Battlecode 2026 Postmortem

Urav Tanna, Seth Lifland, and Andre Mao
of **Generalized Stroke's Theorem**

March 7th, 2026

Contents

1 Introduction **2**
1.1 What is Battlecode? . . . 2
1.2 2026 Overview . . . 2

2 Structure and Modularity **2**

3 Communication **3**
3.1 Shared Array . . . 3
3.2 Squeaks . . . 3

4 Pathfinding **3**
4.1 Pre-2026: BFS Begins . . . 4
4.1.1 "Unrolled BFS" . . . 4
4.1.2 Bug Nav . . . 4
4.1.3 BugBFS . . . 4
4.2 Adapting for 2026: The Dirt Plight . . . 5
4.2.1 Start → Sprint 1 . . . 5
4.2.2 Sprint 1 → Sprint 2 . . . 5
4.2.3 Sprint 2 → US Qualifiers . . . 5
4.2.4 US Qualifiers → Finals . . . 6

5 Macro **6**
5.1 Improving Macro . . . 7
5.2 Baby Rat Taxi . . . 7

6 Micro **7**
6.1 Information Maximization . . . 8
6.2 Cat Micro . . . 8
6.3 Throwing . . . 9
6.4 Improving Micro . . . 9

7 Conclusion and Reflection **9**

1 Introduction

We (Seth, Urav, and Andre) are all currently 2nd year students at the University of Virginia. This is Seth and Urav's fifth year competing in Battlecode and Andre's first year; this was also the second year we qualified as finalists and ended up placing 2nd in the final tournament. We had an amazing experience and are truly passionate about Battlecode, so we wanted to share our strategies and thought process for this year's tournament. The code for our bot can be found here.

1.1 What is Battlecode?

From the Battlecode website - "In Battlecode, two teams of virtual robots roam the screen, managing resources and executing different offensive strategies against each other." Teams from around the world participate by coding, refining, and competing over a three-week period in January. Hourly ranked scrimmages take place, and a live leaderboard is continuously updated.

1.2 2026 Overview

This year's game was titled "Uneasy Alliances". It was the first to feature non-player robots, with cats being an additional neutral threat. There was an option to work together to defeat the cats, but most games would typically end in one team backstabbing another before the cats were destroyed. There were only two units, Baby Rats and Rat Kings with the goal being to either destroy all cats, or destroy all enemy Rat Kings.

2 Structure and Modularity

A recurring pain point from previous years was a lack of modularity, which severely hampered our ability to iterate during the high-pressure final stages of the competition. To combat this, I implemented the Strategy design pattern (thrilling stuff, I know). Our core architectural framework revolves around two primary interfaces: the **State Updater** and the **Behavior** interface.

Think of State Updaters as the macro-level decision makers. At the beginning of each turn, the State Updater analyzes the current sensed data and determines the appropriate behavior for the Baby Rat to execute. We developed discrete Behavior implementations for tasks such as micro, exploration, and cheese scavenging. Once a behavior is selected, its **.execute()** method is called to run the corresponding logic for that turn. Each behavior is designed as a singleton, allowing it to maintain persistent data even as the bot transitions between different states.

While the jury is still out on whether this significantly increased our iteration speed, it undeniably allowed us to experiment with new ideas while keeping the codebase organized. This architecture saved us from the 3000-line, non-codegen

files that plagued us in previous years, which I believe helped us maintain iteration speed in the later stages of the tournament. The only downside is that it occasionally makes our codebase look a little like this.

3 Communication

3.1 Shared Array

Truth be told, we did not utilize the shared array extensively this year. We used it primarily to track high-level strategic data, such as Rat King locations, map symmetry, cheese mines, and potential Rat King formation coordinates. Because the game mechanics prevented Baby Rats from writing to the shared array, our communication focus shifted heavily toward Squeaking as our primary decentralized messaging system.

3.2 Squeaks

I designed a Squeak system that was intentionally easy to iterate upon and extend. The first three bits of every Squeak functioned as a header to indicate the message type. Each specific SqueakInfo implementation extends a sealed interface, which allowed us to leverage polymorphism to keep the communication logic clean and modular.

Since Squeaking was the only way for Baby Rats to coordinate independently of the Rat King, establishing a robust framework for generating and decoding these transmissions was vital. This bit-level approach ensured we could pack whatever data we needed into the limited Squeak size while maintaining an organized codebase that was easy to debug as the tournament progressed.

