Operation: Code Clash Programming Competition 2014 Presents

1 The Challenge

Paintball Panic is a virtual paintball game between a red team and a blue team. Each player controls a team of four children whose goal is to score more points than the opponent. Points can be scored by hitting the other team's children with paintballs, playing defense and by claiming ownership over areas of the field. The details and rules of the game are in the game manual, Section 5 of this document.

1.1 Organization of this Manual

This manual covers everything you should need to know in order to participate in Operation: Code Clash. The topics covered are as follows:

Competition Format	Section 2
Player Execution Environment	Section 3
The Game Package	Section 4
Game Rules	Section 5
Tournament Rules	Section 6

2 Competition Format

The competition will take place over the course of 6 weeks. The schedule is as follows:

September 27 (Noon EDT)	Game is released
October 11 (Noon EDT)	Qualification Round begins
October 25 (Noon EDT)	Competition Registration ends
November 9 (5pm EST)	Qualification Round ends
November 10	Playoffs Begin
November 13	Competition Ends; Champion Announced

During the first two weeks after the game is released, teams spend time creating a strategy and coming up with initial solutions. Teams will not be able to submit code to the OCC website during this time. However, teams can play against the provided sample players, any other players the team creates and can even practice against other teams' players.

When the Qualification Round begins, teams are able to submit their players to the OCC website where they will be pitted against other submissions. Matches are played and rankings are computed based on the outcomes. A scoreboard is available that will always show the current rankings and allows participants to view all played matches.

During Qualification, teams may submit a player as many times as they want. The idea is that each submission improves on the last so as to improve your overall ranking. Details on how we rank teams is in Section 2.1.

Once Qualification ends, the top 20%[1] of teams as determined by their final rankings will advance to the playoffs.

2.1 Rankings

The Scoreboard will list teams in order of their *ranking*. A ranking is computed based on your current win-loss record against all opponents who have submitted players. Each win is worth 2 points, each tie is worth 1 point, and each loss is worth 0 points. When two teams have the same win-loss record (i.e., the same number of points), the tie is broken by looking first at average number of points in each match played, then average margin of victory, and then average number of field domain points, and then finally by the submission that was made earliest.

When a team submits a player for the first time, a match will be played against each already submitted player from each team. This initial record will compute a team's first ranking. When a team submits a subsequent version of their player, their last win-loss record will be removed and a new win-loss record will be computed by playing the new player against each already submitted player from each team. Therefore, your ranking is always computed based solely on your last submitted player. And because your new submission may effect previous outcomes with other teams, other team rankings may be adjusted accordingly.

At the end of the qualification round, your ranking - which was computed based on your last submission - will be used to determine whether you advance to the playoff round. The last submitted player will be used during the playoffs (teams are not permitted to submit a new player after the qualification round closes).

2.2 Qualification Matches

A team submits code to be run via the OCC website on the team page. Submitting code in this fashion is the only way to make an official competition submission. Each time you submit a new entry, that entry

[1]If we have less than 40 teams, then the top 8 teams will advance to the playoffs.

becomes your official submission. The last such submission would be the one used during the playoffs if your team advances.

For each submission, you must submit *all* the code, not just what changed. For Python submissions, this is always a single file (all Python code must be contained in a single file). For Java and C++ submissions, this can be one or more files, but all files that are needed to compile a successful entry must be submitted each time.

Once a submission is received, the OCC website will queue a set of games between your new code and all the latest submissions from each of the other teams who have submitted code. Once all the games have been played, the rankings will be re-computed based on the results and the scores will be shown on the scoreboard. This can take a few minutes depending on how many other entries are already queued. Once the games have been played, you can view the results and you can watch any of the matches including those in which your team did not participate.

2.3 Playoffs

Teams advancing to the playoffs play in a single-elimination tournament (See Figure 1). Teams are paired based on seeding where the seed is the team's final rank. Each match is decided by a best 2-out-of-3 games played between the same two teams. The first team to win two games will advance to the next round.

The playoffs will be staged over a period of consecutive days after the qualification round ends. An entire round will be played each day with the results posted after all the matches have been played. Teams can watch each match via the OCC visualization engine.

For the Championship matches, we will attempt (but we do not promise) to run the games in front of a live, on-line audience so that everyone can see who wins (and how) in real-time. We will provide details as we get closer to playoff time.

We will hold a third-place match between the two semi-finalist teams that do not advance to the Championship match. This will also be played on the same day as the Championship match.

3 Player Execution Environment

3.1 Player/Game Interface

Your goal is to write a *player*. The implementation of a player is a stand-alone program written in Java, Python, or C++. It will run external to the game engine itself (indeed, it will run in its own process space) and communicates with the game engine via standard input and standard output. The game engine will automatically start each player and that player is expected to run continuously until the game is finished. At the start of each turn, the game engine sends each player a description of the state of the game. The player would read the description from standard input, choose an action for each child (a *move*) and then write the move back to the engine via standard output. The details of what data travels between the player and the game engine are described in Section 5.6.

3.2 Player-Centric Encoding

In order to simplify implementation, your player can *always* assume it is playing as the Red team. Internally, the game engine maps the team designated as Blue such that they appear to be playing from the Blue origin. This is important - even though you can play as either Blue or Red, you can just write

