Battlecode 2025: Chromatic Conflict

Table of contents

Background

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?

Objective

In Battlecode: *Chromatic Conflict*, your goal is to paint the map. The first team to paint more than 70% of paintable squares on the map wins the game. Teams can also win by destroying all of the enemy team's robots and towers.

Tiebreakers

If neither team has painted more than 70% of the **paintable squares** on the map after 2000 rounds, the game will end immediately. The following tiebreakers are applied in order to determine the winning team:

- Area painted on map
- Number of allied towers currently alive
- Total amount of money
- Sum of paint across all robots and towers
- Number of allied robots currently alive
- A uniformly random team will be selected

Good luck!

Map overview

Each Battlecode game will be played on a map. The map is a discrete 2-dimensional rectangular grid, of size ranging between 20×20 and 60×60 inclusive. The bottom-left corner of the map will have coordinates (0, 0); coordinates increase East (right) and North (up). Coordinates on the map are represented as MapLocation objects holding the x and y coordinates of the location.

In order to prevent maps from favoring one player over another, it is guaranteed that the world is symmetric either by rotation or reflection.

Passability and Visibility

Robots can always sense map features and other robots up to sqrt(20) units away. **Units cannot move over towers or other robots.**

Maps may have walls, which robots cannot pass through or paint on. No more than 20% of a map will be walls. Walls cannot be removed or built by competitors. Robots can also not pass through or paint on ruins.

Towers and Ruins

There are three types of towers—money-generating towers, paint-generating towers, and defense towers. Towers also serve as spawn zones for each team and can attack units from the opposing team.

To build a tower, a team must paint a specific shape onto a ruin (see towers section), a 5x5 zone pre-placed on the map. Each game starts with some number of ruins, and the ruins cannot be moved or destroyed. The centers of ruins will be at least 5 units apart, **and there will never be any walls in the 5x5 zone around the center of a ruin.** Each team starts with 2 towers, 1 paint and 1 money tower. Ruins cannot be seen by robots or towers unless they are within the vision radius.

Special Resource Pattern (5 by 5 shape)

This game has a special resource pattern (SRP). For each instance of this pattern that is painted on the map, all allied mining towers will mine 3 more resources per turn. Robots can mark the map with this pattern using the markResourcePattern() function. Robots must call the completeResourcePattern() function once the pattern is painted on the map for it to start boosting resource production.

Resources

Paint

Paint is the primary resource of the game. It is used to mark territory and count scores. Each robot and tower hold paint individually, up to a certain capacity determined by their type. Robots face increased cooldowns or an inability to act at low paint, but they can refill their paint from moppers or by withdrawing from paint-generating towers. Each robot type (described below) has a distinct attack that will either consume paint to place it on a square or remove the opponent's paint from a tile. Robots can choose between using a team's primary and secondary color to paint a tile. These two colors are treated identically and are only distinguished when checking for completion of a paint pattern.

Chips

Chips can be used to build units & towers. Chips are generated each turn by money generating towers, and can be used by any allied robot once generated. Each team starts the game with 2500 chips.

Units

The Battlecode world contains many kinds of robots. All robots can perform actions such as moving, sensing, and communicating with each other. In each battle, your robots will face one opposing enemy team.

The game is turn-based and divided into **rounds**. In each round, every robot gets a **turn** in which it gets a chance to run code and take actions. Code that a robot runs costs **bytecodes**, a measure of computational resources. A robot only has a predetermined amount of bytecodes available per turn, after which the robot's turn is immediately ended and computations are resumed on its next turn. If your robot has finished its turn, it should call Clock.yield() to wait for the next turn to begin.

All robots have a certain amount of HP (also known as hitpoints, health, life, or such). When a robot's HP reaches zero, the robot is immediately removed from the game.

Robots are assigned unique random IDs no smaller than 10,000.

Robots interact with only their nearby surroundings through sensing, moving, and special abilities. Each robot runs an independent copy of your code. Robots will be unable to share static variables (they will each have their own copy), because they are run in separate JVMs.

Two or more robots may not be on the same square. When their movement cooldown goes below 10, robots can move onto any of the 8 neighboring squares.