4 Pathfinding

Although the specifics of Battlecode change every year, the necessity for an effective pathfinding algorithm remains a constant. This year introduced a major hurdle: the robot vision cone. Unlike previous years, where robots enjoyed a full sensing radius, we were now limited to a directional field of view, significantly complicating environmental mapping.

I have split this section into two parts. The first outlines the basic BFS we settled on prior to this year (taken straight from the corpse of our unfinished 2025 Postmortem) while the second details how we modified and upgraded that system for the 2026 season. While Part One covers what has become common knowledge within the Battlecode community, I wanted to explain it in detail for anyone new to the competition.

4.1 Pre-2026: BFS Begins

4.1.1 "Unrolled BFS"

Battlecode operates under strict computational constraints enforced by the bytecode system. Since Java files are compiled into bytecode for the JVM, Battlecode can track exactly how many instructions your bot executes. Exceeding the turn limit (17,500 bytecodes this year) causes your bot to halt mid-execution, effectively wasting a turn.

To stay within this budget, a common strategy is **loop unrolling**. In standard loops, the JVM performs a termination check after every iteration; over hundreds of iterations, these checks balloon bytecode usage. Consequently, traditional pathfinding like Dijkstra's or A* are often unusable without extreme optimization. Our solution was an unrolled Breadth-First Search. By exploring nodes in a fixed, radially outward order, we can generate a reliable path estimate with minimal overhead. While this method doesn't strictly guarantee the shortest path even within a fixed radius, it is "good enough" for most scenarios.

4.1.2 Bug Nav

While BFS is excellent for local navigation, it struggles with larger, complex obstacles. To handle these, we implemented Bug Nav. We began with the basic implementation outlined in the developer lecture series, but quickly found it too simplistic for more complex maps.

Seeking more robust guidance, we turned to Gone Fishin's 2023 Postmortem and adopted their stack-based Bug Nav strategy. As the bot navigates an obstacle, it pushes impassable directions onto a stack. If the stack is non-empty, the bot attempts to move in those stored directions; once empty, it resumes its direct path toward the target. When an obstacle is first hit, the bot simulates both a left-hand and right-hand turn, choosing the direction that puts it closer to the target after a set number of steps. We also added logic to immediately turn away from map borders, as hugging the edge is rarely optimal. Finally, to prevent the bot from getting trapped in infinite loops, we maintain a set of "turn locations", if a bot tries to turn where it has turned before, it automatically reverses direction.

4.1.3 BugBFS

The integration of these two systems is straightforward: the bot performs an unrolled BFS and monitors how much progress it makes. If the BFS fails to move the bot closer to the target, it hands control over to Bug Nav. Once Bug Nav clears the obstruction and the BFS can again reduce the distance, the bot switches back. While this hybrid is bytecode-intensive, it works the vast majority of the time.

4.2 Adapting for 2026: The Dirt Plight

4.2.1 Start → Sprint 1

Initially, I simply ported my 2025 codebase and made minor adjustments for the new vision cone. I introduced a global 2D array to track persistent data like dirt and passability across multiple turns, which allowed the robots to make informed decisions about areas currently outside their vision. However, this early version was completely broken for Rat Kings. Because they occupy a 3x3 footprint, my 1x1-centric logic failed them entirely. Fortunately, mobile Rat Kings didn't become meta until later in the competition. During this phase, I also implemented a safety check to keep Baby Rats from wandering into cat-visible tiles, preventing a lot of senseless deaths.

4.2.2 Sprint 1 → Sprint 2

This period was a developmental wash. Seth was away at an AI Safety conference in San Francisco and I was preoccupied with running fraternity rush, leaving the bot untouched for an entire week. Looking back, that lost week of development was a massive blow. Watching our matches during Sprint 2 was painful; our lack of effective dirt-handling logic was a glaring weakness. We lost several matches purely because our bots didn't know how to handle environmental obstructions, making "dirt-aware" pathfinding my top priority moving forward.

4.2.3 Sprint 2 → US Qualifiers

My first fix was to treat all dirt as passable, simply stalling the bot if its action cooldown became too high. This forced the bots to mine through obstacles and explore high-dirt regions, which proved surprisingly effective. Combined with our exploration strategy, this allowed us to win decisively on maps like Figure 1, where vital cheese mines were completely buried in dirt.