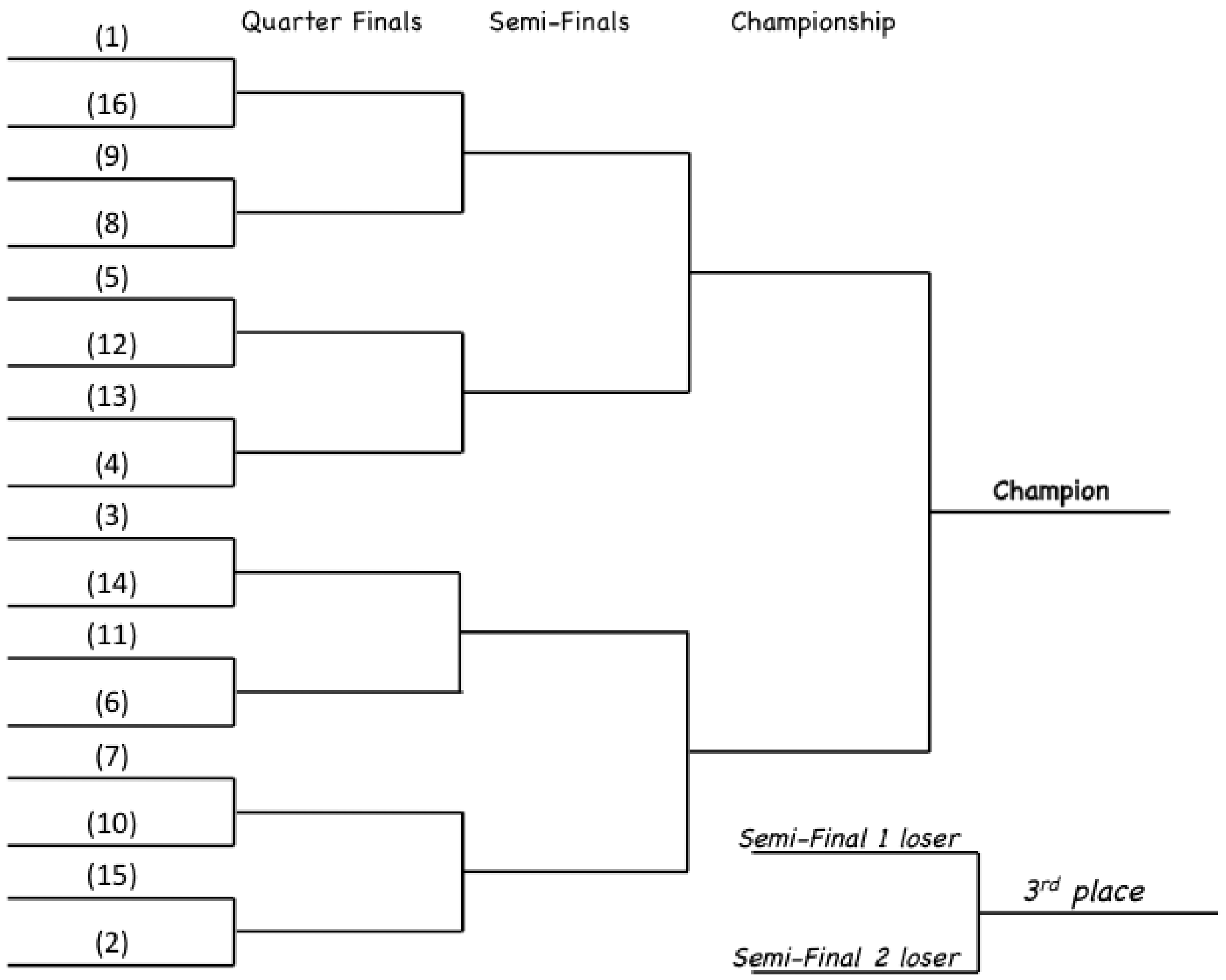

Figure 1: 16 team tournament bracket

the code assuming you are Red. Since the field is always symmetric, there is no difference in what color you are assigned when a match is played.

3.3 Debugging

Since communication with the engine uses standard output, you will not be able to use that in order to print out debug information to the console. Instead, use standard error instead (e.g. in Java this is `System.err`). You can also configure the game engine to provide you information on what it is doing to help you debug.

By default, the game engine operates in *practice* mode. In this mode, the engine will wait indefinitely for a player to provide a next move. Also in this mode, the player can open and write to files - this can be used, for example, to log your own debug output. However, when submitted, all players are placed in *tournament* mode. In this mode, players must provide moves within 0.5 seconds of being asked (otherwise their move is skipped). Also, when in tournament mode, no files or network connections may be used in any form (*intentional use could result in disqualification*). So be sure to test your player in tournament mode before submitted.

Since you are writing your player as a stand alone program, it will have its own `main()` function. When in tournament mode, this function will be passed a single argument containing the string `"tournament"` (in practice mode, no arguments are passed in). Your player code can use this argument to determine whether it is in tournament mode or not so that you can write conditional debug code.

See Section 4.2 for more details.

3.4 Tournament Execution Environment

Submitting your code via the OCC website is the only way to officially participate in the competition. Only members of your team can submit code on behalf of the team. The latest submission is always considered the current submission for the team and will be the one used to compute rankings.

OCC supports players written in Java, Python and C++. The following table shows the details for submissions in each supported language:

Language	*Supported Versions*	*Number Files Allowed*	*Entry Name to Engine*	*Environment*
Java	1.8.0_20	Any number ending in .java	Class with `main()`	javac (to compile) java -Xmx256M
Python	2.7.5, 3.4.1	1 .py file	Name of file	python2 or python3
C++	N/A	Any number ending in .cpp or .h	File with `main()`	g++ 4.8.2 -lpthread -O

3.4.1 Java

Once submitted, Java code will be compiled and run using Oracle JDK 1.8.0_20 - you may only use packages that are included in standard JDK 8. Your submission may consist of one or more `.java` files. As part of the submission process, you must provide the name of your main class. Note that it is required that all classes composing your player be in the default package (i.e., don't use packages for your player). During a tournament match, your Java player will be given a maximum heap size of 256MB. The submission is considered invalid if it does not successfully compile and the system will continue to use the last successful submission until a new submission that correctly compiles is submitted.

3.4.2 Python

Python code will be run using Python 2 or Python 3 depending on what was selected at upload time. Only one `.py` file containing Python can be submitted. No checks are run on Python code until a match starts, so all Python submission will appear to be successful as soon as they are submitted. At execution time, the appropriate Python engine will be used to run the code.

3.4.3 C++

C++ submissions are in the form of one or more `.cpp` and `.h` files. All `.cpp` files will be compiled and linked together into an executable. Only the C++ standard library will be available. Note that the only I/O that is allowed is in communicating with the game engine via standard in (`stdin` or `cin`) and out (`stdout` or `cout`). During compilation, the files are all placed and compiled from a single directory. Code will be compiled and linked with both the `-O` and `-lpthread` options.

4 Game Package

When the competition begins, you will be able to download the following resources:

1. This Competition Guide (`PaintballPanicGuide-1.0.pdf`)
2. The Paintball Panic Game Engine (`PaintballPanicEngine-1.0.jar`)
3. The Paintball Panic Game Viewer (`PaintballPanicViewer-1.0.jar`)
4. Sample Player in Java (`SamplePlayerSource/Java/*.java`)
5. Sample Player in Python (`SamplePlayerSource/Python/SamplePaintballPlayer.py`)
6. Sample Player in C++ (`SamplePlayerSource/C++/SamplePaintballPlayer.cpp`)
7. Two practice fields (`maps/Field1.txt, maps/Field2.txt`)

All of the above will be available in a single .zip file for easy download and install.