All robots also have a stash of paint, allowing them to store paint up until their capacity. Robots face increased movement and action cooldowns the lower their stash of paint is. Robots with a stash of paint that is X% full, where X is less than 50, face (100-2X)% increased cooldowns. Once a robot’s stash of paint is entirely depleted, it is unable to move or perform actions other than self destructing until it receives paint. Robots can stay in this state for 10 turns, at the end of which they will self-destruct if not given any paint. Robots spawn with their paint stash 100% full and can have it refilled at a tower or by a mopper (described below). Robots lose 1 paint when ending their turn on neutral territory, 2 paint when ending their turn on enemy territory, and no paint to end their turn on allied territory. Robots also face an additional paint penalty of 2 times the number of adjacent allied bots (bots in the 8 squares around the robot) for each turn they spend on enemy territory.

Robots can be spawned within an action radius of sqrt(4) by allied towers, costing some amount of paint and chips according to their type.

Soldier

Soldiers have 250 health and have a maximum paint stash of 200. They cost 200 paint and 250 chips to produce. They can attack a tile, painting it if it is empty or has allied paint on it as well as dealing 50 damage to enemy towers (not robots). This attack costs 5 paint, has an action cooldown of 10, and has a radius of 3.

Splashers

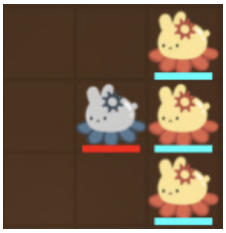
Splashers have 150 health and a maximum paint stash of 300, costing 300 paint and 400 chips to produce. Their attack costs 50 paint and has an action cooldown of 50, but it affects a circular area within a radius of 2 from its center point. Within a radius of sqrt(2) of the center location, it can also paint over enemy paint. It targets a center location up to 2 units away. This attack deals 100 damage to enemy towers (not robots) and paints all empty/allied tiles.

Mopper

Moppers have 50 health and a maximum paint stash of 100, costing 100 paint and 300 chips to produce. They can choose to mop a tile, removing enemy paint on the tile. If an enemy robot is on the tile, they lose 10 paint which is converted to 5 paint for the mopper. This action incurs a 30 action cooldown. Moppers can also transfer a specified amount of paint from their own paint stash to allied robots or towers at a 10 action cooldown. These actions have a radius of sqrt(2) units. However, moppers face double paint penalties from moving and staying on enemy terrain. Moppers can also swing their mop in

one of the 4 cardinal directions, removing 5 paint each from adjacent 3 enemy robots in a line (see diagram). Mop swinging has an action cooldown of 40.

Examples of mop sweeps



Silver swings east (right). All three gold bots lose paint.

Silver swings north (up). All three gold bots lose paint.

Silver swings south (down). The bottom gold bot loses paint.

Silver swings west (left). No gold bots lose paint.

Most of the robot statistics are summarized in the table below.

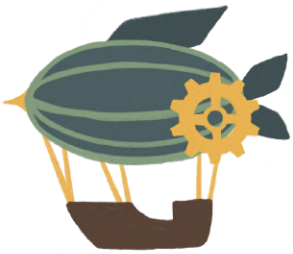
Name	Health	Paint Capacity	Paint Cost	Money Cost	Attack Cost	Attack Radius	Attack Damage	Attack Cooldown
Soldier	250	200	200	250	5	3	50 health to towers	10
Splasher	150	300	300	400	50	2	100 health to towers	50
Mopper	50	100	100	300	0	$\sqrt{2}$	-10 paint to robots	30

Towers

There are three types of towers: money-generating, paint-generating, and defense. By default, every team will start with a money-generating tower and a paint-generating tower.

Money-generating tower

Paint-generating tower



Defense tower

All towers are built with the same procedure. Towers can only be built on ruins, after a certain 5x5 pattern of paint is placed around the ruin. Call markTowerPattern() to automatically place markers on the 5x5 area surrounding a ruin, designating where to place paint of the primary color. This method costs 25 paint.

-  Effect of calling markTowerPattern()

Once the shape is painted, a robot can call completeTowerPattern() to spawn the designated tower at the ruin location. This tower will remain alive until its health reaches 0, even if the paint pattern around the ruin is later removed. Note that a tower pattern can be completed without needing to first call markTowerPattern(), as long as the pattern of primary and secondary paint is matched.

- 

Generated towers will occupy a 1x1 block in the center of the spawn shape. Different shapes correspond to different types of towers with different ranges, attack rates, mining rates, and health (as detailed in the table below). Towers will always be built at level 1 and can be upgraded to higher levels using the upgradeTower() method within a radius of sqrt(2) units. Neither team may have more than 25 towers at any given time. Each tower can store 1000 paint. Each defense tower that is alive increases the single-block attack strength of all allied towers by +5/+7/+9 based on its level respectively. This buff does not affect the AoE attack of allied towers.