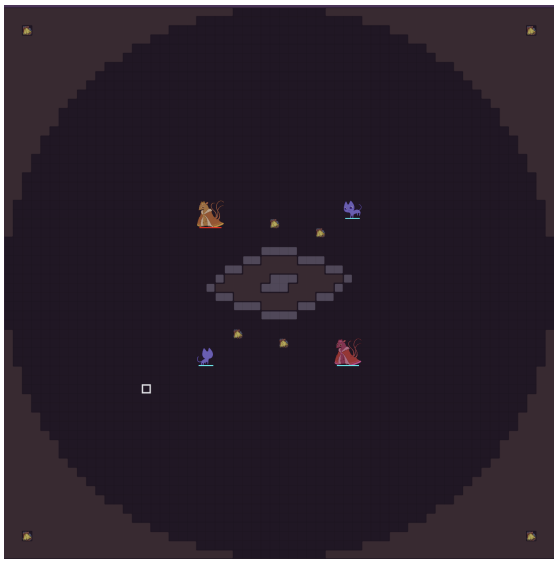

Figure 1: Dirty Cheese Mines

Eventually, I realized we were doing a lot of unnecessary digging, which was particularly detrimental when our Rat Kings were facing cheese famines. I introduced an additional cost to the BFS for digging, allowing me to fine-tune the "laziness" of the Baby Rats. That said, I still believe that aggressive digging is often the correct play; it opens up navigation lanes and increases long-term cheese throughput, which is a net gain for economy.

4.2.4 US Qualifiers → Finals

Late in the tournament, Seth noticed a micro issue where rats were getting stuck trying to micro through walls. To solve this, I developed a cheap reachability solution using bit shifting. This allowed us to calculate full reachability (even for areas outside the vision cone) for only about 300 bytecodes.

This success, combined with insights from Om Nom's 2025 Postmortem, inspired me to rewrite our BFS from scratch using bit shifting. The version I had been using was code I wrote back in high school, it was robust, but it cost a staggering 6-8K bytecodes. While I made significant progress on the bit-shifted version, I couldn't get it quite as reliable as the old implementation before the final deadline. I definitely plan on fully adapting Om Nom's method for Battlecode 2027.

5 Macro

This year's Battlecode game was definitely micro heavy, but still had some interesting macro tradeoffs. For example, from what I could gather, teams

often had very different heuristics for when to bring cheese back to the Rat King.

5.1 Improving Macro

I think that macro is best improved by scrimmaging against good teams, and copying what they do. It may be the case that our team just doesn't like working on macro much, but we don't try too hard to be original with our thoughts when it comes to macro. We also didn't try iterating on ideas too much. For the most part, we would queue scrims against teams above us on the leaderboard, reverse engineer their macro (which is much easier than reverse engineering micro, pathfinding, or communication), copy whatever strategy we gleaned, and if it made us better then we would keep it. This is how we arrived at our strategy for forming Rat Kings on cheese mines, our magic numbers for how many Baby Rats to build and when to build more Baby Rats, and many other parts of our macro.

5.2 Baby Rat Taxi

In contrast to the last section, we did have one unique idea this year for macro! This was our Baby Rat taxi system, in which Baby Rats with low cheese would help carry Baby Rats with excessive amounts of cheese back to the Rat King. This was incredibly helpful for bringing Baby Rats who ended up with 150+ cheese back to the Rat King, but this situation was rare enough that the taxi system didn't have a super large impact on our economy. The system also comes with tradeoffs, such as the increased action cool down in case you then run into an enemy, and the fact that a cat pouncing on you will kill both rats immediately.

6 Micro

Our micro, as with many teams, was based on "XSquare Style Micro"; once we decided to use micro, our bots would go into a mode in which each square was evaluated based on a variety of factors, such as passability as well as proximity to allies, enemies, and cats, and then a chain of if/else statements decide whether any given square is better than another. This is done over the whole set of possible squares (9, including not moving) in order to decide the best square to move to.