4.1 Installing Java 8

The game engine and the game viewer are both Java programs that require Java 8. If you do not already have Java 8 installed on your local machine, then you will need to download and install it. You can check if you have it by typing

```
java -version
```

If you see something that looks like:

```
java version "1.8.0_20"
```

(the numbers after the 1.8 don't matter) then you are good to go. Otherwise, follow these steps:

1. Open your browser to:

   `http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html`

2. Click on "Accept License Agreement" in the middle of the page.
3. Click on the appropriate download package. Most of you will use either the Max OS X x64 or the Windows x64 packages.
4. Follow the instructions for installing JDK 8 on your machine.

When this is complete, you should be able to run the above test to ensure that you can not only see Java in your path, but that you have the correct version. You may need to log out/in and/or restart your machine in order for the install to complete depending on your OS.

4.2 Using the Game Engine

By downloading the game engine, you are able to run your players locally so that you can create and debug a solution before you upload a submission to the OCC website. You can play against the sample player provided or you can create multiple players yourself to play against.

There are a number of options you can give to the game engine to help you during debug. They are documented in the following table.

Command-line option	Required?	Description
-map *map-file*	Yes	Specifies a field map file to use
-detail trace *trace-file-name*	No	Requests that the engine perform tracing which is output to the given file
-detail turns *turn-file-name*	No	Requests that the engine output all the turns to the given file
-tournament	No	Runs the engine in tournament mode. Must appear before -red and -blue.
-red *language entry*	Yes	Specifies the red player. See details below.
-blue *language entry*	Yes	Specifies the blue player. See details below.

The map file can be one of the sample fields provided (`Field1.txt` or `Field2.txt`) or it can be a file containing a map that you created.

The trace file is required if you want to run the Game Viewer (Section 4.3). It contains detailed output showing what the game engine actually did for each move.

The turns file shows what each player would see if looking at the output from the game engine. This conforms to the game encodings as described in the game manual with the exception that the entire board is always shown as visible. This file is useful for debugging as you can see the state of the game after each turn.

Before you submit your code, you should run the engine with the `-tournament` flag set. This flag must appear before `-red` and `-blue`. This mode will enforce that the player is not opening or accessing any files or network connections. In addition, this mode will only allow the player 0.5 seconds to provide a move (1.0 seconds for the very first move). If a move is not provided during this time, the engine will just skip the player's move. When `-tournament` is *not* present (i.e., the default), then the game engine runs *synchronously* with the player, meaning that the game engine will block until the player provides a move, making it easier to debug.

The `-red` and `-blue` flags are used to tell the engine where the code for each player is located and in what language it is in. The choices for *language* are either `java`, `python`, or `cpp`. The *entry* parameter states the name of your main class (in the case of Java or C++) or the name of the Python file.

When testing Java or Python players in your local environment, the game engine will use the command `java` or `python` respectively to run your player. This means that the appropriate run-time interpreter must be available in your path. In other words, you should be able to type `java` or `python` on the command-line and it should bring up the appropriate language interpreter. If you get an error, then you must adjust your operating system path in order for the command you are using to be present.

When testing a C++ player, the entry name is just the name of the executable that you compiled and linked.

4.2.1 Examples

The following example shows how you would run a Java practice match (the command would be typed on one line - it is formatted here for readability). They all assume you are running the java command from within the same directory as where you unzipped the game materials download package.

*N*ote: These examples assume that you have already written a player called MyPlayer and you are trying to test that. If you just want to run a sample player against itself to see how it works, see Section 4.4.

```
java
    -jar PaintballPanicEngine-1.0.jar
    -map maps/Field1.txt
    -detail trace game.out
    -red java MyPlayer
    -blue java SampleJavaPlayer.BasicSamplePlayer
```

The following example shows how you would run a Python practice match:

```
java
    -jar PaintballPanicEngine-1.0.jar
    -map maps/Field1.txt
    -detail trace game.out
    -red python MyPlayer.py
    -blue java SampleJavaPlayer.BasicSamplePlayer
```

Again, this assumes that the command `python` is in your path and it maps to the proper version you want, and that your player is in the directory with the engine JAR file.

The following example shows how you would run a C++ practice match:

```
java
    -jar PaintballPanicEngine-1.0.jar
    -map maps/Field1.txt
    -detail trace game.out
    -red cpp MyPlayer
    -blue java SampleJavaPlayer.BasicSamplePlayer
```

This assumes that you have compiled and linked your C++ code into an executable named MyPlayer that is in the directory with the engine JAR file.

The command that the server will use to compile a C++ entry is as follows:

`g++ -lpthread -O` *your-player.cpp*

4.3 Viewing a Game

To make debugging easier - and the competition even more fun and interesting - we provide a Game Viewer application[2]. The viewer takes the trace file that was output via the `-detail trace` option to the game engine. It will then show the game that is described in this file. The command is:

[2]The Game Viewer is a Java program that uses JavaFX to animate the game. This requires Java 8 in order to run correctly. See 4.1 for details on how to install it

```
java -jar PaintballPanicViewer-1.0.jar -trace  trace-file
```

where *trace-file* is the name of the trace output file (e.g., `game.out` from the examples in the previous section). If the visualization does not fit entirely on your screen, you can scale it by providing an additional `-scale` option that takes a number that specifies the percent of the original size to view the game. For example, adding `-scale 80` would show the game at 80% the size of the default viewing configuration.

Table 1 shows you what the various icons in the view mean. If a question mark appears over a child, it means that the action for the child was canceled because one or more pre-conditions for executing the action was not meant. These appear as alerts to the console and in the trace file.

In addition, the viewer shows you what parts of the field are visible at any given time by shading the visible squares in either light red (for what the red team can see) or light blue (for what the blue team can see). When both teams can see the same space at the same time, the square will be shaded in purple. Remember that field visibility is determined by a combination of planted flags and child locations.

🛠	Rapid-fire Adapter	🔵	Launched Blue paintball
🧍	Standing Blue Child	🧍	Crouching Blue Child
🧍	Defending (standing) Blue Child	🧍	Defending (crouched) Blue Child
⚑	Blue Flag	✸	Burst Blue paintball
✺	Blue paintball splatter	🧱	High Wall
Y	Basic Launcher	▬	Low Wall
●	Paintballs on the field	🔫	Rapid-fire Launcher
🔴	Launched Red paintball	🧍	Standing Red Child
🧍	Crouching Red Child	🧍	Defending (standing) Red Child
🧍	Defending (crouched) Red Child	⚑	Red Flag
✸	Burst Red paintball	✺	Red paintball splatter
🛡	Shield	🌲	Tree

Table 1: Key to icons in Game Viewer

4.4 Running Sample Players

Sample players are provided in each supported language. Much of the code can be re-used exactly as is, especially the code that reads and writes to the game engine. However, you are free to start from scratch, modify the sample players, etc.

