Battlecode 2026 Postmortem

Alex Thummalapalli
(Team food)

Table of Contents

1. Introduction

I'm Alex, a CS undergraduate at Georgia Tech. I placed in the top 12 at the Battlecode 2026 Finals as team "food".

As a solo competitor in my first year of doing Battlecode, I had to learn a lot in a very short time frame to build a competitive bot. I'm writing this postmortem to share the lessons and strategies I picked up along the way to help new competitors get up to speed faster. I'll mostly be going over the strategies I used for this year's game, but I'll end with some high-level thoughts that I hope will be more applicable to beginners as they figure out how to best approach Battlecode.

Feel free to check out my Battlecode repo, my LinkedIn, and my portfolio!

2. Game Overview

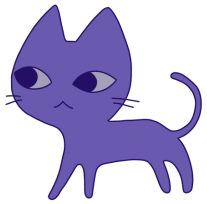
Battlecode is a month-long programming competition where teams iteratively design and develop AI algorithms to control virtual robots in a fast, turn-based strategy game. This year's theme involved warring rat factions fighting over territory, resources, and survival, all while contending with erratic NPC cats.

Figure 1. Map "squaresaremodernart" from US Qualifiers.

Units

Baby Rats are 1x1 units with a limited directional view. These units can move, attack, place traps, carry/throw/drop enemy baby rats, and pick up cheese (the main currency this year).

Rat Kings are 3x3 units with a larger view, but slower movement. These units can move, attack, and place traps as well. They also spawn baby rats by spending cheese. Destroying all enemy rat kings will win you the game, but you can spawn new rat kings using 7 baby rats. There can only be 5 rat kings per team, and they steadily consume cheese, losing HP when you run out.



Cats are 2x2 units with a 180-degree view. They are neutral units that target both teams indiscriminately. They can move, pounce, and attack all types of rats. They move between fixed, invisible waypoints spread across the map.

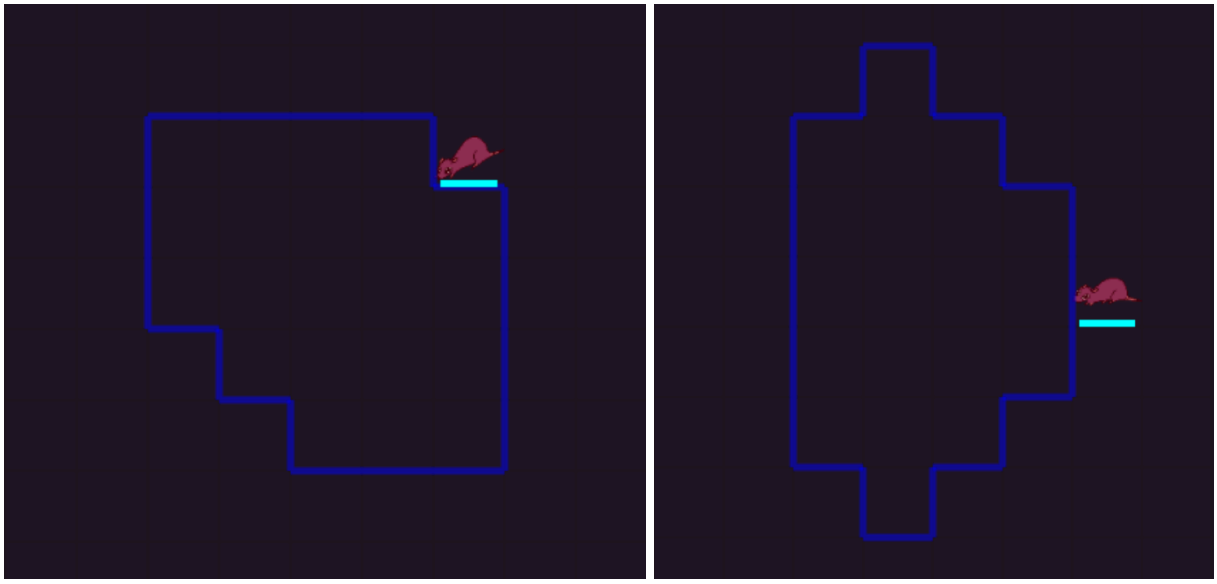


Figure 2. Baby rat vision.

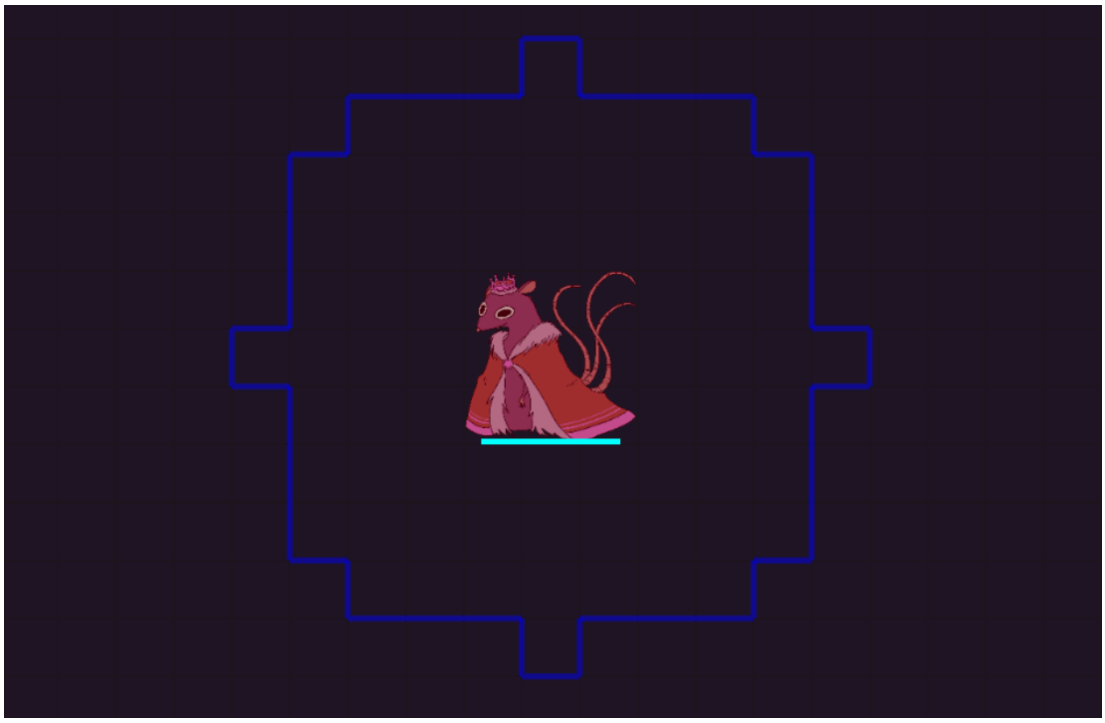


Figure 3. Rat king vision.

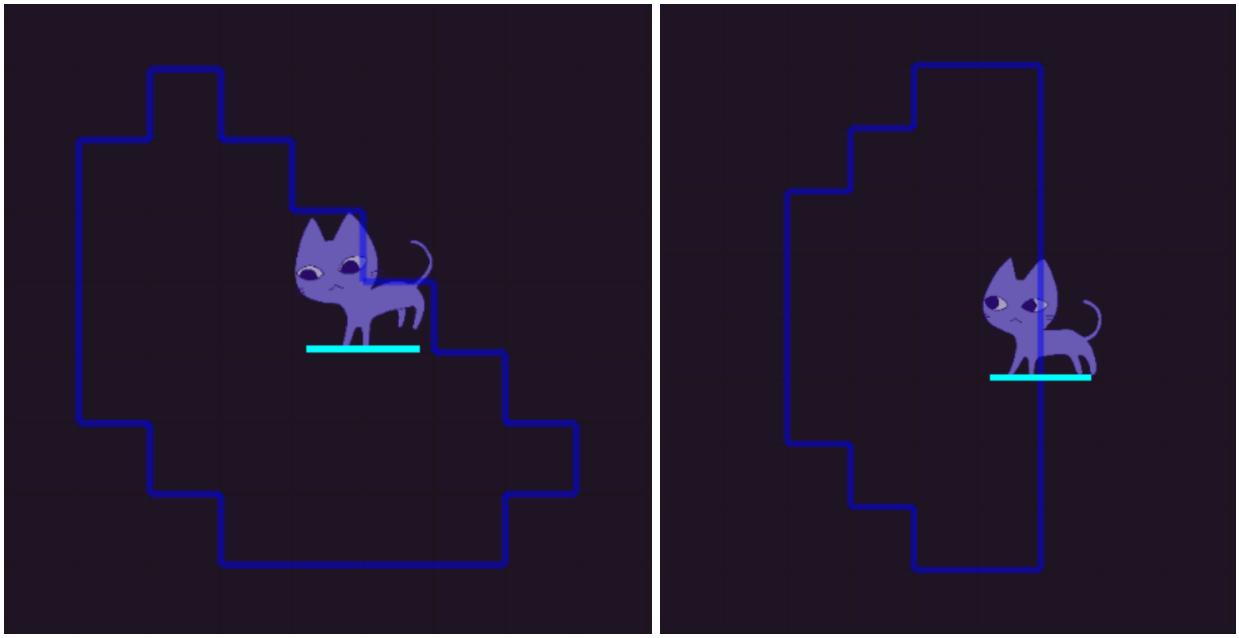


Figure 4. Cat vision.

Map

Maps consist of walls, dirt, and cheese mines and range in size between 20x20 and 60x60. Both walls and dirt block unit movement, but dirt tiles can be broken by all units, including cats, and placed by rats. Cheese will spawn randomly on tiles near cheese mines. Maps are guaranteed to be symmetric, either vertically, horizontally, or rotationally.

Figure 5. Key map elements and units.

