**Battlecode: Crusade Official Game Specs**

Updated 1/7/19 7:00PM EST

The planet of Mars is a house divided. Only ten short years after the great war for the red planet, two opposing religious dogmas have emerged from the chaos. The Religious Exploratory Doctrinists (RED) believe that the only route to peace in the galaxy is by spreading the robot way of peace, while the Believers of Lasting Unity Everywhere (BLUE) claim that only by non-aggression can robotkind remain. The only possible resolution? Total war.

**Game Format**

Battlecode: Crusade is a turn based game, where robots on a tiled grid are each controlled by individual computer programs. Robots include Castles, Churches, Pilgrims, Crusaders, Prophets, and Preachers. The objective of the game is to destroy the enemy team Castles. If by 1000 rounds both blue and red Castles remain, the winner is determined by the team with more castles, followed by the team with more unit value, followed by a coin flip.

**Map and Resources Overview**

Game maps are procedurally generated, and are square 2d grids ranging between 32x32 and 64x64 tiles. Every map is either horizontally or vertically symmetric, and the top left corner has the coordinates (0,0). Each tile in the map is either passable or impassable rocky terrain, and each team starts with 1-3 Castles on the map, 100 Karbonite, and 500 Fuel.

Passable tiles can have resource points on them which when mined by Pilgrims provide either Karbonite, which is used to construct units, or Fuel, which is used to run them. Once mined, these resources can be transferred between units and deposited for global usage at Castles or Churches. Before being deposited at a Castle or Church, resources are unrefined, and cannot be utilized. Almost any action in Battlecode Crusade consumes either Karbonite or Fuel, all from the global refined stores. Note that rather than being distributed evenly, Karbonite and Fuel depots are usually found in small discrete clumps on the map. In addition to the resources teams start with and mine, at every round each team receives 25 fuel.

Robots have knowledge of the full map at the beginning of the game (including resource depots), and can only see robots within their vision radius.

**Units Overview**

Unlike last year’s Battlecode game, each unit is controlled by its own process. Each unit is initialized with a 100ms chess clock, and receives 20ms of additional computation each round. Each turn is additionally capped at 200ms, after which code will be stopped. If a robot exceeds its chess clock, it cannot move until it has > 0 time in its clock.

When a unit is spawned, it is assigned a unique 32 bit integer ID, and always occupies a single tile. When the health of a unit is reduced to 0, the unit is immediately removed from the game.

There are two types of units: robots and structures. Robots are mobile units that fight, move, build factories, carry resources, or mine fuel and karbonite from the map. There are two types of structures: Castles and Churches. Castles are like Churches that cannot be created and carry special abilities. Churches produce robots, and provide a depot for Pilgrims to deposit resources into the global economy.

**Castles**

Each team starts with 1-3 castles on the map, each with initial health 100 and vision radius 100. Castles have all the abilities of Churches, but cannot be built, and have greater health. Castles also have unique communication abilities; not only can all units send messages to Castles for free (discussed in the Communication section), but Castles can also trade Karbonite and Fuel with opposing team castles.

Each turn, a castle can offer a Barter to a castle of the opposing team. Barters are offers to trade X Karbonite for Y Fuel (or vice versa). Players can use this functionality to collaborate with the opposing team for mutual benefit.

When all of a team’s castles are destroyed, the team is considered defeated.

**Churches**

Churches are structures with the ability to produce robots for their Karbonite and Fuel cost. In any given turn a church or castle can spawn a robot in any adjacent square (where adjacent is defined to include diagonals), with that robot added to the end of the turn queue. Robots adjacent to churches and castles in their turn can deposit Fuel and Karbonite, adding those resources to the team’s global stores.

Churches can be constructed by Pilgrims for 50 Karbonite and 200 Fuel, and have an initial starting health of 50 and a vision radius of 100.

**Robots**

There are four classes of robots: Pilgrims, Crusaders, Prophets, and Preachers. Pilgrims are scouting, mining, and building robots, while the other robots are only capable of combat and resource transportation. Below is a summary of the robot types, with more description following.

|  | **Pilgrim** | **Crusader** | **Prophet** | **Preacher** |
| --- | --- | --- | --- | --- |
| **Construction Karbonite** | 10 | 20 | 25 | 30 |
| **Construction Fuel** | 50 | 50 | 50 | 50 |
| **Karbonite Carrying Capacity** | 20 | 20 | 20 | 20 |
| **Fuel Carrying Capacity** | 100 | 100 | 100 | 100 |
| **Movement Speed (r^2)** | 4 | 9 | 4 | 4 |
| **Movement Fuel Cost (per r^2)** | 1 | 1 | 2 | 3 |
| **Starting Health** | 10 | 40 | 20 | 60 |
| **Vision Radius (r^2)** | 100 | 36 | 64 | 16 |
| **Attack Damage** | N/A | 10HP | 10HP | 20HP for 3 r^2 |
| **Attack Range (r^2)** | N/A | 1-16 | 16-64 | 1-16 |
| **Attack Fuel Cost** | N/A | 10 | 25 | 15 |

Pilgrims are non-combat robots that can mine unrefined Karbonite or Fuel and deliver them to Castles and Churches. For each turn a Pilgrim mines a Karbonite depot, they receive 2 unrefined Karbonite. Similarly, for each turn a Pilgrim mines a Fuel depot they receive 10 unrefined Fuel. Pilgrims can also construct Churches.