Name	Build/ Upgrade cost	Attack Range	Single-block Attack Strength	AoE Attack Strength	Base Mining Rate	Health
Money Lv1	1000 chips	3	20	10	20 chips/turn	1000
Lv2	2500 chips	3	20	10	30 chips/turn	1500
Lv3	5000 chips	3	20	10	40 chips/turn	2000
Paint, Lv1	1000 chips	3	20	10	5 paint/turn	1000
Lv2	2500 chips	3	20	10	10 paint/turn	1500
Lv3	5000 chips	3	20	10	15 paint/turn	2000
Defense, Lv1	1000 chips	4	40	20	0	2000
Lv2	2500 chips	4	50	25	0	2500
Lv3	5000 chips	4	60	35	0	3000

Attack

To defend themselves, towers are able to attack units. Towers have two types of attacks—a single-block attack that attacks an enemy unit on one block and a ranged area of effect (AoE) attack that does damage to all enemy units within its range. Every turn, a tower can perform one single-block attack and one AoE attack.

Mining

Every turn, money and paint towers will passively mine resources. Paint is stored in the tower and can be removed by allied robots. Money is available to all allied units.

Every tower starts with 500 paint to spawn different types of units. It can spawn a bot within 2 blocks of the tower. If a tower runs out of paint it will not be able to spawn units until it is refilled or regenerates paint on its own. A tower can be refilled by the paint collected by moppers.

Communication

Robots can only see their immediate surroundings and are independently controlled by copies of your code, making coordination very challenging. You will be unable to share any variables between them; note that even static variables will not be shared, as each robot will receive its own copy.

Communication is done through two methods:

1. Messages, which can contain any information, but are restricted to 4 bytes.
2. Markers, which will state which color a space should be repainted to, and is visible to all allied robots. Markers do not have any other effect on the game.

Messages

The robots and towers who are connected by paint can talk, but the robot who is not on paint cannot talk to the tower.

A robot is “in range” of a tower if it is within sqrt(20) of the tower and connected to the tower by paint (i.e., it can move to the tower by only walking on blocks with its own paint). Any robot in range of a tower can send a message to the tower and vice versa.

A robot or tower will store all messages it has received in the last 5 turns in its buffer, after which, they will be deleted. Robots can send 1 message per turn, towers can send 20 messages per turn. A message

consists of one 32 bit integer, as well as the round number that the message was sent in and the id of the robot that sent the message.

Markers

Markers are information that can be placed/erased on the map by robots. They can either indicate “secondary color” or “primary color.” They are only visible to allied robots and remain there until either erased or another marker is placed on the spot by an ally. Markers can be placed manually on a square within sqrt(2) units of the square the robot currently occupies, or with the markTowerPattern()/markResourcePattern() function, which marks the 5x5 area around a square within sqrt(2) of the robot with the appropriate pattern. Markers can only be placed on tiles that are paintable.

Actions and Cooldowns

Robots perform actions to interact with the game world, and some cannot be performed multiple times in a single turn or in a short period of time. These actions are:

- *Attacking/Painting.* Attacking and painting are performed with the same method. Attacking can only be done if the robot's action cooldown is less than 10 (can check with the isActionReady() method). Attacking can be performed by calling the attack() method. This targets a particular square, dealing damage or painting according to the robot’s type as described in the Robots section. This increases the robot's action cooldown as determined by the robot type.

- *Moving.* Moving can only be done if the robot's movement cooldown is less than 10 (can check with the isMovementReady() method) and if the ending square is unoccupied and passible. Moving can be performed by calling the move() method. This moves the robot in the specified direction. This increases the robot's movement cooldown by 10. If you wish to check whether a move is valid, you can use the canMove() method.

- *Marking.* All robots can mark their current tile or remove an allied mark from their current tile at the cost of 1 paint, incurring no action cooldown. This can be done by calling the mark() or removeMark() methods. These marks do not affect territory ownership (i.e. marking a tile does not set it as allied territory) but can be sensed by ally robots as part of the senseNearbyMapInfos() method. The markTowerPattern()/markResourcePattern() function uses 25 paint and marks a 5 by 5 space with the tower pattern. Marks can only be placed on squares that are paintable (i.e. has no wall or ruin on it).

- *Transferring.* Moppers can transfer paint if the robot’s action cooldown is less than 10 (can check with the isActionReady() method), and can be performed by calling the transferPaint()method.