Figure 6. Map “winter” from US Qualifiers. An example of a rotationally symmetric map.

Scoring

Another unique feature this year was the cooperation/backstab system, where teams initially work together to kill the cat NPCs, until one team attacks the other, upon which the win condition changes to killing the enemy rat kings.

In practice, most games ended in backstab mode; the payoff for early aggression (map control, resource denial, etc) pretty much always outweighs the uncertain outcomes of cooperation. But on certain maps where it is difficult or impossible for teams to interact, it is possible for a match to end in cooperation.

If a match lasts 2000 rounds, or all the cats are killed in cooperation, teams are scored based on percentage of cheese transferred to rat kings, number of rat kings, and damage dealt to cats, with different weights in cooperation and backstab mode.

Communication

Rat kings can share information with all ally rats by writing directly to a shared global array of 64 10-bit integers. Rats can also 'squeak' a 32-bit integer that can be heard by any nearby ally rat, as well as cats (which will target the source of the squeak!).

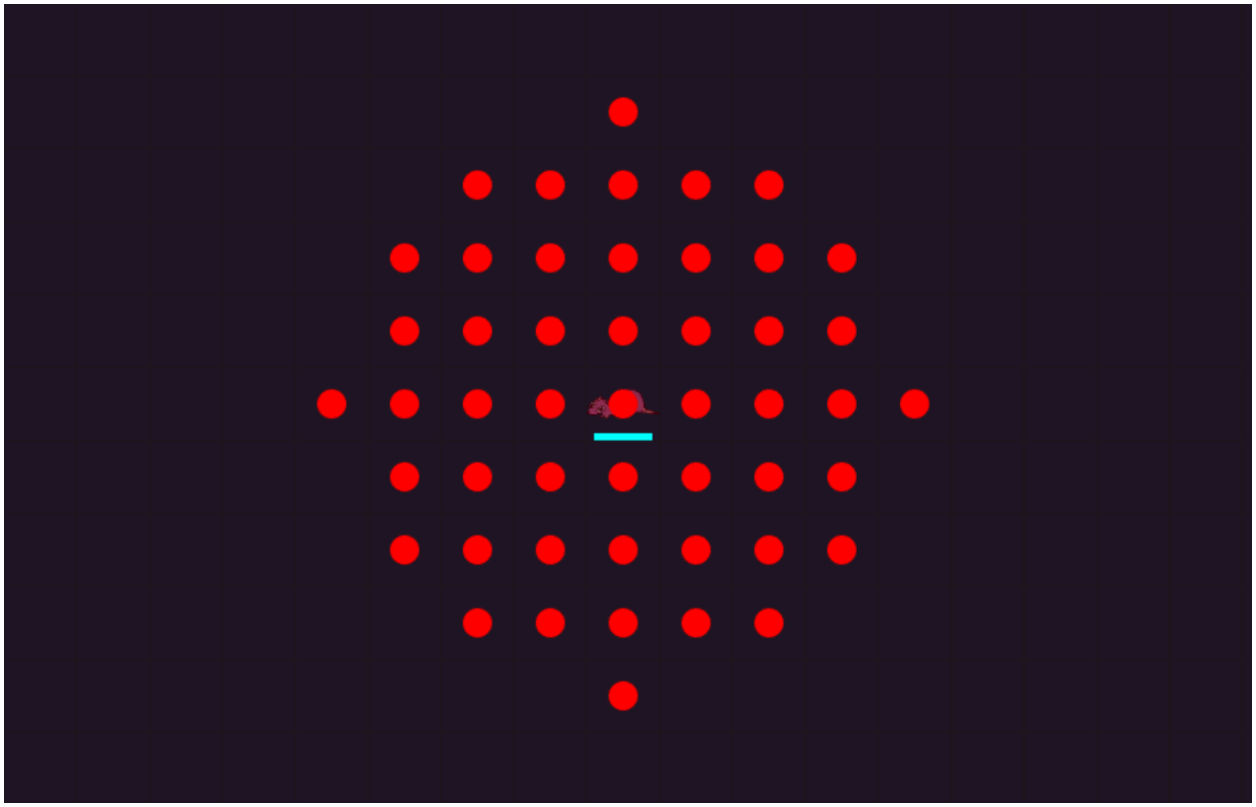


Figure 7. Squeak range. Any ally rat on a red dot tile can hear the plum rat's squeak.

If you want a more in-depth overview of this year's game, you can find the specs here, the API reference here, the scaffold repo here, and the client repo here.

3. My Experience

Timeline

Before the game specs were released, I spent a couple of days reading postmortems and scouring through codebases from past years. This proved exceptionally beneficial in understanding how Battlecode works and how top teams approach the game.

In the week leading up to sprint 1, I built a rush bot that prioritized rushing the three possible enemy spawns (via the three possible symmetry modes), narrowing down the

enemy rat king's location by eliminating possible symmetries, and then swarming it as quickly as possible.

In the week leading up to sprint 2, I completely rewrote my bot code, focusing on building a cleaner codebase and an XSquare-esque heuristic micro system.

In the last two weeks of competition leading up to US Quals and Finals, I started an internship and found myself with far less free time than I hoped. Hence, I mainly worked on improving my existing systems, focusing on optimizing pathfinding, cheese collection, micro combat, etc. I also did some refactors to fix issues that had popped up.

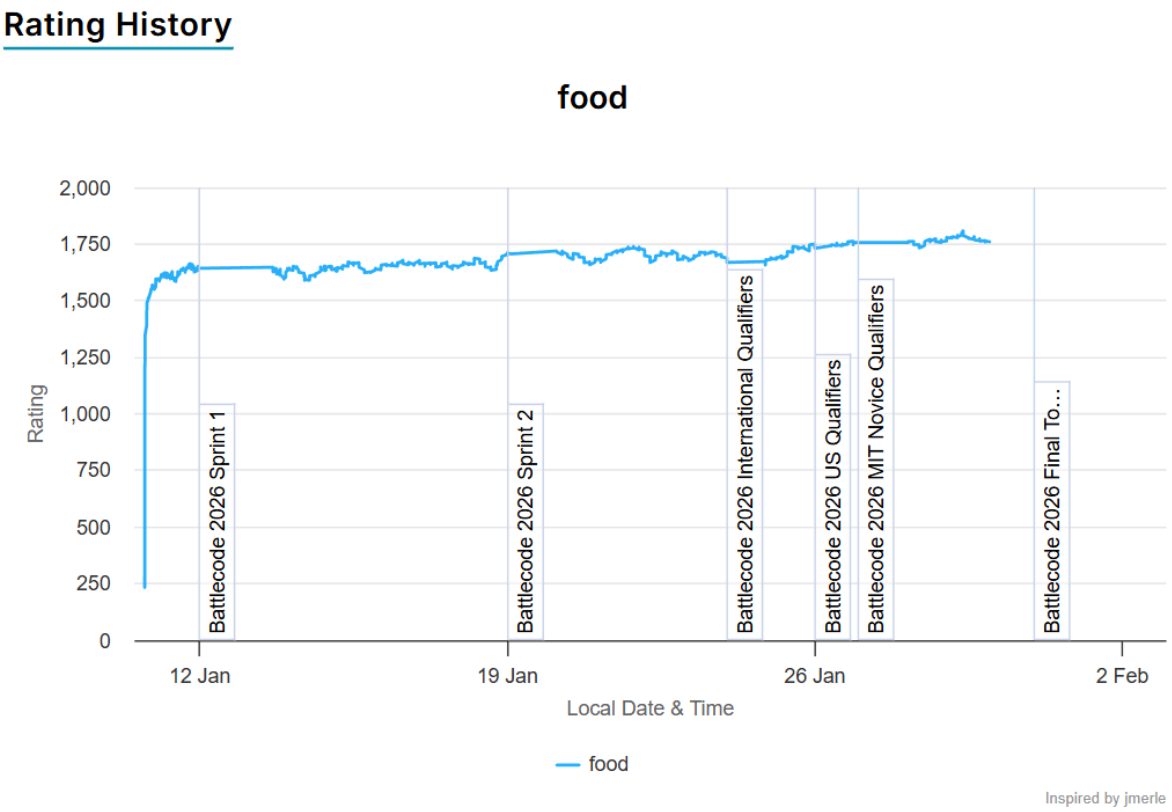


Figure 8. My rating over the course of the competition.

Results

I placed top 32 in Sprint 1, top 16 in Sprint 2, 5th-6th in US Quals, and 9th-12th in Finals.

At the final competition, I went in as the 10th seed. I lost 2-3 to Mango Tango (7th seed, 5th-6th overall) and 2-3 to Muscallonge (4th seed, 3rd overall). My sole win was 3-2 against Lorem Ipsum (15th seed, 13th-16th overall).

Going into Finals, I felt that if my bot were just 10% better, I'd stand a real chance of contention. This feeling was highlighted by the two narrow losses I faced against high-ranking teams; if I had flipped just one map, I could have gone on a much better run. Nonetheless, I'm quite proud of the results I was able to achieve as a first-time Battlecode participant, and I plan to come back next year and do even better.

4. Strategy

Macro

State Machine

Every turn, baby rats choose 1 of 5 possible states:

1. **FIND_KING:** Navigate to a known enemy king or a possible enemy spawn location.
2. **COLLECT:** Navigate to the nearest visible cheese or known mine location.
3. **EXPLORE:** Navigate to a remote or unvisited map location.
4. **BUILD_RAT_KING:** Navigate to a desired formation point.
5. **RETURN:** Navigate to the nearest ally king.

My goal was to have as few states as possible. This makes it easier to:

1. Debug state-driven behaviors and state transitions
2. Share behaviors across situations (i.e. baby rats act the same way when returning to the king, with no need for separate RETURN_CHEESE and RETURN_INFO states)
3. Coordinate global strategies since more rats will be in the same state
4. Reason through basic principles for how rats should act

I could have further condensed states by combining FIND_KING and EXPLORE, since I moved away from my initial rush strategy. I just never got around to changing this.

