+1 KROM - 8IXI8 ™

Last Modified: (01/24/2020 13:14:43)

The Goal

Destroy the opposing player's factory

Website: [8ixi8.com]

Type of Game

- 2 players
- synchronous, turn based
- human vs. human
- network/online play

Starting game configuration

- Game is played on 16 x 16 grid of squares in a rectangular playfield
- Each player is either red or blue.

- The game server chooses each player's color when the game starts.
- Each player has 3 different kinds of *pieces*:

1. Factory
2. Wall
3. Gun

- Each player also has Camo (camouflage) that they can apply to spaces which covers and hides their pieces from the opponent. (see *Camo*)
- Initial game set up consists of each player having one factory with 3 walls immediately adjacent to it, between it and the opponent's factory/walls. (See *Diagram*)
- The game ends when one of the factories is destroyed.
- The player whose factory is destroyed is the loser and the other player is the winner.

Turns

- Red always gets the first turn.
- On each turn a player is awarded 3 moves
- During a player's turn, they may take 1 or more of the following moves:

1. Move a piece
2. Create a piece
3. Fire a gun
4. Deploy camo
5. Pass

Each of these moves consumes a specific number of moves as defined in the table below. *Pass* allows a player to end his/her turn without consuming any more moves.

When a player *passes*, any unused moves roll over to their next turn. In this way, a player can save up moves for later use.

The number of moves that a player can make during their turn is determined by the number of moves they have. For example, if a player passes on their first turn, they will be able to consume up to 5 moves on their second turn.

Moving a Piece

Moving a piece means to change the board position of a piece from one space on the board to an adjacent space. A piece can only be moved straight up, down, left or right. They can not move diagonally, skip a space, or jump a piece.

- A piece can only move by one space.
- Pieces can only move to an open space.
- A player can only move their own pieces.

Creating a piece

Creating a pieces involves adding a new piece to the board. Created pieces can only appear in a space that is adjacent (N,S,E,W) to a player's factory or some other piece of the player's color, which is itself, adjacent to their factory, or any distance away, as long as

there is an unbroken chain of adjacent pieces belonging to the player, leading all the way back to the player's factory. (see *Diagram*)

- Diagonally adjacent pieces are not considered *adjacent*.
- Only wall and gun pieces can be created.
- Only pieces of the player's own color can be created.

Firing a gun

Only guns can be fired. Guns can only fire in a straight line, in one of the 4 cardinal directions The closest piece occupying a space in the firing direction from the gun is considered *hit*. Different pieces require a different numbers of hits before they are considered *destroyed*. (see *Hit Table*) When a piece is *destroyed* it is removed from the playfield. Guns can destroy any piece of any color. If no pieces are in the path of the gun's projectile, then the shot disappears harmlessly into the void; No hit occurs.

Camo

Camo, or *camouflage* is not a *piece* but more a *property* of a playfield space. It has no player color associated with it. A player can see his/her own pieces when underneath camouflaged spaces, but not pieces belonging to the other player. A player can move, create and fire pieces from underneath camo and their opponent won't be able to see these moves, though firing a gun which results in a hit beyond the camo, may give the opponent some hints as to where the shot originated. When a piece is destroyed, any camo that was applied to that space is also destroyed. Deploying camo follows similar rules to those of creating pieces:

- A player can cover any space that is adjacent to their factory, or to their factory in a chain of *adjacentness* comprised of their own pieces.
- When camo is deployed, only the *deployment* space is covered.

User Interface

The actual mechanics of how a user indicates their moves to the computer during their turn is not defined here. This can be implemented in any way the developer chooses.

This game is designed as a *pure* game concept and only defines the rules of the game. The details of the UI and how it functions are not part of the rules and are thus not defined here. This is intentional. Developers are free to create a game client in any way they choose. It could be played as a real-time 3D first person experience, or it could be played using a text based TTY. Either way, the rules and the server to which they connect are the same, so all players participate on an equal footing.

Scoring and Ranking

There are two modes of play:

1. Ranked play
2. Free play

Ranked play

- *Ranked play* requires registration of a *universally unique player id* through the *official registration organization*.
- When a player plays the game, their game client connects to an *officiated matching server* that pairs the player with whoever is next in line to play.
- The player has no control over who they will play against.
- The server drives the game and enforces the rules — cheating is not possible.
- Wins and losses are recorded and computed to produce the player's universal ranking.

Free play

- Registration is not required.
- Players are free to choose any opponent.
- Wins and losses *do not* figure into the universal rankings.
- Players may rendezvous using any official or privately run game server.
- The server drives the game and enforces the rules.
- Arbitrary organizations of players based on skill level, location, or any other criteria are allowed to form so that players can have some control over who they play against.

Client/Server protocol

Transport

Connections use ordinary TCP Internet socket connections.

Both client and server communicate using *messages*. Individual *messages* are in the form of compliant JSON strings.

Communication consists of opening a socket, reading and writing 1 or more complete messages, and then closing the socket.

If a client loses their connection to the server during game play, they will forfeit the game to the other player. A forfeit is counted as a win for ranked play. However, the win is also marked as a forfeit for statistical and historical purposes.

Data types

User and ORP IDs

A *userid* is something that uniquely identifies a player. For non-ranked play, this can be any convenient value that a server needs to distinguish between users.

For ranked-play, the *user-id* is an official, and universally unique value that is assigned to a single player forever. It is used to record official data in the worldwide ranked-play database. This special type of *user-id* is referred to as an *Official Ranked Player ID* or *orp-id* .

