The Kragle - Battlecode 25 Postmortem

Justin Ottesen, Andrew Bank, Matt Voynovich

Last Updated: February 5, 2025

Contents

1 Introduction

We believe our postmortem will be of unique help to future Battlecode participants. Often, teams who make postmortems are those who have consistently placed highly from the start. We have the perspective of a team that spent years with no consideration of qualifiers, who this year became one of the higher level contenders, going toe-to-toe with the 2 seed in the US qualifiers.

Hopefully you find this useful!

1.1 Our Team - The Kragle

We are a team of three computer science students at Rensselaer Polytechnic Institute (RPI). In this year's competition, we made "the leap" from being a team that couldn't submit a final bot to being a contender for the final tournament. We'll outline some tips and tricks for other aspiring teams to make 'the leap' themselves. We are historically terrible at coming up with team names, and ended up choosing **The Kragle** after Justin had a burst of inspiration working on a `MicroManager` class shortly after re-watching The Lego Movie[1]. Just you wait for what we cook up next year. Some more information about us is below:

Andrew Bank has competed in Battlecode since 2021. He is a Computer Science and Computer Systems Engineer at RPI and will graduate in this spring 2025. Andrew has interned at Johns Hopkins Applied Physics Lab, he has created the Circuit Randomizer for Personalized Learning as an undergraduate research project under Professor Shayla Sawyer, and is currently working on another undergraduate research project: the MusicX project for Professor Sawyer's Mercer Xlab. In his free time, Andrew enjoys having more Strava followers than Instagram followers, and he enjoys treating Stardew Valley like an operating systems optimization problem while playing with his siblings.

Justin Ottesen has competed in Battlecode since 2022. He began his undergraduate degree in Computer Science in Fall 2021, and graduated a year early in Spring 2024. He stayed at RPI for his Master's, where he does research with the BRAINS Lab under Oshani Seneviratne with a focus on incentive design for smart contract protocols, and is on track to graduate in Fall 2025. Outside of classwork, he has worked as an intern at Nasuni since Summer 2023, and is on the D3 NCAA Cross Country and Track teams at RPI. In his free time, he enjoys hiking, running, programming, and playing Mario Kart Wii.

Matthew Voynovich started competing in Battlecode this year. He began his undergraduate degree majoring in both Computer Science and Information Technology and Web Science at RPI in Fall of 2023. He is scheduled to graduate in Spring 2026 and plans to pursue a masters under the Co-Terminal program at RPI. Currently, Matthew works as an undergrad researcher under Thomas Morgan studying quantum phase investigations, and in his free time enjoys long distance running with his friends in the Rensselaer Running Club.

[1] One of the greatest movies of all time

1.2 Past Performance

Andrew was the only member to compete in 2021. In 2022, Justin joined Andrew, working together again in 2023 and 2024 along with some other teammates. We did not perform particularly well in any of these years, typically holding a 1200-1500 rating, with no notable performances in any of the tournaments. Unfortunately, the RPI Spring semester always starts very early, so we were always balancing classwork along with the competition. This year we went all in, and entered the US qualifier tournament with the 10 seed, rated at 1730. I guess you'll have read this to see how we did...

Although we aren't sure our code will be useful to anyone else, our GitHub repositories can be found below:

2021 **Fire Nation** - Lost to the sands of time...

2022 **Kernel Byters** - https://github.com/justinottesen/Kernel-Byters

2023 **PC Ghosts** - https://github.com/andrewkbank/bc2023

2024 **Goat House** - https://github.com/justinottesen/battlecode24

2025 **The Kragle** - https://github.com/justinottesen/battlecode25

1.3 Game Overview

This year's introduction message is shown below:

> *The bread and food of yore has begun to run out, forcing robot society to adapt. Gone are the jovial ducks, replaced by steampunk robot bunnies who have converted their need for nutrients into a reliance on paint. These bunnies have become territorial, forming clans and defense formations to protect the resource that keeps them running.*
>
> *For the past two centuries, these bunnies have stayed within their own territory, but clans have begun to degrade their environment and need to start branching out. Will these clans be able to expand their territory and generate enough paint to protect their families? Or will they stray too close to other clans and be wiped out in conflict?*

As hinted above, this year's game was a competition of territory control between paint-crazy bunnies. The first team to paint 70% of the map would win. Each team had two starting towers, which could produce resources and robots. These robots had different abilities, but their goal was to work together to build more towers and paint as much territory as possible.

We saved copies of both the initial and final specs to our repository in case they are taken down in the future. For a full description of the game, see our repository. If you are new to Battlecode, we highly recommend this Battlecode Guide, written by **XSquare**, a long time Battlecode competitor.

2 Strategy & Implementation

2.1 Sprint 1

2.1.1 Setup

The first thing we did was decide how to structure our code. We took a lot of inspiration from **XSquare**'s 2024 code. We created an abstract `Robot` class, and created a subclass for each robot type (`Soldier`, `Splasher`, `Mopper`, `Tower`). This heavily simplified our `RobotPlayer` class, and helped us split functionality between differing robot types[2]. Along with these classes, we also had utility classes, like `MapData`, `Explore`, `Bugpath`, and others.

Our first priority was getting a bot that could effectively explore and capture towers. Building towers would allow our team to increase their resource income and create more robots. We had each tower spawn a soldier and a mopper who would search for a ruin. We copied the capture logic from the provided `examplefuncsplayer`. The soldier was to paint the pattern around the tower, and the mopper was to clear any enemy paint in the way.

2.1.2 Resources & Towers

We made two observations very early on about resources:

1. It is not worth upgrading towers.
2. It Money towers are always better than Paint towers.

The first one is relatively obvious looking at the costs of new towers vs upgrades. For Money towers, the following table shows the cost associated with each level of the tower:

Level	Upgrade Cost (Chips)	Mining Rate (Chips / Turn)
1	1000	20
2	2500	30
3	5000	40

From this table, we can graph the cost and mining rates:

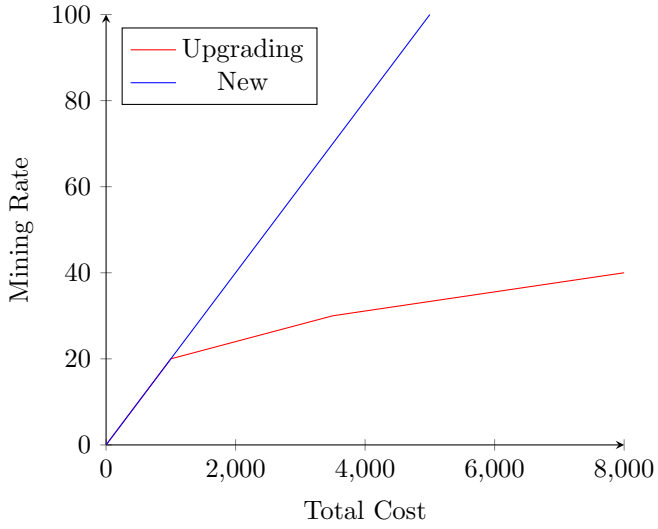


Figure 1: A comparison of the Chip Mining Rate for the "Upgrading" and "New Tower" strategy