We decide state by checking each condition in the following order:

1. **BUILD_RAT_KING** if there is a desired formation location
2. **COLLECT** if baby rat just spawned and either:
    a. We are low on global cheese
    b. Coin flip (50% chance)
3. **FIND_KING** if baby rat just spawned and lost coin flip

4. **RETURN** if either:
    a. We are carrying an ally baby rat that is carrying cheese
    b. We detect a nearby ally king sending an SOS
    c. We determined map symmetry and it isn't yet globally known
    d. We have at least 40 cheese
    e. We have any cheese at all and are low on global cheese
5. **FIND_KING** if we know the enemy king location
6. **COLLECT** if either:
    a. We see cheese and aren't already returning
    b. We are running low on global cheese
    c. Coin flip (50% chance)
7. **FIND_KING** if
    a. We lost coin flip, are already in **FIND_KING**, and haven't checked all possible enemy spawn locations
8. **EXPLORE** if none of the above apply

This state decision mechanism results in shifting behavior over the course of a match. In the early game, half our rats will be in COLLECT and half will be in FIND_KING, but with no known mines, this is effectively the same as an EXPLORE / FIND_KING split. Over time, almost all of our rats will be in COLLECT or RETURN, and since we store known mine locations in the global shared array, rats will mostly frequent already explored areas.

This heavily biases toward exploration and enemy king rushes in the early game and survival in the late game. Since most games end before tiebreaker anyway, survival is the most important element of the macro, and there is no need to rush an enemy king unless doing so is convenient. That being said, moving toward possible enemy king spawns has the added benefit of helping us fight for map control.

Target Destination

Once we reach a destination or switch states, we choose a new target based on state:

1. **FIND_KING:** We navigate to a sensed or squeaked enemy king location. If we haven't found the king, we randomly check each possible enemy king spawn, based on the symmetry of the map.
2. **COLLECT:** We navigate to the nearest sensed cheese tile. If we don't sense any cheese, we randomly navigate to a discovered mine. We mark mines as on cooldown if we don't find cheese there or there is an ally king nearby. If there are no mines left to check, we use EXPLORE logic.

3. **EXPLORE:** Before turn 200, we randomly choose a remote location (corners, edge midpoints, or center). After turn 200, we choose a nearby unexplored location.
4. **BUILD_RAT_KING:** We navigate to the formation point stored in the global shared array, or heard from a squeak.
5. **RETURN:** We navigate to the nearest ally king, based on info in the shared array.

Infrastructure

Symmetry Detection

For each tile we sense, we check if the corresponding tile in each of the three possible symmetry types (horizontal, vertical, rotational) matches. If there is a mismatch (i.e. we find a dirt tile in the upper left corner and a wall tile in the lower right corner, which violates rotational symmetry) then we eliminate that symmetry type.

Once 2 possible symmetries are eliminated, we mark the symmetry as determined and return to the nearest rat king to communicate it. It is then added to the shared array.

Communication

Rat kings can write to a shared array containing 64 10-bit integers. I used the shared array to store the following information:

Slot	**Purpose**
0	Map symmetry
1	Starting king position
2-6	Current king positions of up to 5 kings
7	King heartbeat
8	King SOS
9	Formation location for new rat king
10	Number of discovered mines
11-63	Discovered mine locations

Since there are up to 3600 map locations (maps can be up to 60x60) but only 1024 possible integer values in 10 bits, you can't store an exact map location in a single slot. Thus, I stored a lossless location if the map is small, and a lossy location otherwise.

If a king senses an enemy, it toggles a unique bit in the king SOS integer to indicate it is in danger. Every turn, each king also toggles a bit in the king heartbeat integer. If a different king notices the bit wasn't toggled, it clears the deceased king's location and SOS bits. This ensures that baby rats don't get stuck returning to or defending a dead king.

There was a possible race condition where a king could exceed bytecode, not be able to toggle its heartbeat, and have its location cleared by another king. Usually this is fine; the first king just sets its location again the following turn. In an unlikely scenario, a king could exceed bytecode, have its location cleared, and have its spot in the shared array taken over by a newly created king; however, I never saw an instance of this happening.

I also used baby rat squeaks to communicate information. When close enough to an ally rat king, baby rats squeak important new info (determined symmetry or newly found mines). The king adds this info to the shared array for all baby rats to access. Baby rats also squeak rat king formation locations if they are in BUILD_RAT_KING mode and squeak enemy rat locations if they sense an enemy rat. If a rat hears an ally rat squeak the location of the enemy rat king, it will propagate the squeak by "resqueaking" the enemy king location if it is further from the enemy king than the ally it heard squeak.

Figure 9. Map "stuckinthemiddle" from US Qualifiers. Only two cheddar baby rats see the enemy plum king. However, when they squeak the enemy king location, other ally baby rats start squeaking the enemy king location as well to propagate the squeak further away.

Baby rats avoid squeaking when sensing a cat to avoid being targeted.

Squeaking enemy locations greatly improved my combat micro, since baby rats could still detect enemies when facing away from them, as long as a nearby ally rat senses the enemy.

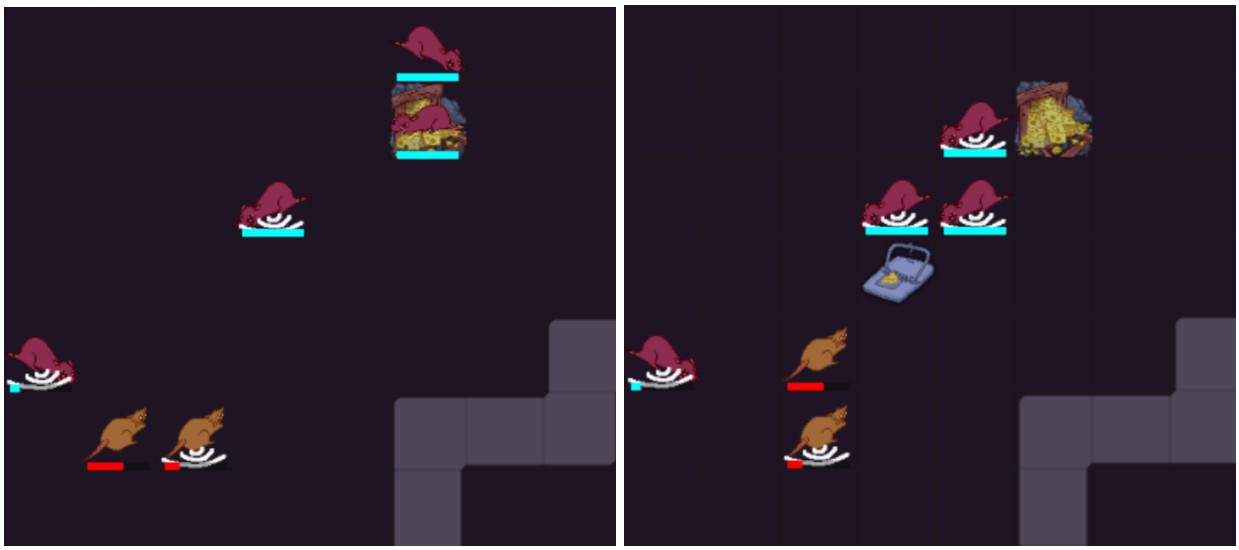


Figure 10. Map “warhammer” from US Qualifiers. Two plum rats see enemy cheddar rats and squeak their location. The two plum rats in the upper right can’t directly sense the enemies, but they hear the squeaks and move to enter combat.

Pathfinding

Both baby rats and rat kings use bugnav copied from XSquare. I considered implementing a more advanced pathfinding algorithm; however, many of the tournament maps favored greedy pathfinding. Since baby rats can transfer cheese across walls, there were frequent situations where baby rats had to navigate through a maze to get out of an unfavorable spawn location, but could simply get ‘close enough’ to an ally king without going through the maze again. With my time stretched thin, I kept bugnav for the entire competition.

Figure 11. Map “Hike” from Finals. Rats must navigate a convoluted maze to escape the spawn area, but can easily transfer cheese without entering the maze again.

Another shortcut I took was to assume dirt tiles were passable if we have enough cheese and aren’t on action cooldown. It is cheap to dig dirt, and we mostly dig in highly frequented locations, so I didn’t worry too much about optimizing the digging logic.

One bug I noticed was that when encountering a large dirt patch, rats would start digging and then immediately turn around and bugnav around the dirt patch. I realized later this was due to my dirt passability workaround; since digging dirt takes 25 action cooldown, rats would dig, move forward, and then perceive the way forward as impassable the next turn after the cooldown took effect. I addressed this by having rats wait after digging if the next tile forward was also blocked by dirt.

Figure 12. Map “RUN” map from Finals. Rats dig straight paths through dirt when navigating to a fixed destination, waiting each turn for the action cooldown to be ready.

If I had more time, I would have checked reachability in the pathfinding. With my current implementation, there were situations where rats would continually target cheese mines that either can’t be reached, or can only be reached through a longer winding path.

Micro

Entering Combat

Baby rats switch to using micro when they sense a cat or enemy rat. We don’t use micro if enemy rats are unreachable (on the other side of a wall, for example).

If we don’t sense an enemy, but an ally rat squeaks an enemy location, we turn to face the nearest such location before checking if we should enter micro. If we sense no enemies and hear no squeaks, but take damage, we move forward and turn around if the damage is equal to cat scratch damage, and just turn around otherwise. This lets us position toward possible assailants and enter micro if necessary.

When entering micro, we always attack greedily, then turn/move if needed, then attack greedily again. When attacking, we first try throwing a carried rat, then ratnapping, then biting, then placing a trap. However, we won’t place a trap before moving if we think we can ratnap after moving.