Crusaders are capable of shorter-range combat, Prophets are longer range, and Preachers deal AOE damage.

Robots can move to or attack any square within their speed or attack radius, even if that terrain is technically unreachable using a smaller step size. In each turn, a unit can only perform one physical action, including moving, attacking, depositing/giving, mining, trading, and building.

**Reclaim**

When units are destroyed, the robot that destroyed them receives half of the Karbonite required to build the destroyed unit, in addition to any resources they may have been carrying, all divided by the r^2 between the attacker and the target. So, if a Pilgrim were destroyed by a Crusader with dx,dy=(1,1) and was carrying 10 Fuel and 3 Karbonite, the attacker would now have an additional 5 Fuel and 4 Karbonite.

**Communication**

Each unit on the board has its own process, and is sandboxed from other units. To facilitate communication and global planning, each unit has two possible methods of communication.

Radio is the primary method of communication usable by unit. In any given turn, a unit can broadcast a 16 bit message to all units within squared radius X^2, consuming X^2 Fuel. For example, a unit with id 1984 that wanted to broadcast a message with a squared radius of 10 squares would need to expend 10 Fuel. On the next round, all units within that radius will see that the a unit with ID 1984 broadcasted the given message. Units can radio broadcast simultaneously with all other actions. Note that robots can see the unit ID that produced a broadcast, but not which team the unit belongs to.

Units also have a direct channel to communicate an 8 bit value to all their team’s Castles for free from any distance. This can also be combined with any other action, including general radio communications.

**Turn Queue**

Battlecode Crusade games consist of up to 1000 rounds, and each round consists of a turn for every unit on the board at that time. This is acheived by cycling each round through a queue that consists of all units on the map. This queue is initialized with each team’s Castles in alternating Red, Blue order. Then, whenever a unit produces a new unit, that unit is added to the end of the turn queue as soon as the constructor unit’s turn ends. To rephrase, units built in a round will get a turn in the same round. A round consists of a full pass through the turn queue.

**Installation and CLI usage**

Updated 1/7/19 7:00PM EST

This year, Battlecode will be run through the Node Package Manager (npm). Installation for npm varies from operating system to operating system, but generally achieved through the [Node Website](https://nodejs.org/en/). If you are on a Mac, download Homebrew and install from there using brew install node npm.

1. Install npm.
2. npm install -g bc19.
3. Run or compile your code using bc19run or bc19compile. Note that the bot code needs to be in its own directory. Example (using the [examplefuncsplayer](https://github.com/npfoss/examplefuncsplayer) ): bc19run -b bots/exampy -r bots/example\_js --chi 1000.
4. Upload compiled code using bc19upload. Make sure you've defined environment variables BC\_USERNAME and BC\_PASSWORD, which should be the credentials you use to access this site.

You must have internet access to compile Python and Java code. Additionally, be sure to frequently update by running npm install -g bc19. If you are not running the most recent distribution, replays will not render correctly.

#### Java Bot Reference

Updated 1/7/19 7:00PM EST

Below is a bare minimum bot example in Java:

package bc19;

public class MyRobot extends BCAbstractRobot {

public Action turn() {

return move(1,0);

}

}

The main container of your bot code is the MyRobot class, which must extend BCAbstractRobot. BCAbstractRobot contains all sorts of useful methods that will make developing your bot easier.

When your bot is spawned, a MyRobot object is created in its own global scope. For every turn, the turn() method of your class is called. This is where the heart of your robot code lives. If you want the robot to perform an action, the turn() method should return it.

Note that the same MyRobot class is used for all units. Some API methods will only be available for some units, and will throw an error if called by unallowed units.

You cannot change the name of the MyRobot class.

Java is compiled into Javascript before running games. This introduces some unexpected bugs. Known bugs are listed below.

###### State Information

* Robot me: The robot object (see below) for this robot.
* boolean[][] map: The full map. Boolean grid where true indicates passable and false indicates impassable.
* boolean[][] karboniteMap: The Karbonite map. Boolean grid where true indicates that Karbonite is present and false indicates that it is not.
* boolean[][] fuelMap: The Fuel map. Boolean grid where true indicates that Fuel is present and false indicates that it is not.
* int karbonite: The global amount of Karbonite that the team possesses.
* int fuel: The global amount of Fuel that the team possesses.
* int[][] lastOffer: A 2 by 2 grid containing the last trade offers by both teams. lastOffer[0] is the last offer made by RED and contains a list of two integers, where the first one is the amount of Karbonite and the second one is the amount of Fuel. Similarly, lastOffer[1] is the last offer made by BLUE. For both offers, a positive amount signifies that the resource goes from RED to BLUE. Available for Castles (always null for other units).