[2]In hindsight, it may have been even smarter to have another level to the hierarchy where we had Unit as the base class, Robot and Tower as subclasses, and other types inherit from these. Towers don't need to do pathfinding, and they can't really learn much about the map, so this was bad organization and wasted bytecode

As is clearly shown here, when you have the choice between spending on a new tower, or upgrading an existing one, there is no reason to upgrade. The same can be said for Paint towers, but the next observation means we don't have to worry about them.

Even though paint is arguably the more important resource, Paint towers are useless[3] because money towers actually produce paint more effectively than paint towers. Each tower spawns with 500 paint. Since money towers produce 20 chips per turn and they cost 1,000 chips, they pay for themselves in 50 turns. So, if a Money tower calls `rc.disintegrate()`, we can trade 1,000 chips for 500 paint. If we do this every 50 turns, money towers have an effective paint mining rate of 10 paint per turn, which is double what paint towers are capable of.

Here is the corresponding graph to the above for paint production vs chip cost:

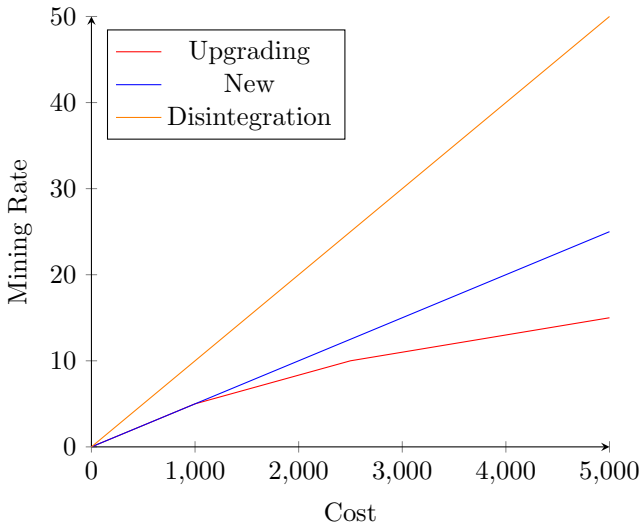


Figure 2: The same comparison as the previous figure, this time including the "Disintegration" strategy

There are a few other big bonuses of this strategy:

1. We now have the choice of converting chips to paint. With paint towers, if we have more paint than we need or can use, we are stuck with it.
2. We no longer have to coordinate what tower types we are building, since we are always building money towers (we ignore defense towers for now).
3. Since chips are global and paint is local, we can use money towers that are far away from the active parts of the map to subsidize paint production where it is needed most.

With this, it made a very useful optimization trivial. Marking a tower pattern costs 25 paint, which is significant considering the soldier's paint capacity is 200, and it already takes a minimum of 120 paint to capture a ruin. Since we knew we were only building money towers, we could avoid marking the pattern and save paint.

As a side note, we thought this strategy was pretty well known, however this ended up winning **Gone Whalin'** the "Most Innovative Breakthrough Prize", and most teams noticed it when they started doing it right before US Qualifiers. Overall they simply had a much better bot than us, holding a pretty stable rating around 1900, and we are happy to have independently found the same insight as such a strong team.

[3]This is true until you factor in SRPs, but we are getting there

2.1.3 Special Resource Patterns

Once SRPs are introduced, the math on paint production changes. Each active SRP boosts paint production by 3 units per tower per turn. The following table shows how the number of SRPs affects the per turn production rates of different towers:

SRPs	Money Tower Chips	Money Tower Paint	Paint Tower Paint
0	20	10	5
1	23	11.5	8
2	26	13	11
3	29	14.5	14
4	32	16	17
5	35	17.5	20

As you can see, once there are 4 or more SRPs, paint towers are better suited for their intended purpose. We decided not to worry about this and stick with our money tower only strategy, to keep things simple. Plus, we (regrettably) did not prioritize SRPs for Sprint 1, our soldiers simply checked if the current location could have an SRP and if so, they marked one. SRPs were a huge dominating force, and were very quickly nerfed after some horrific matches between the top teams in this tournament.

2.1.4 Pathfinding & Map Representation

Units stored their knowledge of the map in memory, however we ended up rewriting this system, so this will be described further later. Units traversed the map using a BugNav algorithm. We used the BugNav algorithm from XSquare's 2024 code, and it worked with minor tweaking. If you have questions about how to implement this, there was a lecture on it this year here.

Optimal pathfinding wasn't a high priority this year since ruins/towers guaranteed a lot of open space, and the specs guaranteed walls to take up less than 20% of the map. In theory, Teh Devs could've made maps with few ruins and many walls, but that never really happened[4]. Even though something like unrolled BFS would've been optimal, BugNav worked well enough in practice, and because source code for it already existed, we saved a lot of time not worrying about pathfinding.

2.1.5 Sprint 1 Performance

The bracket for Sprint 1 can be found here.

We entered the tournament as the 55 seed out of 160 teams, with a rating of 1533. Our first match was a 4-1 win against the 74 seed, **be right back** with a rating of 1483. Their strategy was to do an early rush with soldiers to try and take out our towers, then produce a single splasher to cover some paint. This worked against us on the first game, as our starting soldiers got stuck trying to capture an SRP with enemy paint, and our moppers didn't help them.

[4]They did make a "Maze" map, but this really just corralled units rather than making it hard to pathfind.

The left image below shows the result of our loss, and the right image shows one of our wins. On this particular map, the opponent's rush got trapped against a wall, allowing us to expand freely.