Combat Theory

This year's game was challenging because the optimal strategy for a baby rat to win in combat is to just never move within 8 radius squared of an enemy rat. With ratnapping and biting alone, there are situations where you can safely position yourself relative to the enemy. But once you factor in enemy rat traps, which you cannot see and will trigger just by moving into an adjacent tile, there is no way to guarantee a win in 1v1 rat combat.

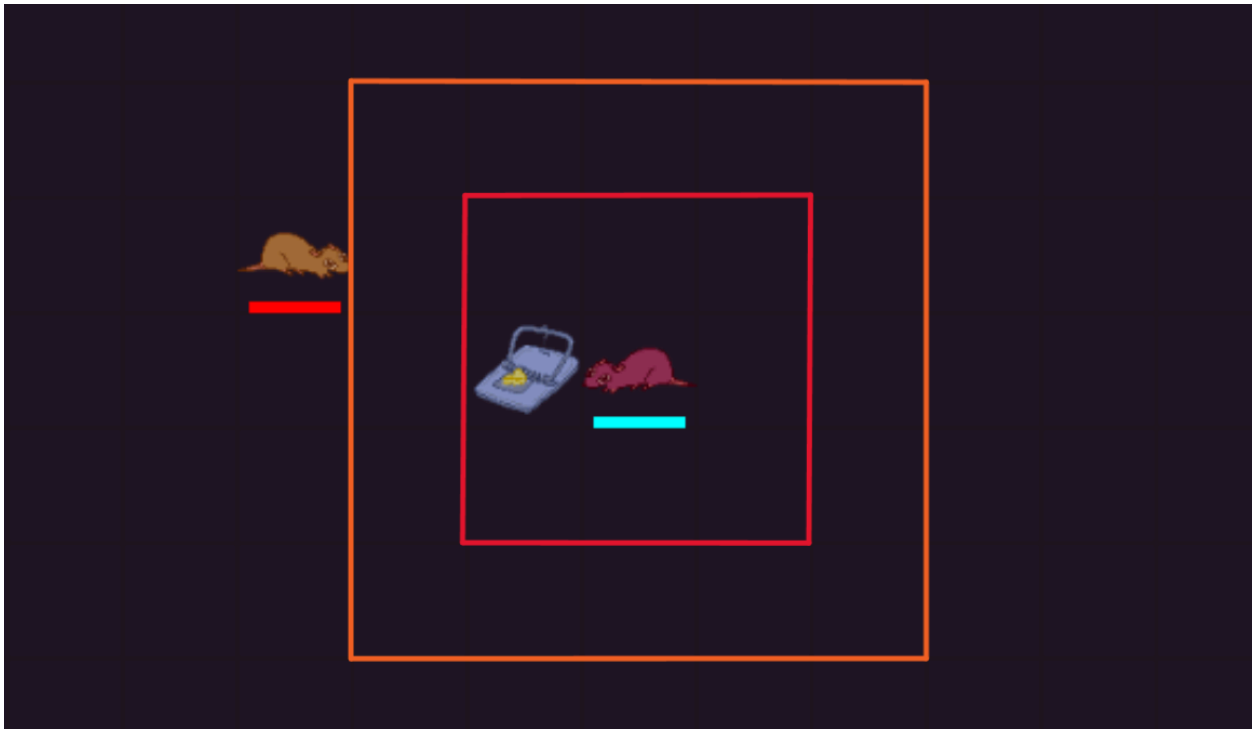


Figure 13. If the plum rat can turn, it can attack anywhere in the red square (2 radius squared). If it can also move, it can attack anywhere in the orange square (8 radius squared). In the example above, the plum rat placed a trap that will detonate once the cheddar rat steps forward.

Generally, baby rats can play aggressively (moving to ratnap or bite when possible) or passively (waiting for the opponent to move first). If one team plays aggressively and the other plays passively, the passive team will usually win out in combat by taking advantage of enemy rats moving into their traps and attack range. However, if two passive teams play each other, you'll end up with rat standoffs spanning up to hundreds of turns as the two opposing sides refuse to step closer to each other. This is bad since you give up substantial map control and have a harder time collecting cheese if your rats are locked in combat.

Figure 14. Map “tiny” from Sprint 2 tournament. Both teams’ baby rats are locked in a standoff, refusing to break the dirt separating the teams and risk a counterattack.

Figure 15. Map “corridorofodoomanddespair” from Sprint 2 tournament. Despite outnumbering the cheddar rat four to one, the plum rats refuse to move forward.

The solution is to balance aggression and passiveness. You should play aggressively when you have the numbers advantage or good positioning, and passive otherwise. Even if you

end up losing rats by playing aggressively, your opponent will also, and then you can win with smart economy management and map control as you spawn new rats.

Combat Movement

If we decide to enter combat, we calculate a heuristic score for each of the 9 possible movement directions (corresponding to moving to the 8 adjacent tiles, or just staying still).

The actual scoring mechanism is fairly complicated. We negatively weight:

- Impassable tiles
- Dirt tiles
- Tiles with adjacent or nearby cats
- Tiles with adjacent or nearby enemies, especially if they are higher HP
- Tiles in the path of a sensed flying rat
- Strafing (if movement cooldown is already used and you aren't facing that tile)
- Tiles that are cardinal to an enemy rat

This last point is especially interesting. Consider an enemy rat standing directly east to you. It could choose to stay still, move up to the northeast adjacent tile, move down to the southeast adjacent tile, move diagonally to the north adjacent tile, or move diagonally to the south adjacent tile. Since baby rats have limited visibility, you can only face up to 3 of the 5 adjacent tiles that the enemy could occupy. As a result, an enemy rat standing in any cardinal direction (north, south, east, west) can always move and ratnap you on the following turn, as long as it isn't on movement cooldown and is still alive.

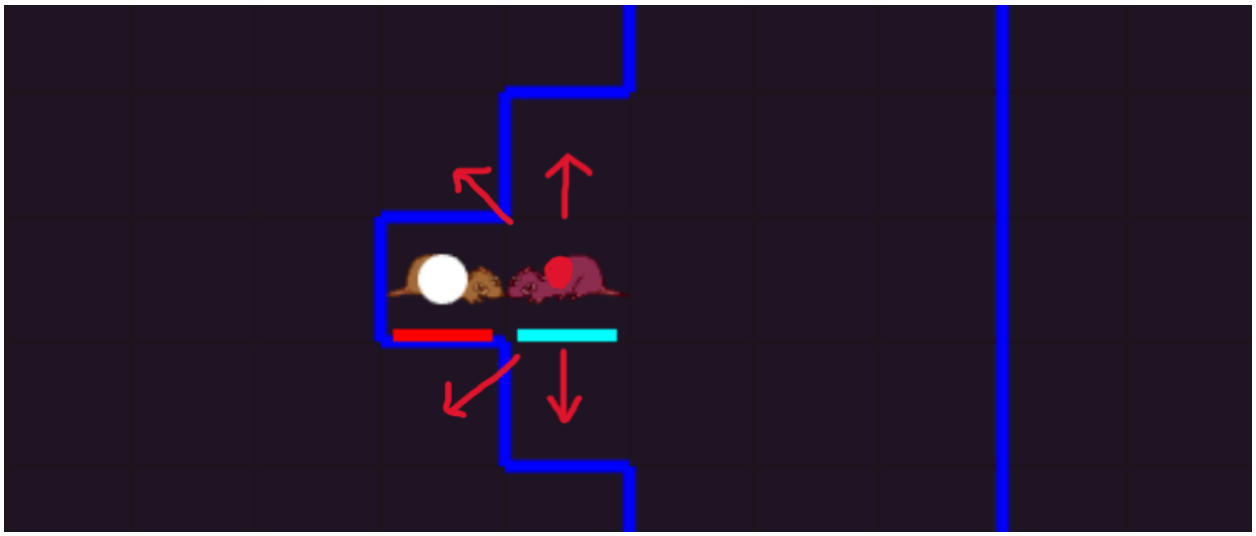


Figure 16. The plum rat can move to 4 possible tiles, or stay on the same tile. The cheddar rat can only see 3 of these tiles, no matter how it turns (the blue outline is the cheddar rat's vision).

We positively weight:

- Tiles where you can ratnap, finish, or attack an enemy (if action cooldown is ready)
- Tiles adjacent or nearby to an enemy rat king
- Tiles adjacent to an enemy you just threw
- Tiles closer to your destination

This first point is the most important consideration. We prioritize in the following order:

1. Ratnapping due to position
2. Ratnapping due to HP
3. Attacking enemy king
4. Finishing enemy
5. Attacking enemy

We weight ratnapping due to position (enemy is looking away) higher than ratnapping due to HP (enemy is lower HP) since moving could trigger a trap, deal substantial damage, and change whether you can ratnap the enemy.

We also weight “finishing” moves (where we will deal enough damage to kill the enemy) higher to prevent the enemy unit from being able to do another attack next turn.

After scoring each of the 9 moves, we choose the move with the highest score. We then turn and move appropriately.

Figure 17. The cheddar baby rat senses a cat in front of it. It scores the 9 tiles it can move to and decides to retreat to the left.