The sample players are provided purely to get you started. They implement no real game logic as they basically choose random actions each move.

Instructions on how to use the sample players to practice against are shown in Section 4.2.1. If you want to just quickly play and watch a game, you can just play the sample player against itself and then watch on the viewer. The following commands show you how:

```
java
    -jar PaintballPanicEngine-1.0.jar
    -map maps/Field1.txt
    -detail trace game.out
    -red java SampleJavaPlayer.BasicSamplePlayer
    -blue java SampleJavaPlayer.BasicSamplePlayer

java
    -jar PaintballPanicViewer-1.0.jar
    -trace game.out
```

5 Game Rules

This section describes the rules for Paintball Panic.

5.1 Overview

Paintball Panic is played on a 31x31 grid (playing field) between a red team and a blue team. Each player controls a team of four children. The goal is for the team to score more points than the opponent during a game. Points are scored for each paintball that bursts on or near an opponent. In addition, points can be scored by collecting certain objects located on the field of play, playing defense, and by claiming areas of the field.

A game is played over the course of 150 rounds where each round consists of each team taking a turn. The turns are run in parallel so that both teams are logically operating at the same time. The team with the highest score after 150 rounds wins. In the event of a tie, the player with the largest field domain (see Section 5.5.1 for an explanation of *field domain*) will win.

Players direct their teams by providing instructions for each child to the game engine at the beginning of each round. Once the game engine has received instructions (a *move*) from both players, the game engine executes the instructions per the rules of the game. At the end of the round, the game engine provides detailed data back to each player as to the current state of the game. The player then computes a next move based on this data to begin the next round and so on.

A visualization of a sample playing field[3] is shown in Figure 2. This figure shows a number of field elements including the $X - Y$ coordinate system, starting positions of the children, static field elements, and items (like shields, rapid-fire adapters, and paintballs) that can be picked up.

5.2 Scoring

Scoring during the game is as shown in Table 2. There are also a number of one time bonuses that are awarded at the end of a game as shown in Table 3.

Objective	Points Awarded
Paintball hits child directly and bursts	30
Paintball hits child directly - no burst	20
Child gets paint splatter indirectly	16
50 rounds with no child getting painted	30
Hits own team (whether bursts or not)	-11
Successful defense	17
Planting Flag	12

Table 2: In game scoring objectives

Paintballs are fired from launchers. A launched paintball that hits a child may or may not burst depending on distance to the child. A paintball can also burst if it hits something solid, which may splatter paint on a nearby child. If a paintball hits a child (regardless of whether it bursts) the child it hit will drop 5 paintballs onto one of the eight randomly chosen (and empty) spaces adjacent to the hit child.

[3]There are many fields on which the game will be played. This is just a sample of what one field might look like.

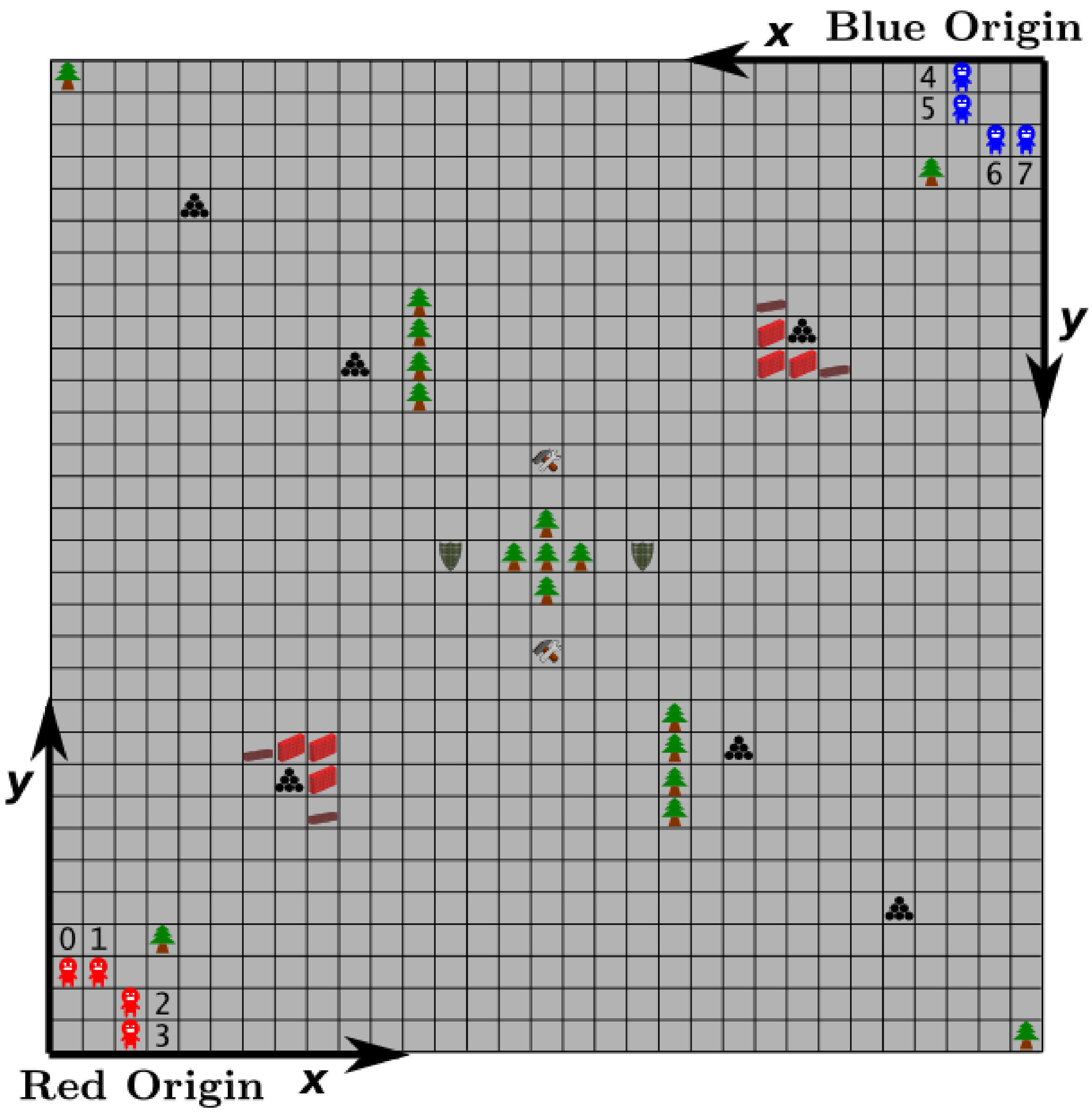

Figure 2: Overview of the full playing field (sample map).