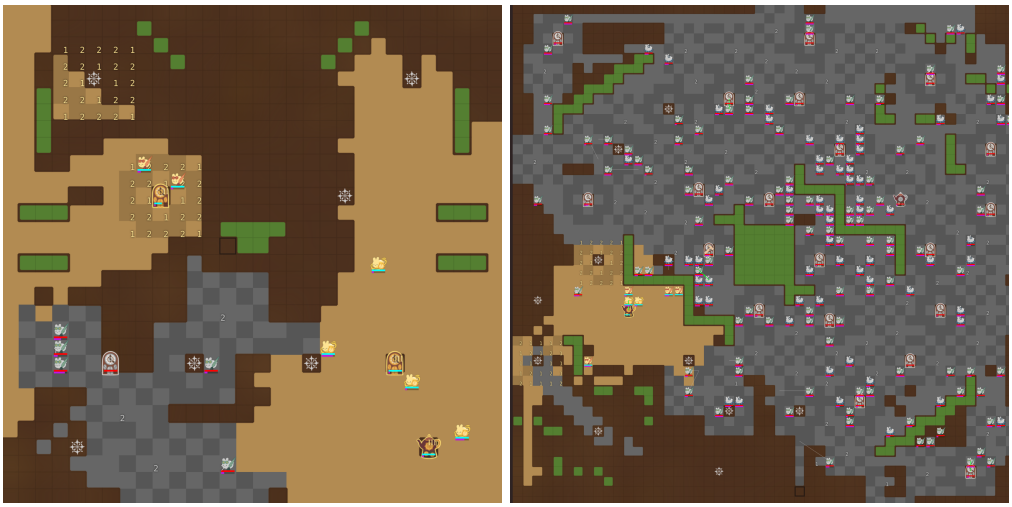


Figure 3: Some screenshots from the discussed sprint 1 matchups

Our next matchup did not go as well for us. We faced against the 10 seed **immutable**, who had a rating of 1737 at the time, and got swept 5-0. We kept up in the early game, our initial soldiers seemed to perform about equally in finding ruins, but after some starting territory was established, they always pulled away with a commanding lead. There were three main reasons for this:

1. They prioritized SRPs heavily. In Sprint 1, this was definitely the meta, since the pattern allowed for an absurd amount of overlap, and there was no cost to making them. This gave them a much stronger economy than ours.

2. They used their economy better, settling around 50% soldiers, and 25% each of moppers and splashers. We usually ended up with about even soldiers and moppers with no splashers, since we did not have time to create splasher logic by this point. Their balance allowed for much better expansion.

3. They explored the map better. Our soldiers would often get stuck on ruins they couldn't complete without help, and our moppers were terrible at finding soldiers to help. This meant we got stuck anytime we saw enemy paint.

We are sure there are many other things they did better than us, but these were the three big differences that made the others negligible.

2.2 Sprint 2

2.2.1 Rewrite

Our `sprint_1` code was messy and not well documented, so we did a full rewrite after the submission deadline. This was in the `jottesen_test` folder, named so because it was initially a test of a new pathfinding algorithm. Similarly to `sprint_1`, each robot type had its own class, along with several utility classes. We ended up (mostly) sticking with these utility classes for the rest of the competition, so we will describe them here.

2.2.2 MapData

This is really the core of "what the bot knows". It stores an array in memory, with one 32-bit int per tile on the map. Below is how we encoded the information in these bits:

```
private static int[] mapData;
private static final int UNKNOWN = 0b0;

// Bits 0-1: Immutable characteristics
private static final int EMPTY             = 0b01;
private static final int RUIN              = 0b10;
private static final int WALL              = 0b11;
private static final int TILE_TYPE_BITMASK = 0b11;

// Bits 2-4: Tower Type Data (Only applicable for ruins)
private static final int UNCLAIMED_RUIN     = 0b001_00;
private static final int MONEY_TOWER        = 0b010_00;
private static final int PAINT_TOWER        = 0b011_00;
private static final int DEFENSE_TOWER      = 0b100_00;
private static final int TOWER_TYPE_BITMASK = 0b111_00;

// Bit 5: Friendly = 1, foe = 0 (Only applicable for claimed ruins)
private static final int FRIENDLY_TOWER = 0b1_000_00;

// Bits 19-20: Goal Tower Type
private static final int GOAL_MONEY_TOWER   = 0b01_00_00000000000_0_000_00;
private static final int GOAL_PAINT_TOWER   = 0b10_00_00000000000_0_000_00;
private static final int GOAL_DEFENSE_TOWER = 0b11_00_00000000000_0_000_00;
private static final int GOAL_TOWER_BITMASK = 0b11_00_00000000000_0_000_00;

// Bit 21: Goal Paint Color
private static final int GOAL_SECONDARY_PAINT = 0b001_00_00_00000000000_0_000_00;
private static final int GOAL_COLOR_KNOWN     = 0b010_00_00_00000000000_0_000_00;
private static final int GOAL_COLOR_CANDIDATE = 0b100_00_00_00000000000_0_000_00;

// Bit 22: Contested
private static final int CONTESTED_TARGET = 0b1_000_00_00_00000000000_0_000_00;
```

Figure 4: The way we encoded map information in bits

We had plenty of bits left over, and many which were initially used and taken out. Upon spawning, each robot called `MapData.updateAllVisible()` to load all known information about the immediate surroundings into the `mapData` array. Each time the robot moved, it updated only the newly visible tiles. This was a bytecode compromise, since updating all the visible locations was expensive, but only updating newly visible leaves a blind spot around the robot. To help mitigate this, we also kept a list of the known ruins, and updated that list every turn.

2.2.3 Painter

We also created a `Painter` class, which heavily relied on the `MapData` class. The `Painter` held the logic for which tiles should be painted, and what color should be used. It used the `GOAL_*` bits from `MapData` to encode these. Whenever we saw a ruin, we set these bits in the 5×5 area around it. We later did the

same with SRPs[5]. This really streamlined our painting logic, and we didn't have to worry about checking the correct color each time.

We also put logic for moppers here, but our moppers were terrible so it isn't really important to discuss. At this point, we still didn't have splashers.

2.2.4 Communication

We added the groundwork for our future communication class, however it was not used at all in this iteration. We wrote functions to communicate map symmetry, but did not use them until much later in the competition.

2.2.5 Pathfinding

This class was one of the big upgrades of the rewrite[6]. In the initial bot, we used a pure BugNav algorithm from a previous year, as mentioned previously. This worked, but was inefficient at times. Since we stored the map internally, we could simulate doing BugNav, but have the bot actually take shortcuts.

The core idea was that the bot would have some goal target that it was trying to move towards. The algorithm is loosely described below:

1. Simulate greedy movement towards the target on the internal `MapData` array. If you hit either the target or an unknown tile, return and follow that path.
2. If you hit an obstacle, simulate BugNav until you make it past, or you hit an unknown tile.
3. Repeat the algorithm on the result of the above BugNav.

We managed the target destinations by creating a fixed buffer `Stack` class. Whenever we needed a checkpoint target (found by the BugNav step), we would push it onto the stack. When we reached that target, we popped it off and recalculated the path to the next target. Another useful optimization we used in pathfinding was caching and pre-calculating moves. This let us calculate several moves in advance without wasting a ton of bytecode on recalculations.