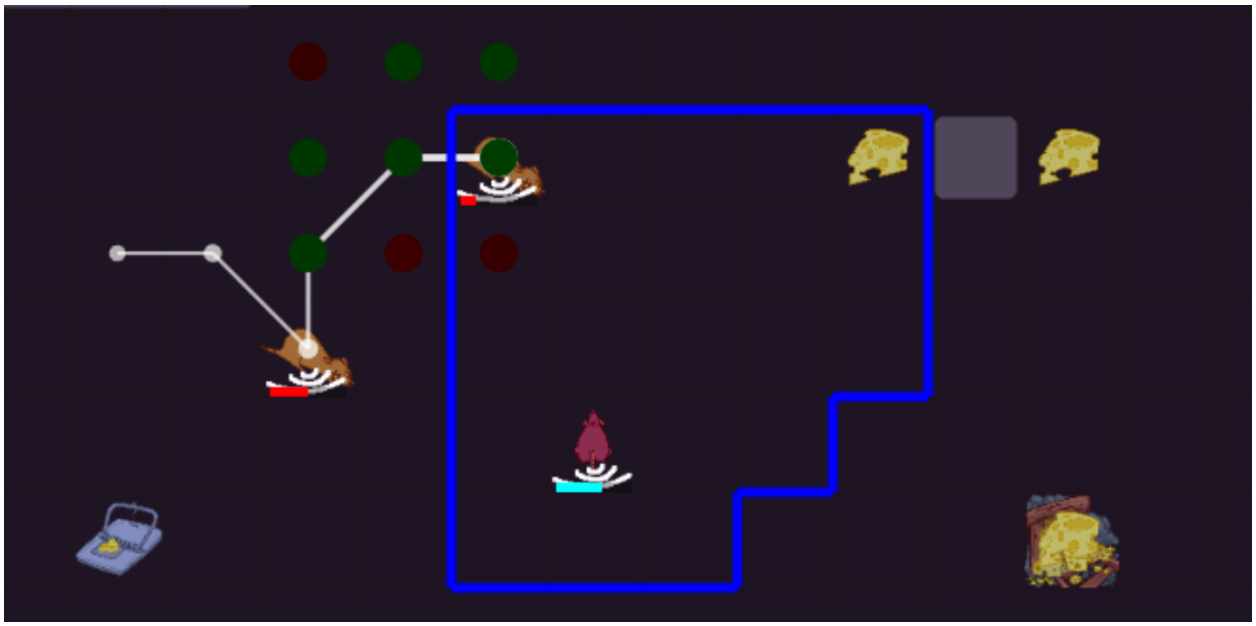


Figure 18. The cheddar baby rat is hesitant to move within attacking range of the plum rat, but it is trying to reach the cheese mine and doesn't want to retreat to the upper left. Thus, it decides to move right.

We calculate two possible turns when making our move; turning to face the desired direction, and turning to defend against enemies. This defensive directioning is essential to making sure we don't move and put an enemy rat in our blind spot, since they could ratnap us next turn. We prioritize facing the defensive direction at the end of our turn, and we decide whether to turn or move first if we can avoid strafing in the process.

One interesting strategy I implemented was inverting enemy proximity penalties when a king SOS is active and the baby rat is near the king. This results in baby rats aggressively moving toward enemy rats instead of avoiding them, diverting attention from the ally king.

I also halved the enemy proximity penalties when allies significantly outnumber enemies (2 to 1 or more) to make rats more aggressive when they have the numbers advantage.

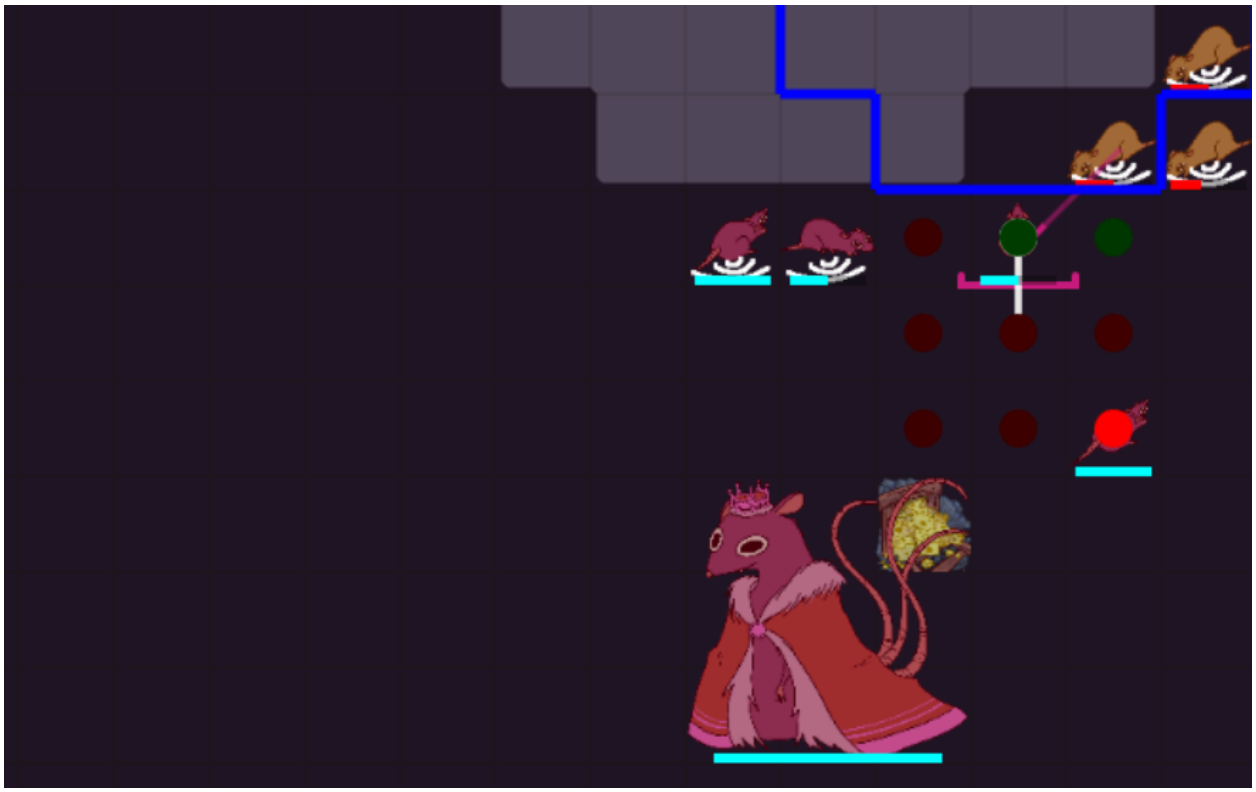


Figure 19. Having recently heard a king SOS, the plum rat decides to move up toward the enemy cheddar rats, despite being outnumbered.

Regular Movement

If we don't enter combat, we move according to our bugnav pathfinding. When moving in a straight line, we alternate between moving first and then turning, and turning first and then moving. This allows us to look left or right of the direction we are actually moving in, giving us additional visibility without the need to strafe.

If we are being carried by an enemy or thrown, we similarly look in different directions to try to gain as much information as possible.

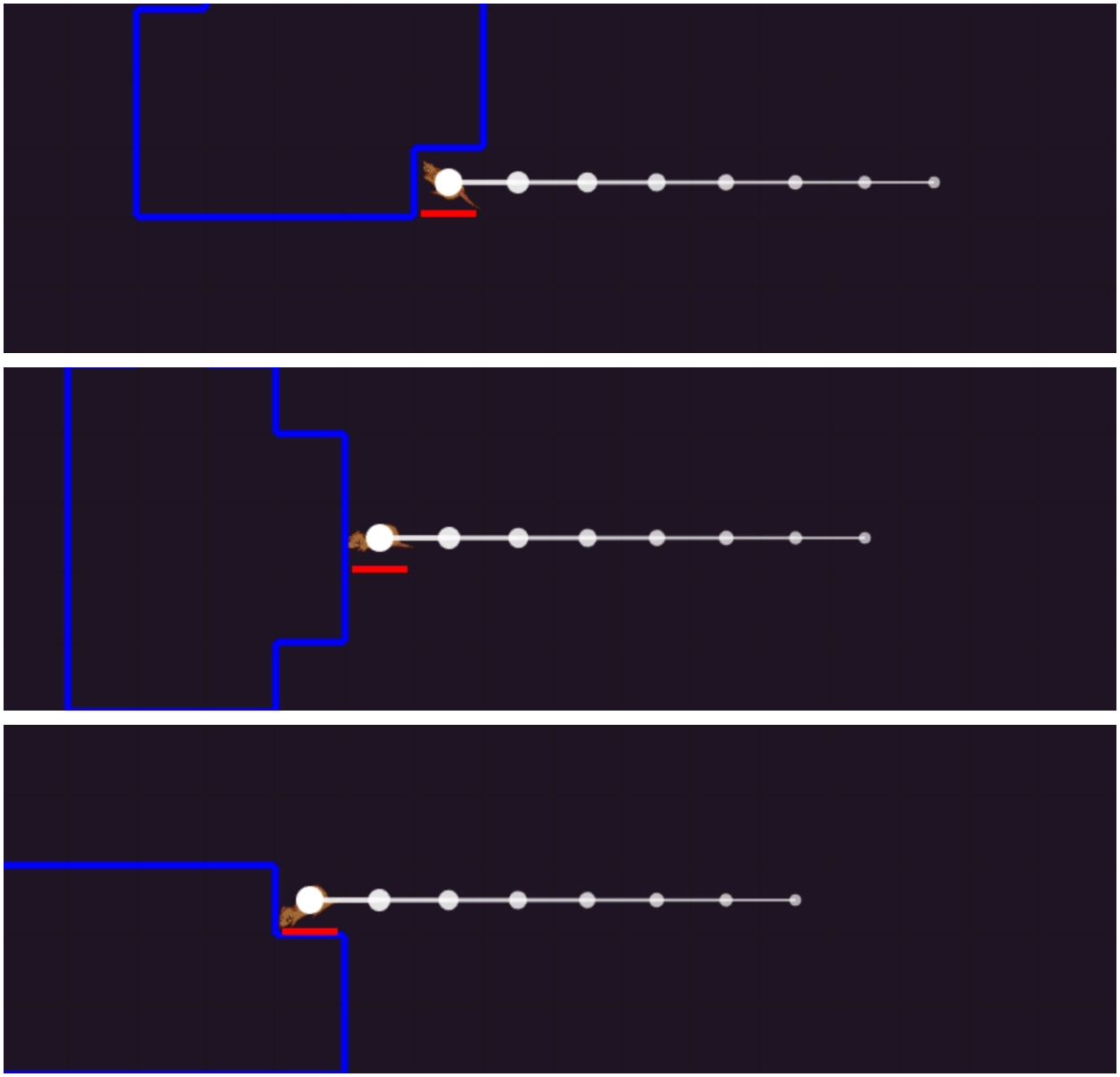


Figure 20. While moving in a straight line, rats alternate between looking left and right.