###### The Robot Object

In the following list, assume that r is a robot object (e.g., r = me). Note that some properties are only available under certain circumstances.

* int r.id: The id of the robot, which is an integer between 1 and 4096. Always available.
* int r.unit: The robot's unit type, where 0 stands for Castle, 1 stands for Church, 2 stands for Pilgrim, 3 stands for Crusader, 4 stands for Prophet and 5 stands for Preacher. Available if visible.
* int r.health: The health of the robot. Only available for r = me.
* int r.team: The team of the robot, where 0 stands for RED and 1 stands for BLUE. Available if visible.
* int r.x: The x position of the robot. Available if visible.
* int r.y: The y position of the robot. Available if visible.
* int r.fuel: The amount of Fuel that the robot carries. Only available for r = me.
* int r.karbonite: The amount of Karbonite that the robot carries. Only available for r = me.
* int r.turn: The turn count of the robot (initialiazed to 0, and incremented just before turn()). Always available.
* int r.signal: The signal of the robot. Available if radioable.
* int r.signalRadius: The signal radius of the robot. Available if radioable.
* int r.castleTalk: The castle talk message sent by the robot. Available if me is a Castle.

Visible means that r is within me's vision radius (particularly, me is always visible to itself). Radioable means that me is within r's signal radius.

###### Actions

The following is a list of methods that can be returned in turn(), to perform an action. Note that the action will only be performed if it is returned; thus, only one of these actions can be performed per turn.

* MoveAction move(int dx, int dy): Move dx steps in the x direction, and dy steps in the y direction. Uses Fuel (depending on unit and distance). Available for Pilgrims, Crusaders, Prophets, Preachers.
* MineAction mine(): Mine 2 Karbonite or 10 Fuel, if on a corresponding resource tile. Uses 1 Fuel. Available for Pilgrims.
* GiveAction give(int dx, int dy, int karbonite, int fuel): Give karbonite Karbonite and fuel Fuel to the robot in the tile that is dx steps in the x direction and dy steps in the y direction from me. A robot can only give to another robot that is in one of its 8 adjacent tiles, and cannot give more than it has. Uses 0 Fuel. Available for all robots. If a unit tries to give a robot more than its capacity, the excess is loss to the void.
* AttackAction attack(int dx, int dy): Attack the robot in the tile that is dx steps in the x direction and dy steps in the y direction from me. A robot can only attack another robot that is within its attack radius (depending on unit). Uses Fuel (depending on unit). Available for Crusaders, Prophets, Preachers.
* BuildAction buildUnit(int unit, int dx, int dy): Build a unit of the type unit (see r.unit) in the tile that is dx steps in the x direction and dy steps in the y direction from me. Can only build in adjacent, empty and passable tiles. Uses Fuel and Karbonite (depending on the constructed unit). Available for Pilgrims, Castles, Churches. Pilgrims can only build Churches, and Castles and Churches can only build Pilgrims, Crusaders, Prophets and Preachers.
* TradeAction proposeTrade(int karbonite, int fuel): Propose a trade with the other team. karbonite and fuel need to be integers. For example, for RED to make the offer "I give you 10 Karbonite if you give me 10 Fuel", the parameters would be karbonite = 10 and fuel = -10 (for BLUE, the signs are reversed). If the proposed trade is the same as the other team's last\_offer, a trade is performed, after which the last\_offer of both teams will be nullified. Available for Castles.

###### Communication

* void signal(int value, int sq\_radius): Broadcast value to all robots within the squared radius sq\_radius. Uses sq\_radius Fuel. value should be an integer between 0 and 2^16-1 (inclusive). Can be called multiple times in one turn(); however, only the most recent signal will be used, while each signal will cost Fuel.
* void castleTalk(int value): Broadcast value to all Castles of the same team. Does not use Fuel. value should be an integer between 0 and 2^8-1 (inclusive). Can be called multiple times in one turn(); however, only the most recent castle talk will be used.

###### Helper Methods

* void log(String message): Print a message to the command line. You cannot use ordinary System.out.print in Battlecode for security reasons.
* Robot[] getVisibleRobots(): Returns a list containing all robots within me's vision radius and all robots whose radio broadcasts can be heard (accessed via other\_r.signal). For castles, robots of the same team not within the vision radius will also be included, to be able to read the castle\_talk property.
* int[][] getVisibleRobotMap(): Returns a 2d grid of integers the size of map. All tiles outside me's vision radius will contain -1. All tiles within the vision will be 0 if empty, and will be a robot id if it contains a robot.
* Robot getRobot(id): Returns a robot object with the given integer id. Returns null if such a robot is not in your vision (for Castles, it also returns a robot object for all robots on me's team that are not in the robot's vision, to access castle\_talk).
* boolean isVisible(Robot robot): Returns true if the given robot object is visible.
* boolean isRadioing(Robot robot): Returns true if the given robot object is currently sending radio (signal).
* boolean[][] getPassableMap(): Returns map.
* boolean[][] getKarboniteMap(): Returns karboniteMap.
* boolean[][] getFuelMap(): Returns fuelMap.



R2