server is implemented. Having a framework capable of code generation for including directly in the game itself would eliminate repeating work on programming the modeled features. In short, having gameplay logic implemented in a simulator driven by agents could be the best starting point for making game design testable.

8 Summary

While conceptually like spreadsheets, agents nonetheless introduce a powerful approach to game modeling and tuning, allowing the execution of many possible scenarios to learn optimal strategies. Even better, agents can close the change-test loop and automate tuning, which can optimize objectives specified by the designers while staying within imposed constraints. Where historically such approaches were infeasible, today massive cloud-based computing makes such simulations entirely practical and even simple modeling and RL can generate valuable results. Search-based methods also may be available from the beginning of the design process. The computational power available to the developers at the current state of the industry enables all spectrum of AI algorithms to drive the proposed agents: from the most straightforward to the cutting-edge, sophisticated algorithms. We anticipate that agents-leveraging AI will become an indispensable tool in the game design process.

9 References

[Borovikov 18] I Borovikov, et al., "Imitation Learning via Bootstrapped Demonstrations in an Open-World Video Game." Workshop on Reinforcement Learning under Partial Observability, NeurIPS 2018.

[Borovikov 19a] I. Borovikov, et al., "Towards Interactive Training of Non-Player

Characters in Video Games," June 2019, ICML 2019 Workshop on Human in the Loop Learning.

[Borovikov 19b] I Borovikov, et al., "From Demonstrations and Knowledge Engineering to a DNN Agent in a Modern Open-World Video Game," AAAI-Make 2019.

[Borovikov 20] I. Borovikov "Imitation Learning: Building Practical Agents to Test and Explore a First-Person Shooter," GDC, 2020.

[Salimans 17] T. Salimans, et al., Evolution Strategies as a Scalable Alternative to Reinforcement Learning, https://arxiv.org/abs/1703.03864, arXiv:1703.03864v2 [stat.ML]

[Silva 18] Fernando De Mesentier Silva, et al., "Exploring Gameplay with AI Agents." AIIDE, 2018.

[Skiena 90] Skiena, S. "Coloring Bipartite Graphs." Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica. MA: Addison-Wesley, p. 213, 1990.

[SMAC 99] Sid Meier's Alien Crossfire, Published by Electronic Arts, The official expansion pack for Sid Meier's Alpha Centauri, Firaxis Games, PC.

[Xue 17] S. Xue et al., Dynamic Difficulty Adjustment for Maximized Engagement in Digital Games, WWW 17, 26th International Conference on World Wide Web, 465-471.