Ratnapping

We prioritize ratnapping the highest HP enemy we can carry, to take out the biggest threat to us. We also prioritize ratnapping an enemy that is carrying an ally rat, to free that ally.

Once we are carrying an enemy, we prioritize throwing them toward other enemies, then toward walls/dirt. If we go several turns without a good throw direction, we will throw away from the nearest ally king, as long as we aren't facing an ally. This last-resort throw ensures we don't accidentally drop or throw an enemy next to our ally king.

We will also occasionally ratnap allies. If you are carrying another rat, there is a movement cooldown multiplier of 1.5. This is the equivalent of carrying 50 cheese. So, if we aren't carrying cheese ourselves, and sense an adjacent ally carrying 100 cheese or more, we pick them up and enter RETURN mode. This allows us to return cheese more quickly to the king. We set the threshold to 100 instead of 50 so that we don't waste a rat's time for minimal gains.

If we enter combat while carrying an ally, we drop them behind us to focus on combat.

Cheese-Boosted Attacks

Rats will spend extra cheese to finish off an enemy; i.e. if the enemy is 13 HP, rats will spend 5 extra cheese to deal 13 damage. Rats will also spend extra cheese to prevent having less health than the enemy. For example, if the enemy has 100 HP and we have 87, we will try to deal 13 damage so that the rats have equal HP next turn. This prevents the enemy rat from being able to immediately ratnap us. We only spend extra cheese when we have enough global cheese, to prevent us from running out.

Previously, I would always invest 2 extra cheese to deal 12 damage with every bite. The idea here was minimal cheese investment adding up to give us the HP (and thus ratnap) advantage over enemies. However, I removed this code after watching many matches and realizing that other sources of damage (throwing, traps, multi-rat attacks) were dealing substantially more damage and a 2 HP advantage was essentially meaningless.

Placing Rat Traps

Traps only trigger when an enemy moves onto the trapped tile or an adjacent tile. Accordingly, they are mostly useful for defense if you plan on staying still or retreating and think the enemy will move forward. This is why I prioritized placing traps only after checking every other method of attacking, since we don't want to place a trap if there is a more direct means of combat.

When placing a trap, we prioritize tiles that are adjacent to multiple enemies, since it is more likely for an enemy to move within range and trigger it. But since the 50 damage + stun effect is so valuable, we will place a trap even if there is just one enemy nearby, as long as we have enough cheese. This is because traps only cost 30 cheese and almost always guarantee the kill when triggered. If we assume the other team tries to maintain 12 to 16 rats, spawning a new rat will cost them between 40 and 50 cheese, so placing a trap to get a kill is a good trade.

Fighting Cats

If there are no sensed enemy rats or squeaked enemy rat locations, baby rats will also bite adjacent cats. If we are still in cooperation mode after round 600 or hear a king SOS, we will also place cat traps near cats.

In retrospect, I wish I had made more aggressive cat attacking behavior. I mostly ignored cats when designing strategies, since cat behavior was so erratic for most of the competition. I only added the round 600 check to account for possible forced cooperation maps. However, I think it would have been smarter to just always attack cats with cat traps to rack up cat damage for tie breakers and reduce the number of baby rats dying to cats.

Figure 21. Map “t_minus_five_minutes” from Finals. The row of wall tiles prevents teams from interacting, leading to a forced cooperation match.

Economy

Collecting Cheese

We start returning to the king after collecting 40 cheese. We will collect up to 100 cheese on our way back, with the cheese limit increasing as we get closer to the rat king. We avoid collecting large amounts of cheese very far away, since collecting cheese slows down rats.

Rats will pick up any amount of cheese if within range, but won't go out of their way to pick up less than 5 cheese. This change was made after seeing some teams in the Discord server discuss picking up 19 cheese instead of 20 to confuse enemy rats.

When an ally king is sensed, we try transferring cheese to all 9 body tiles. This doesn't take much bytecode and makes it easy to transfer cheese over walls and across long distances without need for a more sophisticated algorithm.

Rat kings will also attempt to collect cheese. They will navigate to the nearest discovered mine, unless they find enemies at that mine location. Rat kings will also navigate to sensed cheese if nearby. These two behaviors allow kings to play an important role in cheese collection and economy management, so baby rats can focus on collecting cheese from other mines or attacking.

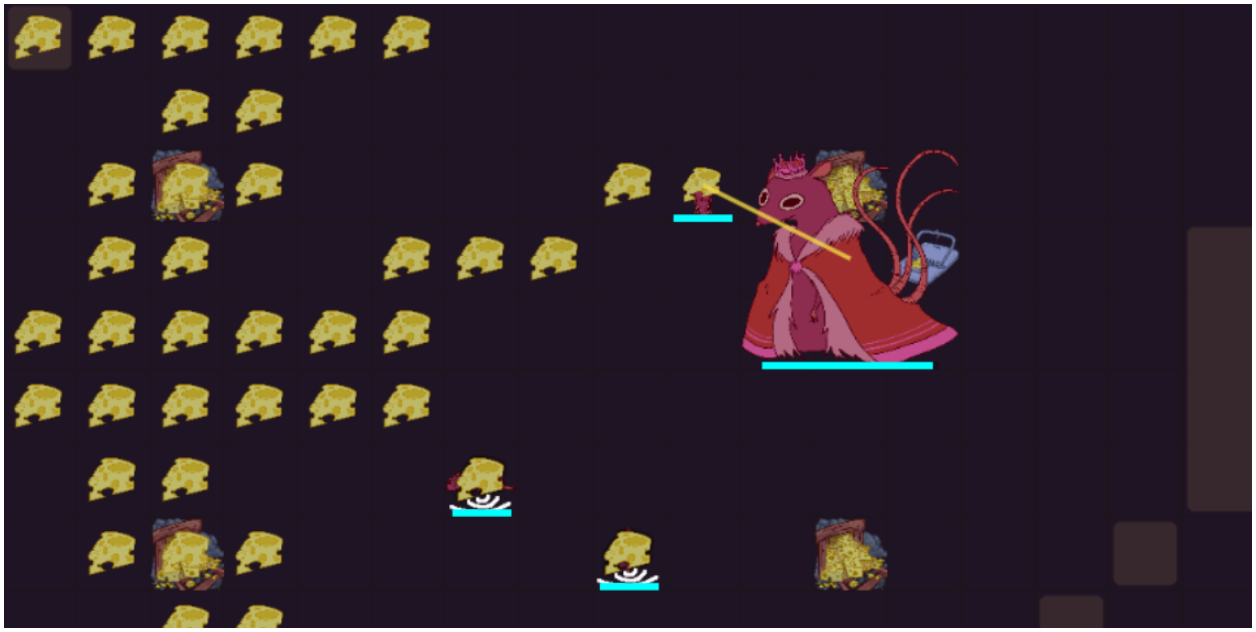


Figure 22. Map "Hike" from Finals. By positioning near a cheese mine, a single rat king maintains control of the cheese-laden middle of the map.

Spawning Rats

Our first rat king spawns rats toward the center of the map, to bias toward center map control. Subsequently formed rat kings will spawn in random directions. One minor optimization I wanted to make is choosing the baby rat's initial target location based on where it spawned in relation to the rat king, but I ran out of time to implement this.

We spawn rats based on global cheese thresholds. We try to always have 16 rats alive (we estimate this using the current cost to spawn a rat, which scales with rat count). The more cheese we have, the more rats we spawn. If we get to round 1950, we spawn many more rats to get some late-game tiebreakers (more cheese collected, more kings formed, etc).

If I had more time, I would have found a better way of setting spawned rat thresholds that doesn't rely on global cheese. It would be smarter to maintain a lower cheese supply at any given time and look instead at cheese collected over time as a metric for determining how many rats we can have in play. However, I didn't have time to optimize this, and used the logic that maintaining larger amounts of cheese on hand has the added benefit of making our kings very resistant to late-game rushes, since we can spawn many rats in defense. It also prevents our own rats from clogging up mazes or small lanes.

I also wish I had scaled the number of rats based on the map size. I changed the minimum number of rats from 12 to 16 near the end of the competition. Around the same time, my bot started performing much worse on smaller combat-oriented maps. I realized after the final tournament that this was because on smaller maps, you trade rats much more quickly with your opponent, and trying to keep 16 rats on hand at any given time means you are spending more cheese per traded rat than your enemy. On larger maps, the need for exploration and collecting cheese from far away mines is much more important, which is why I increased the threshold to begin with; however, I simply hadn't thought of having a variable threshold.