End of game condition	Points Awarded
Unused paintballs in possession	1 per every 2 balls
Possession of wooden shield	6 per shield
Possession of rapid fire launcher	10 per rapid-fire launcher

Table 3: End of game scoring objectives

Each child starts with a single, basic launcher and a supply of 50 paintballs. Additional stashes of paintballs are located in the playing field. If a child reaches a location with the additional balls, then that child can take possession of all or some of those balls. Children on the same team can also transfer balls from one to another. This is accomplished when one child drops paintballs onto the field and another picks them up. This can be coordinated such that the transfer takes place in a single turn.

There are wooden shields on the field. A shield can be used to block inbound paintballs. If a ball strikes the shield, then no points are awarded to the shooting team - this counts as a miss - but the defending team is awarded a Successful Defense score. Once a child obtains a shield, the child can be placed into a defensive mode. While in this mode, the child may not launch any paintballs and has his move distance restricted (see Table 4).

There are also rapid-fire adapters on the field. A basic launcher can turn into a rapid-fire launcher if an adapter is obtained. Once obtained, the basic launcher is transformed into a rapid-fire launcher for the rest of the game. Only one ball at a time can be launched from a basic launcher. A rapid-fire launcher allows 3 balls per launch. Rapid fire launching means that the balls are launched in a spread formation making it easier to hit targets.

Flags may be planted on the field by children. Flags serve two purposes. First, they are worth additional bonus points at the end of a match. Second, they serve to claim *field domain* (see Section 5.5.1). It costs a player 25 paintballs to plant a flag.

Children are restricted in what they can possess (i.e. their *inventory*) at any given time according to these rules:

- Children may carry nothing.
- Children may carry up to 100 paintballs at any one time.
- Flags do not count towards inventory. (A flag is exchanged for the cost of 25 paintballs at the time it is planted.)
- Children may carry any of the following combinations: one launcher (any type), one shield, one shield plus one launcher (any type). With any of those combination, a child may also carry paintballs. See Table 5.

Each game is played on randomly chosen field from a selection of fields. Each is configured in a different way and may include different numbers of shields, paintballs, trees, rapid-fire adapters, and walls. You are guaranteed that the field is always symmetric in terms of where obstacles are placed and where items that can be picked up are placed. There will always be at least two rapid-fire adapters and two shields available on the field.

5.3 Actions of a Child

At the start of a turn, each player specifies a single action that each child on her team will perform. The set of 4 actions (one per child) is called a *game move.*

5.3.1 Idle

This instructs the child to stay in her current position and do nothing. This action is also automatically taken if the specified action cannot be performed for any reason, including violating the rules of the action.

The command:

```
idle
```

5.3.2 Move

Children can move up to 3 Euclidean spaces in a turn (see Figure 3) when standing and not defending, up to 2 Euclidean spaces (see Figure 4) while standing and defending, or 1 Euclidean space (see Figure 5) while crouched (defending or not) - see Table 4. The `move` command takes the grid coordinates of the destination cell. If there is an obstacle between the child and the destination, the child will move along the path until blocked by the obstacle and will remain in the cell before the obstacle for the remainder of the turn. Two children may not occupy the same space at the same time.

	Standing	Crouching
Defending	2	1
Not Defending	3	1

Table 4: Maximum movement distances

Children do not move instantaneously. Instead, they move throughout the turn passing through each grid location on the way to the destination. The child moves at a linear speed, over the course of n steps where n is the Euclidean distance to the destination.

A child cannot move off the field (if attempted, the child will just stop at the grid location at the edge of the field). If the coordinates are invalid, then the child will remain idle for the turn.

The command:

```
move x y
```

5.3.3 Pickup

This action instructs the child to take something currently on the field that is in one of the eight grid locations adjacent to the child. The action takes an argument specifying the location of the item to be picked up. If there are paintballs at the location, then an optional second argument indicates the number of paintballs to pickup. If the child tries to take more balls than what is there or does not provide a count, then all the balls will be taken.

This action will have no effect in the following cases:

- If the child is picking up a rapid-fire adapter, but does not possess a basic launcher.

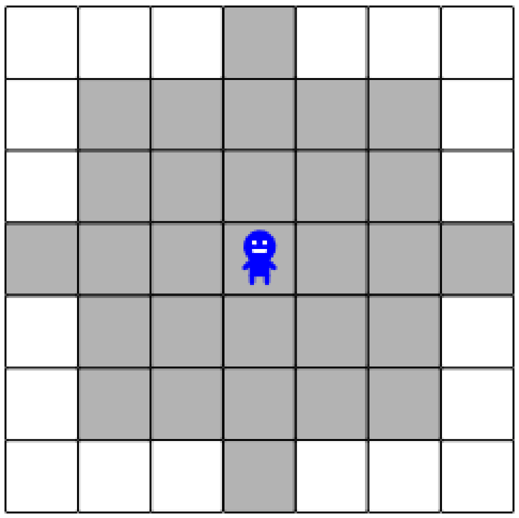

Figure 3: Valid moves within 3 Euclidean spaces.

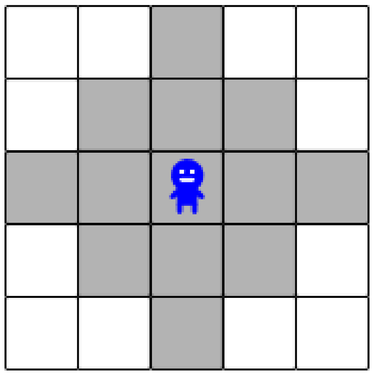

Figure 4: Valid moves within 2 Euclidean spaces.

- If the child is picking up a shield, but already holds a shield.
- If the child is picking up a launcher, but already holds a launcher (i.e., a child can hold only one launcher at a time).
- If the child is picking up paintballs, but already is holding the maximum allowed.
- If two children try to pickup from the same space on the same turn.

If the action has no effect or there is no object to pick up, then the child will remain idle for this turn. Picking up an item must result in one of the inventory configurations as shown in Table 5.

The command:

```
pickup x y [ n ]
```

where n is the number of paintballs to pick up.

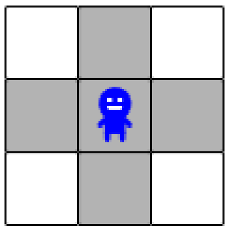

Figure 5: Valid moves within 1 Euclidean spaces.

5.3.4 Launch

This action tells the child to launch a paintball towards a grid location. A paintball can travel at most 24 Euclidean spaces. The paintball starts at a height of 9 units and drops at the following rate: the first half of the flight is at the starting height. The second half drops linearly over the remaining distance (see Figure 6). The ball stops when it hits something, exits the field, or its height is zero. If the ball hits the ground within one adjacent location of an opponent child, it counts as splattering the child unless the child was using the shield to defend.