Moppers must be within sqrt(2) units of the location to transfer resources. This transfers the specified amount of paint to the target and removes it from the mopper's stash. If the given quantity is negative, the robot instead removes the specified paint. This increases the robot's action cooldown as specified in the robot details.

- *Withdrawing*. All robots can withdraw paint from allied towers to replenish their paint stash. This requires an action cooldown of less than 10 and will increase their action cooldown by 10. It has an action radius of sqrt(2) units and can be called with the transferPaint() method, using a negative value to denote the amount to withdraw.

- *Spawning.* Towers can spawn robots, but only when their action cooldown is less than 10 (can check with the isActionReady() method), and can be performed by calling the buildRobot() method. This deducts the cost of the robot from the tower's stockpile, then creates one robot of the specified type in the specified square. It also increments the tower's action cooldown by 10.

- *Communicating.* Robots can use sendMessage() to send a message to an allied tower within their range, while towers can use sendMessage() to send a message to an allied robot within range. A bot can use readMessages() to read any new messages sent within the past 5 rounds from their message buffer. These methods do not increase action cooldown. Two units must be on squares connected by ally paint to communicate with each other.

- *Disintegrating.* Any robot can call the disintegrate() method. This immediately destroys the robot that calls it.

After every turn, the movement and action cooldowns of all robots are decremented by 10.

Bytecode limits

Robots are also very limited in the amount of computation they are allowed to perform per turn. Bytecodes are a convenient measure of computation in languages like Java, where one Java bytecode corresponds roughly to one basic operation such as "subtract" or "get field", and a single line of code generally contains several bytecodes (for details see here). Because bytecodes are a feature of the compiled code itself, the same program will always compile to the same bytecodes and thus take the same amount of computation on the same inputs. This is great, because it allows us to avoid using time as a measure of computation, which leads to problems such as nondeterminism. With bytecode cutoffs, re-running the same match between the same bots produces exactly the same results - a feature you will find very useful for debugging.

Every round each robot sequentially takes its turn. If a robot attempts to exceed its bytecode limit (usually unexpectedly, if you have too big of a loop or something), its computation will be paused and then resumed at exactly that point next turn. The code will resume running just fine, but this can cause

problems if, for example, you check if a tile is empty, then the robot is cut off and the others take their turns, and then you attempt to move into a now-occupied tile. Instead, use the Clock.yield() function to end a robot's turn. This will pause computation where you choose, and resume on the next line next turn.

The bytecode limit for all robots is **15000**, and the bytecode limit for all towers is **20000.**

Some standard functions such as the math library and sensing functions have fixed bytecode costs, available here. More details on this at the end of the spec.

Appendix: Other resources and utilities

Sample player

examplefuncsplayer, a simple player which performs various game actions, is included with battlecode. It includes helpful comments and is a template you can use to see what RobotPlayer files should look like.

If you are interested, you may find the full game engine implementation here. This is not at all required, but may be helpful if you are curious about the engine's implementation specifics.

Debugging

Debugging is extremely important. See the debugging tips to learn about our useful debug tools.

Monitoring

The Clock class provides a way to identify the current round (rc.getRoundNum()), and how many bytecodes have been executed during the current round (Clock.getBytecodeNum()).

GameActionExceptions

GameActionExceptions are thrown when something cannot be done. It is often the result of illegal actions such as moving onto another robot, or an unexpected round change in your code. Thus, you must write your player defensively and handle GameActionExceptions judiciously. You should also be prepared for any ability to fail and make sure that this has as little effect as possible on the control flow of your program.

Throwing any Exceptions cause a bytecode penalty of 500 bytecodes. Unhandled exceptions may paralyze your robot.

Complete documentation

Every function you could possibly use to interact with the game can be found in our javadocs.

Appendix: Other restrictions

Java language usage

Players may use classes from any of the packages listed in AllowedPackages.txt, except for classes listed in DisallowedPackages.txt. These files can be found here.

Furthermore, the following restrictions apply:

Object.wait, Object.notify, Object.notifyAll, Class.forName, and String.intern are not allowed. java.lang.System only supports out, arraycopy, and getProperty. Furthermore, getProperty can only be used to get properties with names beginning with "bc.testing.". java.io.PrintStream may not be used to open files.

Note that violating any of the above restrictions will cause the robots to explode when run, even if the source files compile without problems.

Memory usage

Robots must keep their memory usage reasonable. If a robot uses more than 8 Mb of heap space during a tournament or scrimmage match, the robot may explode.