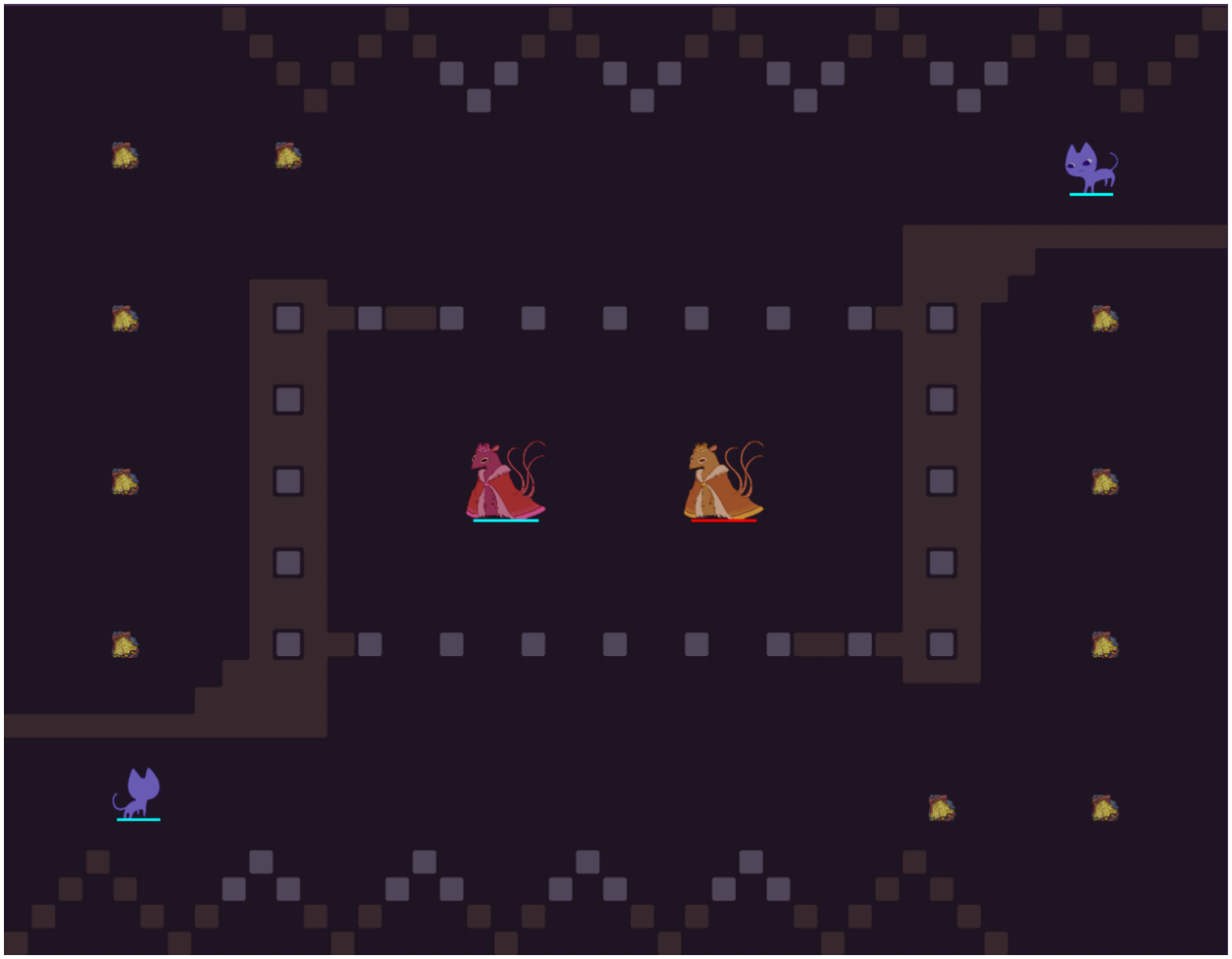


Figure 23. Map “thunderdome” from Sprint 1 tournament. The two rat kings start very close to each other, so a newly spawned rat can very quickly reach the enemy. Hence, there is no reason to spawn too many rats at any given time.

Rat King Formation

When we have enough global cheese (based on a threshold that is proportional to the number of existing kings) and have discovered enough mines, one of our rat kings will write a desired formation location to the shared array. Then, any rat in COLLECT mode will switch to BUILD_RAT_KING mode and pathfind to the formation location.

Baby rats also have the agency to switch to BUILD_RAT_KING mode on their own if they meet the prior conditions and sense many other baby rats nearby. This is very beneficial if, for example, several rats reach a new, desirable mine at the same time; rather than having to return to the king to enact the formation logic, a baby rat can squeak a formation location and let it propagate to the rats around it through “resqueaks”.

When a king is coordinating a formation location, it chooses an already discovered mine; since mines can't be too close to walls, there is guaranteed to be enough space to form a new king. When a baby rat tries to coordinate a new king, it chooses its own location as the formation location, so it also checks that the adjacent tiles are empty.

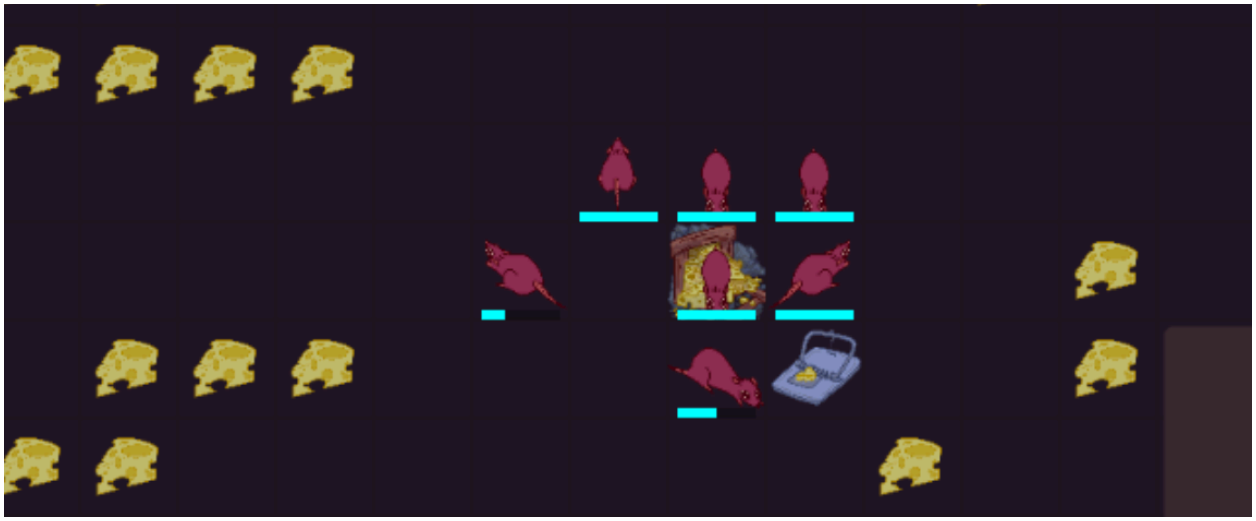


Figure 24. Map "Hike" from Finals. The rat king set a discovered mine location as the formation point, causing baby rats to swarm toward it to form a king.

If the number of alive ally rat kings increases, or we have been trying to form a rat king for many turns, baby rats will clear the stored formation location locally and exit BUILD_RAT_KING mode. Rat kings will also clear the spot in the shared array.

King formation logic is one of the main shortcomings I wish I had addressed before finals. I relied on a static global cheese threshold to initiate king formation. However, there are some situations where optimally you should ignore this (i.e. when discovering a region with lots of cheese, forming a new king would allow the cheese to immediately enter the global supply, without the need to return to an ally king).

In fact, I noticed during the later tournaments that many matches were won simply by who formed a rat king first. There were several maps where the first rat king would be stuck in a bad spawn location and/or there would be a hard-to-reach region with lots of cheese. In either situation, having baby rats coordinate rat king formation even when formation criteria aren't met would have provided a significant advantage.

Figure 25. Map “warhammer” from US Qualifiers. Since the first rat king is trapped in the upper left corner, forming new rat kings is essential for maintaining the cheese economy.

King Defense

Escaping Dirt

If the king is completely surrounded by dirt, it will dig to escape.

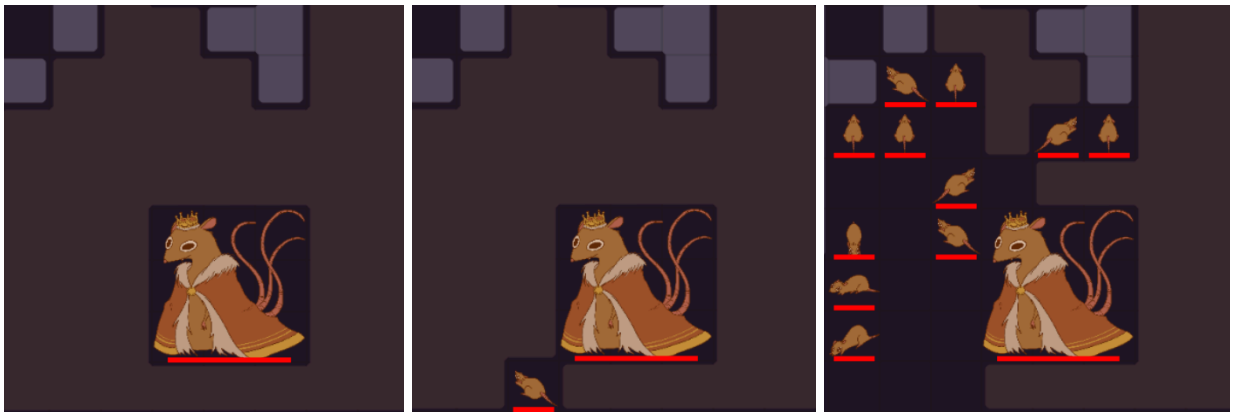


Figure 26. “mushroom” map from US Qualifiers. The rat king starts completely surrounded by dirt. The king randomly digs a dirt tile, freeing up a tile to spawn a baby rat and escape.

Anti-Rush

Rat kings spawn baby rats in the direction of enemies when sensed. They also move away from enemies, bite enemies, place traps, and place dirt against cats.

It isn't always feasible to spawn new baby rats, especially when global cheese is low, so I implemented a king SOS system where any baby rat near an ally king would enter RETURN mode upon hearing the SOS. I initially used rat king squeaks for this SOS, but I was worried about enemy cats targeting my rat king when hearing the squeak, so I switched to using the shared array method as described in the Communication section earlier.