If the child has a rapid-fire launcher, then the launch operation will launch 3 balls concurrently providing something of a scatter effect thereby covering a wider hit radius. In this case, the launch command will specify a start and an end coordinate. One ball will launch at the first, one will launch at the last and the third will launch at the coordinate that bisects the line that connects the two given coordinates.

The command:

```
launch x y [ end-x end-y ]
```

If the extra coordinates are given but the child is not carrying a rapid-fire launcher, then they are just ignored (but a single paintball is fired as normal). If the extra coordinates are not given but the child has a rapid-fire launcher, then 3 paintballs will be fired at the same *x,y* coordinates.

If a ball hits a defenseless child, then that child will drop 5 paintballs on a randomly chosen, empty (or already containing paintballs) space adjacent to the hit child. Splattering a child will not cause the child to drop paintballs.

5.3.5 Defend

This action only works if the child is carrying a shield, otherwise it is ignored. When given, the child is placed into defensive mode: the child may not launch any paintballs and the child will only move up to 2 spaces (see Figure 4) in a turn when standing. Any enemy paintball that would have hit the child while defending will be deflected by the shield thereby registering as a miss (friendly fire hits cannot be defended against). Also, a child will never get any splatter from nearby burst paintballs when defending.

The command:

```
defend
```

5.3.6 Undefend

This action causes the child to stop defending. If the child was not defending, then this action has no effect (the child is idle).

The command:

```
undefend
```

5.3.7 Drop

A child can intentionally drop any object she is carrying. The command takes the adjacent grid coordinates in which to drop and the object (shield, basic launcher, rapid-fire launcher, or paintballs) to drop. In the case of paintballs, an integer *n* is given that represents the number of balls to drop. If the number is greater than the number of paintballs being carried, then all paintballs being carried are dropped in the location specified. If the child is not currently carrying the item to be dropped, then the command is canceled and the child remains idle.

A child might want to drop items if another child from the same time can retrieve them before a child of the other team.

Each space can hold only one item at a time. Hence if there is already an item in the target space when the `drop` command is given, the command is canceled and the child is idle. The exception is that a child can drop paintballs onto a space that already holds paintballs thus increasing the number of paintballs on that space. If two or more children try to drop an item (including paintballs) onto the same space in the same turn, then the action is canceled for both children and both children remain idle.

The command:

```
drop x y item
```

where *item* is one of `shield`, `basiclauncher`, `rapidlauncer`, or *n* where *n* is a positive integer that represents the number of paintballs to be dropped.

5.3.8 Crouch

This command puts the child into a crouching position. When crouched, the child measures 3 units tall. Moving while crouched is considered to be crawling hence the child can only move 1 Euclidean space (see Figure 5) per turn (either forward, backward, left or right).

A child cannot launch paintballs or plant flags while in a crouched posture.

The command:

```
crouch
```

5.3.9 Stand

This command brings the child to a standing position. When standing, the child measures 9 units tall. Moving while standing allows the child to move up to 3 Euclidean spaces per turn (see Table 4).

The command:

```
stand
```

5.3.10 Plant

This command causes a flag to be planted in the designated grid location (which must be adjacent to the child planting the flag). In order for this action to succeed, the child who is planting the flag must be standing and possess at least 25 paintballs (if not, the child is `idle` for this turn). Upon planting, the 25 balls will be deducted from this child's inventory. Note that you do not carry flags explicitly, but instead you are simply swapping 25 paintballs for a flag to plant during this action. Hence you cannot pickup nor can you drop flags.

The command:

```
plant x y
```

5.4 Simulation

Once each player has provided instructions for their children, the game engine checks that the actions are legal. If an action can't be performed, then the `idle` action is taken for that child.

All actions take one turn, but they don't all occur at the same time. The order of actions is as follows:

1. Idle
2. Drop
3. Pickup
4. Stand, Crouch, Defend, Undefend (these actions happen concurrently)
5. Plant
6. Move, Launch (these actions happen concurrently)

To be clear, the above determines the order that each child performs her action. So a child that is to perform a `drop` action will perform it before another child performs a `pickup` action. This allows items to be passed from one child to another during the same turn.

If two children try to pickup (or drop) in the same location at the same time (during the same turn), then their actions are canceled and they will each be idle.

Moving and launching happen concurrently.

5.4.1 Linear Paths

Moving children and launched paintballs follow linear paths in the field of play. Children and paintballs take a number of steps to complete their action and may pass through intermediate spaces along their way to their final destination.

Moving entities move as follows. Consider an entity that starts at position x_1, y_1 and moves towards position x_2, y_2. The movement is implemented in n steps where $n = \max(|x_2 - x_1|, |y_2 - y_1|)$. The steps will be spaced evenly in time during the turn, with the first step occurring $\frac{1}{n}$ of the way through the turn and subsequent steps occurring at times $\frac{2}{n}, \frac{3}{n}, \ldots, \frac{n}{n}$. At time $\frac{t}{n}$, the entity moves to location $x_1 + \text{round}(\frac{t(x_2-x_1)}{n}), y_1 + \text{round}(\frac{t(y_2-y_1)}{n})$. Thus, by the end of the turn (when $t = n$), the entity will reach its destination.

Rounding is performed by a slightly modified version of the Java `Math.round()` function as given below.

```
public static int roundAwayFromZero( double x )
{
    if ( x < 0 )
    {
        return -(int)( Math.round( -x ) );
    }
    else
    {
        return  (int)( Math.round( x ) );
    }
}
```

5.4.2 Paintball Traversals

Paintballs follow the linear path traversal function defined in the previous section. Paintballs also include a height (Z) dimension. A launched paintball always starts at a height of 9 units and attempts travel to the specified target location (as given in the `launch` command). It will travel the first half of its total distance at height 9. It will then drop linearly over its remaining distance. So, assuming the target is 24 spaces away, at 18 spaces from launch point the height will be 4.5 units. See Figure 6.