6.1 Information Maximization

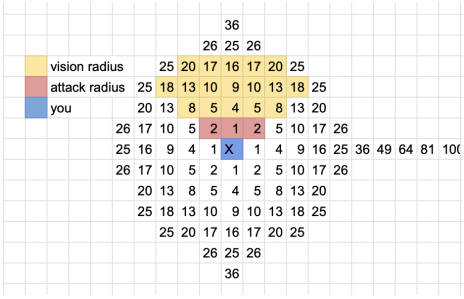

Figure 2: Spreadsheet for Baby Rat Micro

A key aspect of creating strong micro this year was maximizing the information you could consider. As pictured above, units this year had a vision cone, but knowing about enemies and allies to your left, right, and behind you is generally very important in micro, and this year more so than ever because enemies behind you or to the side of you can ratnap you regardless of health. The system we came up with to address this was squeak maximization: every turn in Micro, every Baby Rat would squeak their health, nearest enemy health, nearest enemy direction, and nearest enemy location. This would help our rats build a comprehensive picture of allies and enemies in the surrounding area, giving the ability to make informed decisions and avoid being ratnapped. However, in very crowded engagements, we would almost always go over bytecode, which is not ideal, but we found to be a very worthy tradeoff in return for making smarter decisions.

6.2 Cat Micro

Cat micro meant two things to us: being able to effectively fight cats, and being able to avoid cats while fighting enemies. For the former, the main thing our cat micro emphasized was avoiding cat vision, and if that is not possible, then avoiding the 2-3 squares directly in front of the cat. This set of locations, which we called the cat path, was by far the most dangerous squares around the cat because if the cat moved forward then it would kill anyone in those squares. During regular micro, we found that maximizing cat distance and avoiding the cat path were both important, but could not be prioritized too heavily at the cost of making worse decisions regarding enemies. In general, enemies behave more predictably and smarter than cats, so preventing enemies from ratnapping you was a higher priority for us than avoiding the cat path.

6.3 Throwing

Throwing was super important this year, and one thing that made our micro significantly better was considering every (reasonable, typically valid, and generally beneficial) combination of move + turn + throw, by scoring each path depending on whether it ended by hitting a cat, an enemy, a wall, an ally, or just left vision without any of these occurring. When not in combat and not seeing cats, we would hold enemies for multiple turns before starting to consider throwing, for bytecode reasons as well as because there is always the possibility we find another enemy or a cat for which we can find a much higher value throw. This system did take some trial and error, because some combinations of move + turn + throw generally turned out to be bad in most situations. For example, move forward, face current direction, and throw tended to make us worse when we considered it a valid option, likely, because it would lead us to enemies or cats.

6.4 Improving Micro

From many years of Battlecode, for which most of the time I have focused almost entirely on micro (it tends to be the case that Urav works on pathfinding and communication, and I halfheartedly write macro and don't iterate on it much), I have found that by far the best way to improve micro is to watch a lot of games. This includes games against your own bots, and games against other teams (especially ones with better micro!). I find nothing more helpful than following individual units into combat, and analyzing why they make the decisions they do. This involves our own bots, where I watch them make stupid decisions, and try to make that individual decision less stupid in a way that minimally impacts other smart decisions. This also involves watching bots with better micro (I watched so much Old But Gold this year!), trying to reverse engineer the decisions they make. The latter is much harder because you don't know what information they are processing, and there are many ways to arrive at any given decision, but is still absolutely worth it because it can sometimes provide you amazing strategies.

7 Conclusion and Reflection

We had a blast with Battlecode this year, as we do every year. This competition is unbelievably addictive, and it is something I look forward to every year. I think all too often, teams get caught up in trying to craft the perfect pathfinding algorithm or optimizing a data structure's bytecode to a ridiculous extent. There has been a pervasive sentiment this year that to succeed, one must be "spamming" codegen and similar high-level optimizations, but I could not disagree more.

I believe that focusing on macro strategy, consistently watching scrimmages, and being willing to experiment with a wide variety of ideas is the best way to improve a bot. I, too, can fall into the bad habit of becoming overly attached to

an idea and refusing to accept when it simply isn't working out. However, the most effective path to success is often just trying as many strategies as possible, some will just naturally be better than others and you won't know until you try.

Battlecode rewards you immensely for paying attention to the smaller details, and I suspect we spent far more time reviewing scrims than actually writing code. While algorithms and optimizations are certainly important, they shouldn't come at the cost of neglecting your overall strategy. Congrats to **subscribe to caterpillow** for their well deserved win, and we hope to take that number one spot next year :)