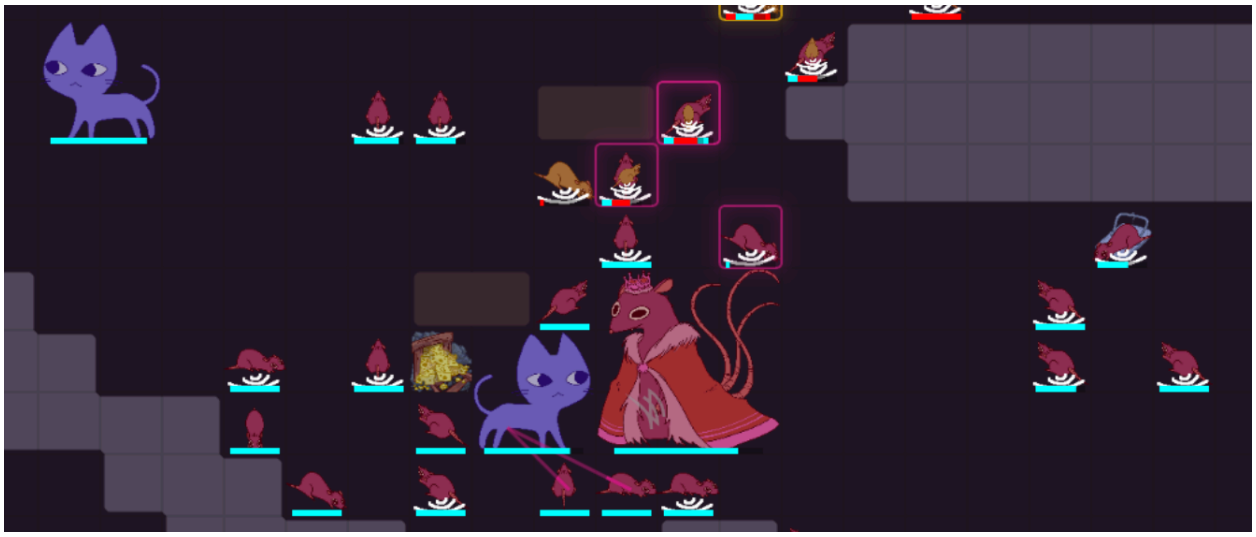


Figure 27. Map "heart" from Finals. A combination of newly spawned and returning baby rats helps defend the plum rat king from both cat and enemy rat threats.

Escaping Cats

Cat behavior was updated near the end of the competition so that cats mostly move between waypoints in a straight path. The main drawback of this change is that rat kings move much slower than cats, so if the rat king is in the cat's path, the cat will quickly catch up to it.

Hence, if the cat is facing the rat king, instead of moving directly away from it, we try to move perpendicular to the cat's path, while still increasing our distance from it. This can help us get out of tricky situations, such as when a cat is pressing the rat king against a wall or map boundary, and the rat king has to move sideways to escape.

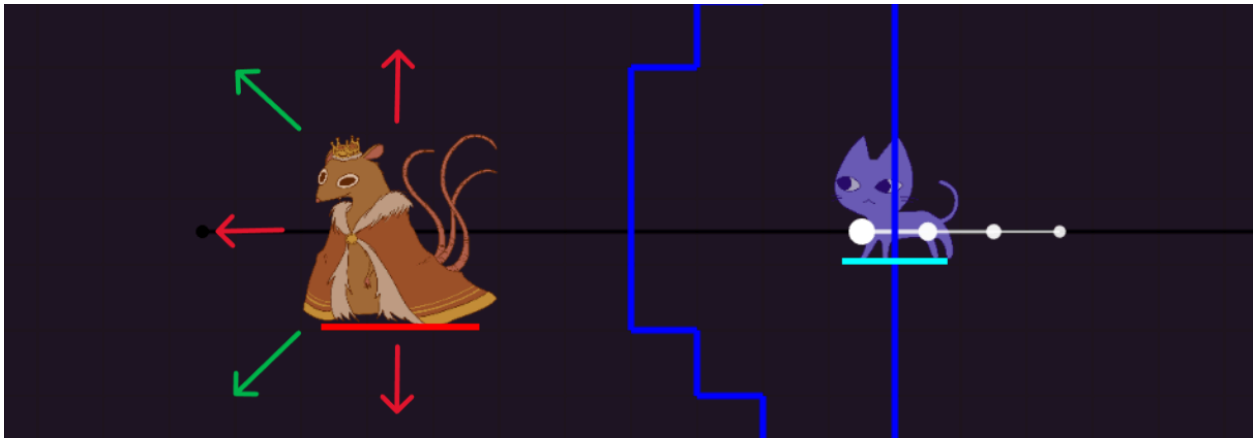


Figure 28. Rather than only moving away from the cat or only moving out of the cat's path (red arrows), the rat king will take both into consideration (green arrows). This gives the king two chances at escape; if the cat switches directions before it reaches the king, or if the king moves out of the way and the cat moves past the king toward the waypoint.

5. Conclusion

As a solo competitor, time was a luxury I couldn't get enough of. The Battlecode kickoff was on January 5th and the Finals submission deadline was on January 29th, so I should have had 25 days to work on my bot. However, I ended up rewriting my bot essentially from scratch after making many mistakes in the first week. I also had to fly from Atlanta to Seattle on January 17th to move into my new apartment and prepare for starting an internship at Stripe, and then fly to San Francisco for a week of in-person onboarding, so most of my free time was taken up with other things.

I plan on competing again next year, though, and I hope I can spend significantly more time on Battlecode.

Final Thoughts

I'd like to end this postmortem with some general ideas I've been bouncing around since Finals on the 'optimal' way to approach the game, after learning from my mistakes this year. If you're a beginner, I definitely think you'll benefit from applying these principles.

Don't pay attention to sprint tournaments or leaderboard rankings. I can understand using these as a motivating factor to work on your bot; however, you should focus on making the improvements that will help you most in the long run, rather than worrying about short term matchups. For example, rush strategies are extremely viable in the first week, when maps are very open and everyone is still learning the game, but will rarely

work by the Final tournament. So instead of wasting time implementing a specific rush strategy that you will quickly abandon, you should focus on the fundamentals like proper economy management, exploration, pathfinding, etc from the start. Building core, reusable systems that you can fine-tune later will make it much easier to test out various strategies, adapt to balance changes, and outsmart your competitors over time.

Be wary of overfitting against any particular bot, team, or map. When you repeatedly test your bot under the same conditions, you might end up exploiting bot-specific or map-specific behaviors rather than actually improving. For example, if you use an older version of your bot as a baseline, and accept any change that increases your win rate, you'll see rapid improvements locally, but you might not perform any better against actual teams who use completely different strategies or on new tournament maps where you have to take a different approach.

Keep it simple. You've likely seen this in other postmortems, but it's important enough to reiterate here. The simplest strategies are often the most effective, especially since your units are operating with very limited information. It is often easiest to start with some core principle; for example, baby rats should stay near a rat king when collecting cheese, so they can easily return without dying or getting lost. Then, you can derive various strategies from that principle:

- Baby rats should visit nearby mines first to avoid ending up very far from the ally rat king.
- Baby rats should scale how much cheese they collect based on how close they are to the ally rat king, so they don't take too long to return.
- Rat kings should be more centrally positioned, to reduce the longest distance a baby rat would have to travel.
- Rat kings should be located near mines, since baby rats will travel between the two.
- Baby rats should pick up and throw each other closer to the rat king to cover the same distance faster.

Not all of these ideas are good; as you implement them, you'll discover problems, like how sending baby rats to nearby mines results in them fighting over a limited amount of cheese, or how throwing your own rats can damage them too. But by following this process of identifying some general principle, thinking through a potential solution, and trying out a specific implementation, you will be able to see how your ideas fare in practice and get a much better sense of the game.

Don't obsess over win rate. Often your bot's performance will get worse when you first start implementing an idea, only to increase much later on when you've ironed out the

specifics. You can speed up this process by watching replays, seeing how your units act, and figuring out what they're doing wrong.

I cannot reiterate this last point enough. The key to improvement is often as simple as realizing your units are frequently doing [stupid thing] when they should be doing [slightly less stupid thing] instead. You can even step through individual robot turns, thinking through where you as a human player would move if you could, and then seeing if your robot makes the same decision or not. The process of coding your bots to act the same way as your (hopefully) far more complex mental model of the game is the most challenging part of Battlecode, and the most fun.

Make new maps to see how your bot adapts to different situations. There's no point testing your complex maze pathfinding improvements on DefaultLarge, or testing exploration strategies on a tiny map. When I was testing, I would often run 20 or 30 maps at a time to get an idea of how my bot had changed, but this was a mistake as it's hard to learn anything from why your bot suddenly lost on some arbitrary map and won on another. Instead, I would recommend making three to five maps to test a specific behavior, such as your combat micro, and watching replays of how your bot performs on these maps as you're iterating on it. Then when you've finished, make a few more maps, or test on the rest of your general maps, and see how your bot fares overall.

That being said, it's almost impossible to avoid bias and overfitting when you're testing the same thing repeatedly; if you only rely on a couple of maps then you'll overfit to those maps, and if you run matches on dozens of maps, then specific, important information and insights are often lost to a simple win rate comparison. But understanding where bias can come from is the best starting point to being able to address it.

Don't try to figure everything out yourself. Ironic from a solo competitor, I know. The Battlecode community is incredibly helpful, and you'll see plenty of great ideas and thoughts in the Discord server. You can also run matches against other teams to see what strategies they're using. You can learn a lot from reverse engineering what a high-ranking team is doing and incorporating similar logic in your own code.

TL;DR: Build a solid foundation early on, come up with your own theories and see if they work, borrow from what other teams are doing, keep things simple, and make sure you test against a variety of bots on a variety of maps. Oh and, have fun in the process!