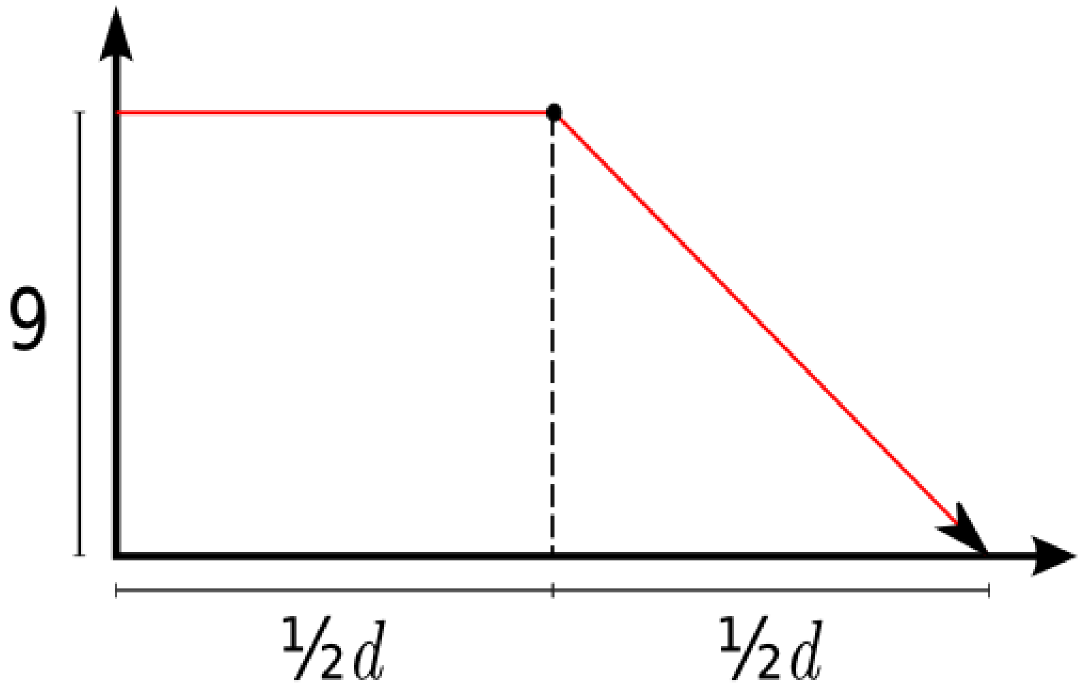

Figure 6: The trajectory of a launched paintball. The initial height is 9 units. The total distance traveled is d. The height of the ball during first half of the flight is at the initial height. The height of the ball during the second half drops linearly over the remaining distance.

The paintball could stop before reaching its target if it exits the field or it hits a child, a child holding a shield, or a field obstacle. If the ball hits an obstacle within the first half of its flight, then it will always burst and splatter. Any child on the opposing team within 1 space adjacent to the obstacle will be splattered *unless* the child is crouched behind a wall or a flag, crouched or standing behind a tree, or currently defending. Note that an obstacle acts as protection only if it is between the launcher and the child. See Figures 7 and 8 for a more detailed explanation and illustration of the splatter and safe zones for this situation.

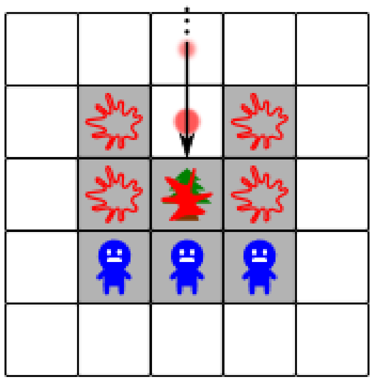

Figure 7: Example of a splatter zone when a paintball is coming from the north. Symmetric zones are established when a paintball comes from the south, west, and east. In this illustration, the paintball hits the obstacle (a tree in this case) and then splatters into the spaces denoted by the open splatter icons. The children icons represent the safe spaces where they will not get splattered. Note that children are always safe from splatter in the safe zone when they are crouched. They are only safe from splatter in the safe zone when standing when behind a tree.

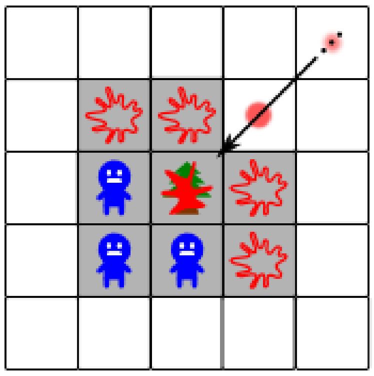

Figure 8: A second example of a splatter zone when a paintball is coming from the northeast. Symmetric zones are established when a paintball comes at an angle from the northwest, southwest, and southeast.

A paintball that directly hits a child that was defending registers as a miss for the launching team and a successful defense for the defending team.

5.5 Playing the Game

The game is played over the course of 150 rounds, where each player directs her four children. This section explains the board and child encodings that are sent from the game engine to each player at the beginning of each turn.

5.5.1 Visibility

Players cannot necessarily see the entire board. Instead, players can see what their children can see plus grid locations within their *field domain.* Any grid location less than 8 Euclidean spaces from their team's planted flag that are closer to their flag than to an enemy flag are part of that team's field domain.

Grid locations that are visible to the player include all of the player's field domain plus all grid locations less than 8 Euclidean spaces from each child on the team.

5.5.2 Attributes of a Child

Each child is either standing or crouching and is either in defensive mode or not. A standing child is always 9 units tall. A crouching child is always 3 units tall.

Each child can hold up to 100 paintballs in addition to a maximum of two items. Table 5 enumerates all the possible combinations that a child may be carrying (not including paintballs) and how they will be encoded by the game engine. Note that a child may not hold two shields or two launchers.

Child Holding	Encoding Symbol
Empty-handed	a
One basic launcher	b
One shield	c
Shield + Basic Launcher	d
One rapid launcher	e
Shield + Rapid Launcher	f

Table 5: Child Inventory Encodings

5.5.3 Board Layout

The field is a 31x31 grid. Spaces are indexed by X and Y, with X going left to right from 0 to 30 and Y going bottom to top from 0 to 30. See Figure 2 for more detail.

Some spaces will be occupied with unmovable obstacles. A *tree* measures 20 units tall and therefore will always block any child or paintball that tries to pass through the space it is in. A *high wall* measures 7 units tall while a *low wall* measures 3 units tall.

Obstacles and field items are placed randomly on the board, but always symmetric so that the field is fair. If an obstacle or item is placed at space x, y, then the same type of obstacle or item will also be placed at space $(30 - x, 30 - y)$. Obstacles will not be placed at the starting position of any child or in a pattern that could make part of the board unreachable.

Once a flag is planted onto a space, it behaves as a high wall (i.e., it has a height of 7 units).

Table 6 details exactly what can be in any space and the symbol used to encode its contents. A space holding paintballs has an associated number of balls. If all the paintballs are removed from a space, then it will become empty (i.e., the encoding $P0$ will never appear).