Authentication info For official ranked-play, digital certificates will be used for authentication of *orp-ids*.

Time stamps

A *timestamp* is a POSIX time stamp. It is the number of seconds since the Epoch. See http://en.wikipedia.org/wiki/Unix_time.

Protocol messages

hello Sent by client.

Client sends this message after a connection to server is established. It must be the first message sent. If this msg is not acceptable to the server in any way, it will simply disconnect. If the server likes the message, it will respond with a *welcome* message.

```
{
        {
                msg: "hello",
                nick: "Bart",
        }}
```

redirect

Sent by server.

May be sent in response to the *hello* message. It is used by a server to tell the client to disconnect and reconnect to a different server.

```
{
        {
                msg: "redirect",
                host: "someotherhost.com",
```

```
                why: "This server is way too busy.",
        }}
```

The *why* field is optional and should include an explanation for the redirection.

welcome

Sent by server.

Sent in response to an acceptable *hello* message from the client.

```
{
        {
                msg: "welcome",
                id: "user-id",
        }}
```

The *id* value is generated by the server. It is used to uniquely identify this player throughout the game.

join

Sent by client.

Sent when the client wishes to play.

```
{
        {
                msg: "join",
        }}
```

wait Sent by server.

Sent when the server has placed the client on the list of those waiting to play.

```
{
        {
                msg: "wait",
                why: "Because there are a lot of players ahead of you",
        }}
```

The *why* string is an explanation for why the player must wait.

start Sent by server.

This message is sent when the server is ready to start the game.

```
{
        {
                "msg":"start",
                "moves":3,
                "game":{
                        "id":"g-5",
                        "start":1284273632458,
                        "players":[
                                {"id":"u-3","nick":"Lisa", "id":"u-4","nick":"Bart"
]}}}}
```

The *game* object holds a unique game identifier and the time the game started. It also holds an array of the players in the game in the order they arrived and were placed into the

waiting queue. The first player in this list is always the one who gets the first turn. The *moves* value is the number of moves each player has at the start of the game.

you

Sent by server.

This is sent when it is a player's turn to make moves. The player who connected earliest to the server always moves first, so they will be the first to receive this message.

```
{
        {
                msg: "you",
                moves: 3,
        }}
```

The *moves* value is the number of moves the player has available to play in this turn.

move

Sent by client.

The client sends this message to fulfill it's turn. It contains an array of moves to perform. Each move is one of those listed in the *Turns* section above.

```
{
        {
                msg: "move",
                moves: [
                        [ "camo", 1, 3 ],
```

```
                        [ "make", 1, 4, "gun" ],
                        [ "move", 1, 2, 1, 3 ],
                        [ "fire", 1, 3, 0 ],
                ]
        }}
```

The number of moves in the array can not be more than the number of moves the player currently has available. However, it *can* be less. If it's less, then the player will carry moves over to their next turn.

When the server receives this message, the appropriate number of moves is deducted from the moves available for the player's next turn. If the moves result in the player winning, then the *over* message will be sent to both clients.

moved

Sent by server.

When a client sends the *move* message (above), the server will update the game state. Then it will pass those moves on to the opposing player with this *moved* message.

```
{
        {
                msg: "moved",
                uid: "bart",
                moves: [
                        [ "camo", 1, 3 ],
                        [ "make", 1, 4, "gun" ],
                        [ "move", 1, 2, 1, 3 ],
                        [ "fire", 1, 3, 0 ],
```

```
                ],
        }}
```

The *uid* value indicates which player performed the moves.

over

Sent by server.

When a game is over, this message is sent by the server to both clients. After this message is sent, the server will close the socket.

```
{
        {
                msg: "over",
                results: {
                        id: "game-id",
                        start: timestamp,
                        end: timestamp,
                        won: "user-id",
                        lost: "user-id",
                        forfeit: false;
                , log: [
 event: 'move', when: timestamp, id: "user-id", move: [ "move", 1, 2 ] ,
 event: 'move', when: timestamp, id: "user-id", move: [ "make", 1, 2, "wall" ] ,
 event: 'move', when: timestamp, id: "user-id", move: [ "fire", 1, 2, 0 ] ,
 event: 'move', when: timestamp, id: "user-id", move: [ "camo", 1, 2 ] ,...
 event: 'disconnect', when: timestamp, id: "user-id" ,
 event: 'over', when: timestamp ]}}
```

Results will contain general information about the game. The *id* and *start* values will be the same value as those from the *start* message. The *end* value is the time-stamp for when the game ended. The *won* and *lost* values are the corresponding *user-ids*. If you are the winner, then *won* will be the same as your user ID.

The *log* array is an ordered listing of all the moves performed by both players during the game as well as other events of note. The entire game, and game state from start to finish can be recreated from this log.

The *disconnect* event may or may be present. It indicates that the connection was lost for a player. If this happens, *forfeit* will be *true*, otherwise it will be *false*.

state

Sent by server.

```
{
        {
                msg: "state",
                id: "game-id",
                start: timestamp,
                players: {
                        r: { id: user-id, color: int, moves: integer, moving: bool ,
```

b: `id: user-id, color: int, moves: integer, moving: bool` , }, board: [, `c: 1` ,

`p: int, t: str, c: 1` , `p: 'r', t: 'w', c: 1` , `p: 'b', t: 'g'` ,

-# -rbyp -wfg,

"—" "-rw" "#rg" "#bf" ...] } }}

This message describes the entire, current state of the game (minus the log). Everything in this message "could" be derived by carefully keeping track of state changes given all the other messages, but this is the server's understanding of the game state and is authoritative.