The main reasons this failed were because of high bytecode cost, greedy search instead of BFS[7], and bad handling of the case of maximum stack depth of locations. It would outperform our old algorithm in "easy" pathfinding scenarios, but would often unpredictably get stuck on the harder situations. Traditional BugNav from XSquare's codebase was a more reliable solution.

2.2.6 Opening Theory

Up until this point, we hadn't really considered any type of coordinated effort. Our entire goal was just to build up economy, and if you see any enemies, try and beat them. This is the point we started to consider higher level strategy, so we made a `sprint_2` bot to test our changes against the previous `jottesen_test` and make sure we were in fact making improvements.

The goal of the opening is to get a tower-count lead over your opponent. SRPs are not worth it since they take too long to activate, they don't provide enough resources, your team always starts with more than enough chips for towers, and there are always available towers to capture (this is something Teh Devs could've, but never really messed with). In the opening, each tower could spawn two robots before running out of initial paint. The logical choices were either spawning 1 soldier and 1 mopper, or spawning 2 soldiers. 2 soldiers is ideal since together, they could capture a tower without any surrounding paint in 12 turns. However, a mopper would allow the its soldier partner to capture a tower with enemy paint in its pattern. This provides quite the dilemna and was one of the most interesting part of optimizing strategy.

After Sprint 1, we were made aware that rushing with a soldier-pair in the opening was a viable strategy. This provided an elegant solution to the opening dilemna: the soldier pair is better.

[5] This bot did not bother to paint SRPs, but still heavily outplayed the previous version

[6] It was honestly a waste of time since we ended up scrapping it later. It used too much bytecode and was too buggy.

[7] During the final tournament stream, the third place finishers **Om Nom** mentioned they used this optimization of BugNav and BFS, so it seems we turned back from mining right as we were about to hit diamonds. Definitely a skill issue on our part

Each tower spawns 2 soldiers. The first priority of the soldier pair is always to capture an "uncontested" ruin[8]. Until the soldiers finished the first ruin, they were in "opening mode." However, in the case that the soldiers only found ruins that were "contested," they would rush. This is theoretically the perfect scenario for rushing since the best counter to rushing is to capture a ruin as fast as possible. However, if every nearby ruin is contested, rushes become stronger since it is more difficult to capture ruins.

Figure 5: Both soldier pairs quickly capturing uncontested ruins by round 17

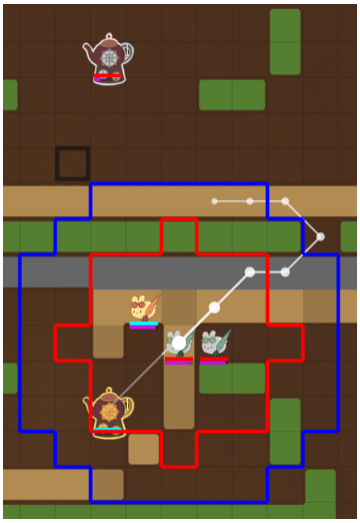


Figure 6: A soldier pairs finding an enemy tower before an uncontested ruin. Note that enemy soldiers aren't close to capturing a ruin yet.

This opening strategy was wildly successful on the ladder. On 90% of the test maps, the soldier pair would find an uncontested ruin and capture it before turn 20. Our logic didn't result in the soldiers rushing often, but when they did, it was often successful in giving our team a large lead. It wasn't infrequent to see our team with a marginal lead over top teams out of the opening.

Our opening strategy remained unchanged after Sprint 2. This lead to some pitfalls. First, requiring the soldiers to pair up meant that if they found an uncontested ruin, they could capture it quicker than if they split up and both found uncontested ruins. However, requiring the soldiers to pair up resulted in high

[8]A ruin with no enemy paint blocking the pattern

variance: sometimes, they would miss an obvious ruin since they didn't spread out. Secondly, since rushes happened so infrequently, we never had good logic for what to do after a rush completed, and relied mostly on the rush being disruptive enough to give us a massive lead that we couldn't fumble.

2.2.7 GoalManager

The last big change we made before Sprint 2 was introducing the `GoalManager` class. We previously were storing the goal location as a single `MapLocation` and the type as an enum. However, this lead to many problems. Often, ruins would be abandoned when robots went back to get their paint refilled, since they were the only robot that knew it was in progress. After refilling paint, they would act as if they were a new soldier.

We decided to use a stack for the goals, giving robots a lot more choice in their intended actions on goals. This ended up being a huge improvement, and we will definitely be bringing an enhanced version of this back in future years.

2.2.8 Paint Towers?

At some point, we made a change where we built some paint towers. I forgot we did this, but we ended up reverting this decision later. As explained in previous sections, money towers are simply better until you have 4+ SRPs. We did not prioritize SRPs whatsoever, so paint towers didn't make any sense for us, especially since we had robots fighting over whether to make paint or money towers, wasting time and resources.

2.2.9 Sprint 2 Performance

The bracket for Sprint 2 can be found here.

We entered the tournament as the 63 seed out of 183 teams, with a rating of 1559. Our first match was a 5-0 win against the 66 seed, **achromatic** with a rating of 1551. There was not much of note in these matches, we outplayed them by a wide margin, expanding faster and winning any direct territory battles.

Our second match was the opposite, we were against the 2 seed **Super Cow Powers** with a rating of 2017. We were no match for them and lost 5-0. While they simply did everything better than us, the big details were that they were much more purposeful with their expansion and we had too many bugs that caused us to sit idle doing nothing.

2.3 US Qualifiers

Between Sprint 2 and the US Qualifier tournament we made our biggest improvements. We came out of Sprint 2 at 1570 ranked 63rd, and finished the competition around 1800, ranked 27th. Granted, our rating after Sprint 2 did not reflect our changes since we made several upgrades right before the submission deadline, but this is where we really started to have a chance of making Finals.

2.3.1 Floating Resources

Thanks to our excellent opening strategy, we often found ourselves with a resource lead, even against higher rated teams. However, our bot was fantastic at throwing leads. We would quickly get out-expanded in the mid-game, and we would float[9] tons of paint in all stages of the game.

One of the unforeseen issues with only using money towers is that if they get stuck with less than 100 paint, they can never build units. This became a major problem when our soldiers would all refill on paint and leave every money tower with around 70 paint. Even with only 10 towers, this would result in us floating 700 paint. Since we were basically the only top team completely reliant on money towers, we could have a paint lead on paper, but if all of that paint was not spendable, it wasn't a paint lead in practice.