Space Contents	Encoding Symbol
Empty	.
Tree	T
Paintballs	Pn
Shield	S
Rapid Fire Adapter	A
Basic Launcher	L
Rapid Fire Launcher	F
Low Wall	V
High Wall	W
Red Flag	R
Blue Flag	B

Table 6: Field Space Encodings

5.6 Communicating with the Game Engine

Players communicate with the game engine by writing text to standard output (`System.out`) and reading text from standard input (`System.in`).

5.6.1 Player Input Format

The game engine communicates with each player so that each player can believe that they are always the Red player (the one that starts at the lower left corner of the field toward position 0,0). At the beginning of each round, the game engine will send the following information to each player:

```
turn number
current score
board configuration
children configurations
```

The turn number begins at zero and increments by one for each turn. At turn zero, each player sees the initial state of the field plus the initial state of each of their own children.

The current score is given as two integers separated by a single space where the first integer is always your score and the second integer is always the opponents score. For example,

```
145 95
```

The board configuration is given in 31 rows, each line containing 31 whitespace-separated[4] space descriptions (see Table 6). The j^{th} space description on line i is one of the space content encodings that describe the field space at coordinates (i, j). Note that since row i in the report actually describes spaces with an x coordinate of i, the field description would appear to be rotated by 90 degrees if you looked at it printed out.

If a space is visible to the player (based on field domain rules), then the space has one of the encodings in Table 6. If a space is not visible, then it will have '*' (a single asterisk) as its description.

[4]One or more space characters.

The board configuration is followed by 8 lines making up the children configuration section. Each line represents a child. The first 4 lines always describe the children on your team and the next 4 lines describe the children on the opposing team. The ordering of the child descriptions matches the order given in the starting position table below:

Child Number	Team	Starting Position
0	Red	0, 2
1	Red	1, 2
2	Red	2, 1
3	Red	2, 0
4	Blue	28, 30
5	Blue	28, 29
6	Blue	29, 28
7	Blue	30, 28

A child is described by six whitespace-separated fields as follows:

x y posture defending inventory paintballs

where ***posture*** is either 'S' for standing or 'C' for crouching, ***defending*** is either 'D' for defending or 'U' for not defending, ***inventory*** is an encoding symbol from Table 5 and ***paintballs*** is the number of paintballs the child is carrying.

If an opponent child is not visible, then the line for that child description will contain a single '*' character. Children on your team are always visible.

5.6.2 Player Output Format

The player will print a single output line to standard output representing a desired action for each child it controls. The order is always for children 0, 1, 2, and 3. The commands are those given in Section 5.3. The following is an example game move for your four children:

```
move 0 5
move 1 5
move 3 2
crouch
```

This moves the first three children and leaves the last child in the same space, but now in a crouched position.

6 Rules

6.1 Tournament Rules

All participants competing in Operation: Code Clash (OCC) must abide by these rules:

1. Any questions related to the tournament or the rules of the game should be raised to the OCC team as soon as possible. OCC will make a decision and post it on the website for all participants. Decisions of OCC are final.

2. Teams are expected to report all bugs as soon as they are noticed so that the OCC team can make any necessary adjustments.

   (a) A *bug* is a contradiction between the behavior of the game engine and the rules of the game, or a contradiction within the rules of the game.

   (b) The intent of the game shall override the behavior of the engine caused by bugs in the engine.

   (c) Teams should report bugs either via the website or to info@operationcodeclash.org as soon as possible. OCC reserves the right to post any bug reports so that all teams can understand the issue. OCC will take measures to ensure that a team's strategy is not revealed in the process of posting a bug report.

   (d) OCC will determine, in its sole discretion, whether or not to fix the bug and what, if any, the fix might entail. In any case, OCC will strive to ensure fairness in any fix it makes.

3. In order to ensure a smooth end to the qualification section of the tournament, no bugs will be fixed within 2 days prior to the final code submission deadline.

   (a) Bugs may still be reported during this time.

   (b) However, neither the game engine nor the game rules will be altered during this time. Hence, the behavior of the engine at that point will be the deciding factor in how rules are interpreted.

   (c) OCC will determine, in its sole discretion, whether any major bugs were reported in the final 2 days that must be fixed before the playoff period begins. If this happens, then teams that qualified for the playoffs will be given 2 days from the time the fixed engine is released to test out their code and then re-submit for the playoffs. Note that we will not re-qualify teams based on the fix; teams will qualify for the playoffs based on the game engine that was available at the end of the qualification section.

4. OCC reserves the right to make other adjustments to the game manual during the course of the competition if necessary based on how the competition is running. We do not expect to make any changes in this way once the qualification section begins.

6.2 Ethics and Honor Code

We expect all teams to compete fairly. Any team that is found to violate these rules are subject to immediate disqualification. All decisions of OCC are final.

- All code submitted to OCC must be written only by the *students* on the team.

- A student can be on one team only. A student should submit code only on behalf of the team on which the student is registered.

- OCC will respond and react to any report of unethical behavior. Such reports should be sent via email to `info@operationcodeclash.org`.
- Teams should report bugs as soon as they are found. Intentionally taking advantage of an unreported bug, or waiting until the end of the qualification period to report a known bug may be considered as a violation of this honor code.
- Teams shall not intentionally manipulate the scoring methods or otherwise try to gain access to the scoring database.
- Teams shall not attempt to alter, disrupt, reverse-engineer, or otherwise interfere with the proper operation of the Operation: Code Clash website.
- Teams shall not attempt to gain access to any OCC restricted information or data.
- Submitted code shall make no attempt to access the local file system or access the computer network. Code is run in a restricted environment and any attempt to open a network connection or access the file system will result in disqualification. If there are any questions, team should not hesitate to contact OCC before submitting any code.
- OCC has established a public forum where questions can be asked and answered by any members of the community. We encourage you to use that resource. Sensitive issues may be addressed directly to OCC via `info@operationcodeclash.org`.
- Offensive or vulgar language, harassment of others, and intentional annoyances are not permitted on the OCC website, its associated forums, or any in submitted code. Any such incident will result in participant expulsion.

7 Revision History

7.1 Version 1.3

- Revert incorrect Figure 3 back to original, correct version.

7.2 Version 1.2

- Fix Figure 3 which showed an invalid 3-space Euclidean move. The text was correct, but the illustration was slightly off.
- Missing the word "travel" in the section on Paintball Traversals

7.3 Version 1.1

- Fix typo in first example from section 4.2.1 (the "s" in sampleJavaPlayer needs to be uppercased).
- Attempt to clarify the wording in section 4.2.1 to explain that these examples are for showing a player that a team wrote and that to run an example showing the sample players playing against each other was in another section.