More information on bytecode costs

Classes in java.util, java.math, and scala and their subpackages are bytecode counted as if they were your own code. The following functions in java.lang are also bytecode counted as if they were your own code.

`Math.random StrictMath.random String.matches String.replaceAll String.replaceFirst String.split`

The function System.arraycopy costs one bytecode for each element copied. All other functions have a fixed bytecode cost. These costs are listed in the MethodCosts.txt file. Methods not listed are free. The bytecode costs of battlecode.common functions are also listed in the javadoc.

Basic operations like integer comparison and array indexing cost small numbers of bytecodes each.

Bytecodes relating to the creation of arrays (specifically NEWARRAY, ANEWARRAY, and MULTIANEWARRAY; see here for reference) have an effective cost greater than a single bytecode. This is because these instructions, although they are represented as a single bytecode, can be vastly more expensive than other instructions in terms of computational cost. To remedy this, these instructions have

a bytecode cost equal to the total length of the instantiated array. Note that this should have minimal impact on the typical team, and is only intended to prevent teams from repeatedly instantiating excessively large arrays.

Appendix: Lingering questions and clarifications

If something is unclear, direct your questions to our Discord where other people may have the same question. We'll update this spec as the competition progresses.

Appendix: Changelog

- V1.4.0
    - Java/Python Engine
        - Initial chips amount increased to 2500
- V1.3.3
    - Java/Python Engine
        - canMark now checks if a square is paintable
        - Fix 2x paint penalty on enemy territory for moppers not being applied
    - Client
        - Add support for mopper attacks during turn stepping
        - Fix tooltip not rerendering during turn step
        - Add config/hotkey for auto-focus robot during turn step
        - Fix crash on older macos versions
- V1.3.2
    - Java/Python Engine
        - Fix timeline markers overflowing between matches
    - Client
        - Fix flipped team colors on the graphs (for real)
        - Attempted fix for client crash when navigating during live replays
- V1.3.1
    - Java/Python Engine
        - Fix bug that allowed building pre-upgraded towers
    - Client
        - Fix flipped team colors on the graphs
        - Add color config v1
- V1.3.0
    - Java/Python Engine
        - Tower costs 100/250/500 chips to build and upgrade -> 1000/2500/5000 chips

        - Defense tower attack range sqrt(20) -> 4, Splasher attack range sqrt(8) -> 2
        - Defense tower buff +10/15/20 -> +5/7/9. AoE effectiveness (from buff) 50% -> 0%
        - Soldier attack damage 20 -> 50, Splasher attack damage 50 -> 100
        - Defense tower damage 60/65/70 -> 40/50/60 (single target), 30/35/40 -> 20/25/30 (aoe)
        - Defense tower health 2500/3000/3500 -> 2000/2500/3000
        - Money towers 10/15/20 chips/turn -> 20/30/40 chips/turn
        - Add PaintType.isEnemy()
        - Add UnitType.getBaseType()
        - Add rc.canPaint()
    - Client
        - Fix paint bar overlay when selecting robot
        - Allow UPS to go up to 128
- V1.2.0
    - Java/Python Engine
        - Add rc.getChips() as an alias for rc.getMoney()
        - Initial paint is properly reversed on the map when towers are (Java fix)
        - canMarkResourcePattern() now checks for walls and ruins in a 5x5 area
        - Fix soldier and splasher attack ranges to 3 and sqrt(8) respectively
        - completeTowerPattern now costs 100 chips
    - Client
        - Ruins under starting towers now properly appear in client
        - Added config to show paint bar under robots
- V1.1.0
    - Java/Python Engine
        - Various spec fixes & add tables
        - Clarify transfer docstrings
        - Add disintegrate method
        - Add rc.getNumberTowers() method
        - Starting towers have ruins under them + map guarantee applies to these ruins
            - Builtin maps have been updated to reflect these guarantees
        - Towers can use both attacks per turn now (Java bug fix)
        - Fix mark() docstring (Java)
        - Indicator dot/lines outside of map now throw exceptions
        - canSendMessage no longer requires message content
        - canAttack for soldiers now returns false if the tile has a wall. canAttack for moppers now returns false if the tile has a wall or ruin.
    - Python Engine
        - Add tower limit

        - Add get_num_towers()
    - Client
        - Fixed some issue with text scaling
        - Various map editor guarantee fixes
        - Fixed profiler not working (you will need to update your build.gradle from the scaffold repo)
- V1.0.0
    - Initial release