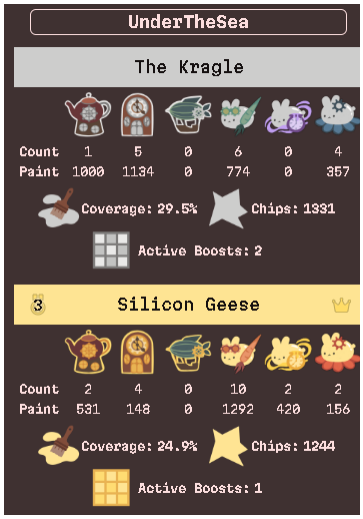


Figure 7: We have a sizeable resource lead against Silicon Geese. However, we are floating thousands of paint in towers that could go towards making robots.

The solution was quite simple: make refilling soldiers leave precise amounts of paint required to build robots. Obviously this was a trivial fix, but the resource efficiency theory behind it was the important part.

2.3.2 Reducing Idle-Time

Besides floating resources, the other main culprit for our bot throwing leads was idle time. Often, we'd find ourselves with not just a resource lead, but a robot count lead. On paper, the team with more robots is always at an advantage. However, a robot-count lead means nothing if half of your robots are derping around.

[9] terminology from RTS games meaning to be in possesion of unused resources. You always want to fully utilize your resource, so floating resources is bad.

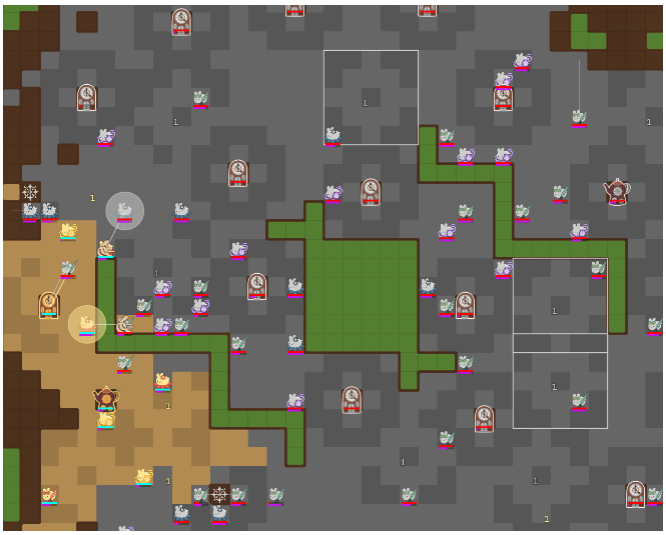


Figure 8: A gaggle of soldiers and splashers in the upper right derping around far away from where they could be useful...

The easiest idea our team had for reducing idle time was communicating battlefronts. Most games that made it to the mid/late game would result in a battlefront where both team's robots would clash near the center of the map. Without communication, our robots would explore around the map, wasting precious time, until they stumbled across a battlefront. With communication, our robots would always have knowledge of the nearest battlefront, so they could waste as little time as possible not contributing to the main fight.

Even though our battlefront communication was mostly patchwork, it still gave massive improvements to our bot. The bots having even a vague idea about where the battlefront was reduced their idle explore time massively.

Finally we came to the obvious realization that other teams didn't bother having their robots refill on paint. Our robots would spend about half of their lifetime traveling back to towers and waiting around towers for paint. This meant that our robots would spend about half of their lifetime in an idle state, not contributing anything.

Figure 9: 10 soldiers waiting for a refill on a money tower that is out of paint...

Only after we started looking carefully at what other teams were doing did we notice that no other team bothered having their bots refill on paint. This was such an obvious improvement: It cuts down on robot idle time, and allows robots to perform longer tasks at the cost of chips. You would only really want to preserve high-value robots, which was just the splashers.

This change came way too late in the competition and was a stark reminder to pay careful attention to other team's strategies. Other teams beat your bot for a reason, and the easiest way to improve is to copy what they do better than you.

2.3.3 Qualifiers Performance

The bracket for the US Qualifiers can be found here.

Thanks to our bots ability to consistently obtain a lead out of the opening and our improved efficiency, we were the 10th ranked US University team. Considering 12 US University teams make the final tournament, we were excited about our prospects of making the final tournament. However, we knew that every US University team ranked between 9th and 16th were extremely close in strength, and were likely separated more by auto-scrim luck than strength.

The US qualifier tournament is a traditional double-elimination tournament. This means 8 teams qualify for the tournament by winning in their top 16-winners matchup, and 4 teams qualify for the tournament by wining their top 24-losers matchup. The way the team strengths were distributed, basically only the top 16 teams had a chance of qualifying. This meant that there were 4 "quadrants" of 4 teams, where 3 teams would make the finals. Our quadrant looked good since it included the 2 seed, the 7 seed, the 10 seed (us), and the 15 seed. If everything went according to seeding, we would lose our matchup against the 7 seed, then win our matchup against the 15 seed to qualify for the finals.

We even believed we had a chance of upsetting the 7 seed, since the day of the submission deadline, we went 7 for 13 against them in scrimmages. However, our game against the 15 seed would not be free, as we had gone 5 for 17 against them the week leading up to the tournament in scrimmages. We knew the tournament was going to be a bloodbath, and several teams of "Finals Caliber" would not make the cut.

Disaster struck for us when the 15 seed, **podemice** miraculously upset the 2 seed, **JDK? More like IDK.** We narrowly lost our matchup against the 7 seed, **placeholder name 2,** in a score of 3-2. We were then given the daunting task of beating the 2 seed, **JDK, More like IDK** in losers. Tragically, we lost again 3-2 to be the only top 12 team not to make the finals.

2.3.4 Tournament Structure Suggestions (and Complaints)

There were a few factors that made this result particularly bitter. The first was the knowledge that both of our decisive sets were 1 game away from flipping. The second was that we didn't get to watch our losses on the stream, despite both games deciding a team to qualify for the finals[10]. The third and final factor was that we believed our team was top-12 strength. It just so happened that **podemice** was horrifically underseeded, since they were very clearly also top-12 strength. As a result, our "quadrant" of the bracket was far and away the hardest one.

The traditional bracket is designed such that the higher seeds will always have the better matchups *if no upsets occur*. This is passable in a single-elimination bracket (such as March Madness), since a lower-seed inheriting the bracket positioning of a higher-seed only punishes the losing team. However, in a double-elimination bracket, upsets mean that a higher-seed inherits the bracket positioning of a lower-seed, which can easily screw teams over. Despite going 2-3 with the both 2 and 7 seeds, never losing a single round against any other team, and going into the tournament ranked in the top 12, we did not finish in the top 12.

Our proposed solution to this issue is to reseed when teams drop to the losers bracket. This means that instead of a rigid bracket, the higher seeds will always have the better matchups regardless of whether upsets occur. As a result, instead of the 2 seed and the 10 seed battling it out for losers top 12, the 2 seed would get reseeded to play the lowest available seed (which would've been the 17 seed). This better ensures the true top-12 strength teams qualify. This does punish the 17 seed however, since their matchup goes from a close game against the 15 seed to a brutal matchup against the 2 seed, so this system is not perfect either. Someone always gets the short end of the stick.

At the end of the day, we could give all the bracket-luck excuses in the world. However, that wouldn't change the fact that we couldn't win the sets that mattered most. **podemice**, **JDK? More like IDK,** and **placeholder name 2** all deserved to make the finals over us, and we have nothing but the highest respect for them. Seeding doesn't matter if you just win the damn games. In the words of Ichiro Suzuki, "I like imperfection. Because one is imperfect, it makes you want to keep moving forward."

[10] We think the qualifier stream should start earlier in the bracket, and stop once the qualifiers are determined. Otherwise, we just end up watching the finals twice, and not a single streamed game "matters" in the qualifier tournament

3 Conclusion

3.1 Reflection of our process

Overall, we exceeded all of our expectations this year. We did an excellent job of identifying reusable code from XSquare, and modifying/augmenting it for our own purposes. This allowed us to get out a high-quality bot much quicker than previous years.

From there, we did a good job of identifying key, game-specific tasks to implement, such as capturing towers, basic combat, and opening logic. This allowed our team to achieve good results quickly, as these tasks had the highest reward for the lowest effort.

Finally, we were able to make consistent improvements to our bot by identifying inefficiencies in resource management and idle time. These efforts were all it took for our team to make the leap to being a top-level team.

However, there were certainly areas where we could improve. Justin spent a lot of time re-doing XSquare code such as pathfinding when realistically, he couldn't really improve it and ended up wasting a lot of time. Andrew spent a lot of time on the SURVIVE behavior of bots, which was not an efficient use of his time since the SURVIVE behavior never really impacted the result of any matches. It was Matt's first year, so he was climbing the learning curve while also being busy with class.

Additionally, the way we ended up dealing with goal objectives was messy. Each robot was able to keep track of multiple different goal objectives, however, we didn't define a formal framework for how to resolve having multiple different goal objectives. As a result, our goal resolution was a bunch of unorganized if-statements that was difficult to modify/debug. We could've benefited greatly from a formal goal resolution framework, so we will give one in the next section.

Finally, we did not pay that much attention to other teams' strategies. Copying other teams' strategies is a Battlecode staple because it allows you to easily identify improvements for your own bot that other teams have conveniently tested for you. Our team neglecting this strategy resulted in a unique bot. However, it also resulted in our bot lacking obvious features such as ignoring refills, having a good build order, and having soldiers seek out enemy towers to kill.

3.2 Reflection on the game

The game this year was probably the best game we've taken part in. The 2025 game did a great job removing all of the game elements that made the last 2/3 years micro-heavy games while also adding new elements that made decision-making interesting.

Opening theory was interesting since choosing between double soldier openings, rushing, mopper openings, or even SRP/splasher openings was a non-obvious choice.

Paint management was a very interesting task since you had to balance normal pathfinding, staying on allied paint, and choosing when to paint tiles where no option was the obvious choice.

Build order being an essential part of the gameplan was back from 2021, since you always had the option to build every robot type and every tower type. Each team had to tackle difficult questions such as "when do we start building splashers?" or "when are defense towers worth it?"

SRPs were a great addition, since they gave access to an NP problem (a packing problem) that could be solved at any time for a small economic gain. However, since it was nearly always more worth it to capture more towers, it provided an interesting choice: explore for ruins, or build SRPs. Additionally, since finding the optimal SRP placement is an NP problem, it meant that every team could have slightly different approximation algorithms for SRP placement.

Overall, Teh Devs did a fantastic job releasing patches. They did a great job of nerfing overtuned strategies such as SRP spam or rushing while not making those strategies useless with the nerfs. However, we have some gripes with how they treated defense towers.

Historically, static defense in Battlecode was never a meta-relevant strategy. In 2022, watchtowers were weak since they were a large investment, which was a death sentence in that year's game since cheap units could snowball very quickly. However, the 2025 game was the perfect opportunity for static defense. Robots not being able to engage in direct combat with each other meant that offense and defense could be separate entities, i.e.: a good offense wouldn't just automatically double as a good defense, which would've given static defense a unique role. Additionally, map control was crucial to the game, so static defense having higher power in exchange for not being able to move would've been an appealing trade-off.

However, Teh Devs caved to early complaints about defense tower being too strong before sprint 1 even happened. The defense tower nerfs came too early in our opinion, since they never saw tournament results, and Teh Devs didn't have faith in the inherent weaknesses of static defense. In the end, defense towers just turned into slightly different money towers.

Static defense is a fascinating game aspect that we believe still hasn't been fully explored by Teh Devs. Static defense's polarizing strength is offset by a resource investment and the opportunity cost of investing in resource-economy instead. In RTS games, the counter to static defense is to disengage and gain an economic advantage since the opponent put a large investment in static defense. We believe this would add an extra dimension to any meta, since every Battlecode meta usually devolves into "always attack."

Overall, Teh Devs made massive improvements this year. They upgraded to a newer Java version, they added Python compatibility, they updated the client, and they made the best game in years. They deserve all the praise, and we are optimistic they can keep the good momentum into next year.

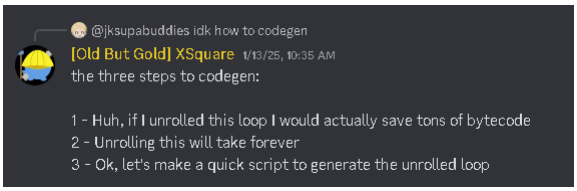
3.3 Advice

We have learned a lot through our years of Battlecode. This includes resources that we've compiled from across many teams' experiences. If you are reading this, it's likely you aren't actively competing. Here's some advice you can either actively follow outside of competition, or keep in mind for the next upcoming competition.

1. **Spend time preparing** - Before the competition starts, read Postmortems, look at code from other teams, and make a general plan for how you want to break up work amongst your team. You can even create a structure for your code ahead of time. For us, using a `Robot` base class and different utility classes (`MapData`, `Communication`, etc.) seems to have worked the best. Additionally, there are aspects of Battlecode that remain constant through nearly every year. They are as follows.

    - **The Map** - The Battlecode map is (nearly) always a coordinate-grid of size between 20x20 and 60x60 inclusive. For fairness, the map is always symmetrical either by rotation or reflection. Essentially all top teams have some sort of map data structure in every bot that stores all known information about the map. Being able to identify which of the 3 symmetries (reflection over x-axis, reflection over y-axis, and rotation) the map contains, and extrapolating known information with known symmetry is Battlecode 101. Example code for map representation and symmetry identification from **XSquare** can be found here and here. Example code for our map representation and symmetry identification can be found here.
    - **Bytecode** - The Battlecode engine has historically been written for Java, and it appears it will continue to do so. **Teh Devs** always put hard limits on computation to encourage competitors to come up with their own solutions instead of solely relying on well established algorithms. They enforce computation limits using Java bytecode. Historically **Teh Devs** have put bytecode limits of around 7500-1500 per robot where 1 bytecode is the approximate equivalent to a single assembly-level instruction. Even performing simple tasks such as breadth-first-search on a 20x20 tile map will use a robot's entire bytecode budget. As a result, understanding how bytecode works and knowing bytecode "shortcuts" is crucial to squeezing the most performance out of your puny bots. Additionally, having an understanding of bytecode will help you make decisions on whether an algorithm is feasible to implement under particular bytecode restrictions (ie: Can I implement A*? Spoiler: probably not). To read up on bytecode, we suggest this and this (section 6) as starting points.

- **Pathfinding** - Your robots need to know how to move from point a to point b in the most efficient path. Battlecode usually breaks down into either binary passability (ie: walls or no walls) or variable passability (ie: various amounts of rubble that slow down bots). In games with binary passability, most top teams default to using Bugnav. Here is XSquare's old implementation of Bugnav. In games with variable passability, most teams default to a greedy movement algorithm (ie: always move in the desired direction). However, many top teams optimize their pathfinding algorithms by making "unrolled"[11] versions of breadth-first-search or Dijkstra's/Bellman-Ford algorithm for binary and variable passability respectively. Here's **Just Woke Up**'s Python script for generating Java code for unrolled breadth-first-search.



- **Decision-making** - Your robots often have numerous goals they would like to fulfill. It may be tempting to tack on goal-types as you think of them, but this results in messy code that is difficult to modify. Additionally, it may be tempting to only store the current goal, but this results in your robots have very short-term memories (ie: they may forget their current task if they get initiated in a fight that overrides their current task). We propose the following framework:

  Goal Priority Queue - Each Goal contains the goal-type and the MapLocation where the goal is to be fulfilled. The priority for goals may depend on the game. For example, goal-types may be simple enough that they can be enumerated such that higher-numbered goals have a higher priority. However, if goal-types require more complicated priority, you can manually define them yourself.

  Adding, Executing, and Stopping Goals - Every goal type should have a shouldStartGoal, executeGoal, and shouldStopGoal methods. This modular design allows for easy modifications and additions to your goal framework.

  Note that this framework only applies to goal types that naturally interfere with one another. If a robot can fulfill multiple goal-types at once (ie: towers/archons attacking + spawning), those should each have their own priority queue.

- **Micro/Macro** - Micro and Macro refer to terminology from real-time-strategy games. Since Battlecode plays similarly to an RTS game, Micro and Macro are always applicable. Micro refers to fine-controlling singular robots, mostly to optimize robot-to-robot combat. Particularly, if a robot can fight another robot, there are certain movements/maneuvers robots can take to swing a fight in their favor. These include, but aren't limited to, weaving in and out of a robot's attack range, splitting robot groups to avoid area-of-affect attacks, kiting enemies, and pushing an engagement when a robot senses it has many allies. Macro refers to controlling army-level actions. These include, but aren't limited to, managing robot production, choosing when to expand, saving/spending your resources properly, and executing long-term strategic plans. While you can't explicitly plan a micro or macro strategy since they depend heavily on the game, it helps to have a framework for both. Here is the infamous **XSquare** micro, a java file that has crushed many-a challenger. Nearly every other top team has benefited from copying the **XSquare** micro to some degree, and here's why:

  (a) **When to initiate micro (doMicro)** - The conditions to initiate micro are usually whether your robot can see enemy robots. However, this can be expanded to include cases such as having low hp and having memory of nearby enemies that may be out of vision.

[11] Loops take extra byetcode overhead to run. If the number of iterations of the loop is always the same, you unroll the loop by manually writing every line of code the loop would've executed. This results in horrific looking code that is very bytecode efficient.

(b) **The micro array (computeMicroArray)** - The micro array contains the heuristic information of all 9 movement options (moving to the 8 adjacent squares or staying still). This allows for easy updates to the heuristic information and comparing heuristic scores via loop unrolling.

(c) **Heuristic information (MicroInfo)** - The heuristic information stores all factors that make a tile appealing/dangerous to move to (ie: number of nearby enemies, distance to closest enemy, etc). Then, once all the heuristic information is calculated, the isBetterThan method can easily compare two MicroInfos.

Note that this micro only involves movement and not the actual attacking. This is because decision making for who and when to attack is usually straightforward (target the lowest hp enemy possible, attack whenever possible). Also note that **XSquare** micro could be completely rewritten with our Goal Priority Queue framework (we might do this next year).

Macro is much more difficult to make a general framework for, since robot-spawning, resources, and high-level strategies vary drastically year-to-year. However, if you use ample communication, globally available info (such as map size), and the Goal Priority Framework, you should have no issue implementing the macro strategy you desire.

2. **Budget Your Time** - As mentioned previously, one of our big downfalls this year was spending a lot of time to perfect things that either don't need to be perfected. Battlecode is a short competition, and no one creates the perfect bot. Often the best tasks to do are the tasks with the best effort-to-result ratio. This frequently ends up being the simplest things, such as hard coding or using greedy algorithms.

3. **Prioritize Economy** - Our workflow generally revolves around economy. The first thing we always do is plan how to get a strong economy as soon as possible. Battlecode games are often snow-bally, where any small resource lead can very quickly turn into a large advantages down the line. As a result, highly optimizing the beginning of games for resources is the best way to kickstart your team's snowball. Once you have done this, you can branch off into working on converting the economy into the win condition. Additionally, **Teh Devs** have historically shown favoritism to teams that prioritize economy-based gameplans (as opposed to rush/attack based gameplans) by making maps in the finals tournament larger and slower.

4. **Learn from Others** - Watch replays, collaborate in the discord, talk with your teammates. Chances are you can learn something from every team out there, whether it is an insight or strategy they use, or some niche condition that breaks your bot. See where you are strong, see where you are weak, find out why you are weak, and figure out how to improve. Many top teams are very friendly, so don't be afraid to ask them what strategies they use on the Discord. Getting Battlecode clout is nearly as important as winning, so they will be happy to flex their superior Battlecode knowledge.

5. **Stand on the Shoulders of Giants** - Similarly to the above, use every resource available to you. There are many past repositories that have been posted on the Discord. We view **XSquare** as the God of Battlecode. He has several years of his past bots posted here, and has consistently been on top of the leaderboard. His micro, exploring, and pathfinding are all unmatched.

6. **Test your code** - We have never bothered to write unit tests, but we realistically should. At minimum, run several games, closely analyzing whatever behavior you changed. Make custom maps, designed to expose your problems. Save old versions of your bot and compete against them. Run scrimmages against other teams. The more your code runs, the more problems you will find and fix. DO NOT upload untested bots right before important submission deadlines. At minimum, test against older versions of your code.

7. **Use Git Effectively** - Git can be intimidating for first time users, but once you understand a few basic commands, it is really not too difficult. I cannot emphasize this enough, **learning command line git will save you and everyone you work with so much time and headache in the long run**. GUIs are good for the simple stuff, but you can break things in unimaginable ways by clicking

buttons that you don't fully understand. Your team should have an agreed upon criteria for branching, commits, and reviews so that everyone is on the same page. Our recommended git workflow is below[12]:

- When you have a new feature you want to add, create a branch for it. This can be done as follows:

  ```
  git fetch
  git checkout -b <branch-name> origin/main
  ```

  `git fetch` will update your local reference to the remote repository (hosted on GitHub or elsewhere), and `git checkout -b` will create a branch named `<branch-name>` from the most up-to-date version of `main` that is in `origin` (the remote repository). Note that depending on how you have it configured, the main branch may be called `master` instead.

- Make incremental progress on the code, and create a new commit every time you hit a "checkpoint". It is up to you to determine what that means, but I usually figure if I do a test run, its probably a good time for a commit. You can create a commit by doing the following:

  ```
  git add <path-to-changes>
  git commit -m "<your-commit-message-here>"
  ```

  `git add` will "stage" the changes at `<path-to-changes>`, which is basically "preparing them to be committed". If you don't want to manually add each individual file, you can do `git add .` to add all changes in the current directory, or `git add <path-to-folder>/*` to add all changes in a folder. If you do either of these, you should run `git status` after, and make sure the files listed as "staged" are the ones you intend to commit. If they are not, you can run `git restore --staged` with a path to remove it from the staging area. `git commit` creates a commit, and `-m` adds the commit message.

- Once you have finished the feature you created the branch for, and have made all your commits, you are ready to merge back into `main`. There are several ways of doing this, but our personal preference is using a rebase strategy, so we preserve a linear commit history without duplicate or merge commits. When you are ready to merge your code into main, do the following:

  ```
  git fetch
  git rebase origin/main
  git push -u origin <branch-name>
  ```

  While the `git fetch` and `git rebase` steps are not fully necessary, they will save you from many problems in the future, and it is usually better to run them anyways. `git rebase` moves the base of your branch to `origin/main`, i.e., it takes the changes you made, and tries to re-apply them on the most up to date main. **Be sure to read the output of the rebase command**, as it may tell you there are conflicts. If so, open up your editor, most of them will have a conflict resolution tab, and fix the conflicts by choosing what to keep, change, or take out. This will only be necessary if you have multiple team members in different branches working on the same part of the code. Once you fix the conflicts, run `git rebase --continue` and repeat if necessary. `git push` sends your changes to the remote (GitHub), and `-u origin <branch-name>` tells GitHub what the name of the branch should be. Make sure the `<branch-name>` you give is the same as what you have locally. It shouldn't cause any problems, but it will certainly be confusing.

- Your branch should now have the latest version of `main` with your new changes pushed on the end of it. To get these changes in `main`, you should create a "Pull Request" on GitHub. You can do this either by following the link that shows up after you push, or going to GitHub and opening the "Pull Requests" tab. Add a name and description, and create the pull request. Your teammates should be able to view and comment on your changes. If you configure things correctly (and pay a monthly fee which we did not), you can pretty easily set up rules for GitHub to restrict merges until approval quotas have been met. When you are ready to merge the pull request, there is a big green button that shows up. **We recommend using the "Rebase and merge"** strategy,

[12] We outline this in a lot of detail since we anticipate a lot of teams are using Battlecode to learn team programming, as we did in previous years.

so it preserves all the commits. This doesn't really matter most of the time, but it is annoying when you want to see what order you did things, or revert changes, and there are messy merge commits that make it hard to tell what was added.

- Make sure you run `git checkout main` and `git pull` to get the most up to date version of the code locally.

Again, this is just personal preference, and there are many other more detailed and thorough guides on similar strategies. What matters most is that you find something that your team agrees on and works for you. We have had similar success in the past with everyone battling it out on `main` with no branches, so with a small enough team, no strategy can work as well.

Obviously, there is more to Battlecode than just these points, but these are the biggest lessons we have learned and hope you find useful as well.

3.4 Until Next Year...

We hope you found this writeup useful, entertaining, inspiring, or positive in some other way. We have really enjoyed the ups and downs of Battlecode every year, and will definitely continue competing even after our eligibility is gone. Next year is definitely the last year for Justin, and probably the last year for Andrew, and we are really hoping to make something special happen. After that, we will make it our mission to win a sprint tournament, we need that discord role.

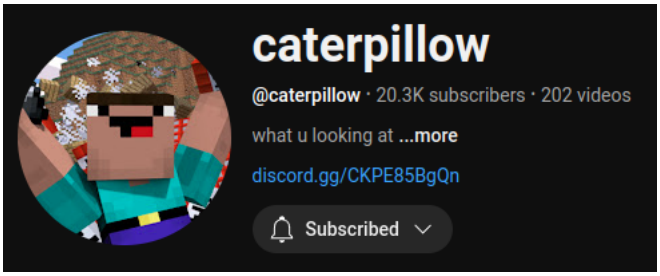


Figure 10: Subscribe to **caterpillow**

As always, thank you to **Teh Devs** for creating and running such a fantastic competition this year and every previous year we have competed. Huge thank you to all the other teams we mentioned previously in this document, particularly **XSquare** for his 10 year commitment to dominating the Battlecode leaderboard and open sourcing his code every year. Congrats to the winners, **Just Woke Up**, after an insane set of rematches against **Confused**, an impressive individual competitor, and **podemice** with the upset of the year in US Qualifiers.

We are incredibly excited for next year, and hope to continue our trend of improvement.