

1 Gameplay

The objective is to be the player with the most assets at the end of the game. Assets are cashed in at 50% of their initial value.

Game play is asynchronous. There are no turns, as soon as you have actions queued up, you can call `sendActions` and your actions will be dispatched to the server. A side affect of this is that the state of the client does not necessarily equal the state of the server. This means that the faster your client is, the more accurate your view of the game state will be. Additionally, if you have a multi-threaded client, it will be better performing than a fully synchronous one.

2 Map

Gameplay occurs on a 50 by 50 grid of tiles. All distance is calculated manhattan style. There are the following types of tiles:

Tile	Passible	Value
Grass	Yes	g
Desert	Yes	d
Ice	Yes	i
Jungle	Yes	j
Prarie	Yes	p
Swamp	No	s
Road	Yes	r
Water	No	w
Hill	No	h
Forest	No	f
Mountain	No	m

The format of the map contains of a header and the body of the map. The header is two integer strings separated by whitespace followed by a newline. The first integer is *height*, and the second integer is *width*.

The body of the map file contains *height* lines, each of which are *width* characters long. Each character corresponds to a tile, where the value of the character determines the type of the tile (see above). The C++ tile for example uses an enum defined as follows:

```
enum Tile
{
    Grass = 'g',
    Desert = 'd',
    Ice = 'i',
    Jungle = 'j',
    Prarie = 'p',
    Swamp = 's',
    Road = 'r',
    Water = 'w',
    Hill = 'h',
    Forest = 'f',
    Mountain = 'm'
};
```

3 Server Architecture

The server this year runs on Java. All of the necessary Jars are provided. The server round robins between the actions of the clients. When it is finished with all of the actions for a given client, it sends the client the latest state of the game. The server only does validation of data after it chooses an action for a user. This means that a user will lose out on potential actions if they are not careful to validate their actions.

Maps for testing may be chosen by running `./pickMap M` in the server directory, where `M` is a map name. Valid maps are "green, mtdoom, wateringhole, iceflows", in order of difficulty.

To run the server for testing, use `./run X Y`, where `X` is the number of clients to wait for, and `Y` is the number of seconds to run the server. The tournament matches will last 15 seconds, so this may be a good time to test with.

4 Your Client

Each client is required to contain a `run.sh` file and a `Makefile`. When your client is submitted, we first run `make`, and then run `./run.sh $SERVER $PORT $MAP`.

We provide a generic `run.sh` and `Makefile` with each of the client libraries.

In addition, all of the provided client libraries provide an `install.sh` that when run will install all of the necessary libraries in your home directory.

5 The Arena

The arena is a place where you can submit your code as often as you like. All clients submitted to the arena will be pitted against each other continuously throughout the programming period. Results are posted to <http://bender.acm.uiuc.edu>. For every round you participate in, you can see the final scores, anything you print to `STDOUT`, and a log of the game that you can download and replay with your debug visualizer.

You can submit by first committing to SVN (`svn commit`) and then running the `arena_push.sh` script.

You are not required to submit your code to the arena, but the number of games that you win in the arena determines how you are placed in the bracket.

6 General API

6.1 Object IDs

Each object in the game has a fully unique id. For example, you cannot have a treasure and a player both with ID. Object IDs are persistent and will never be reused, which makes them ideal for persistent "pointers".

6.2 Objects

Hero Each team has one hero unit that is used to pick up objects, such as treasures, powerups and plasma. The hero also marks the location where new units summon to. The hero's initial statistics are: attackDamage: 50 attackRange: 0 visibilityRange: 7 health: 1000

- `maxTreasures` : `Unsigned Integer`

The maximum number of treasures that the hero unit can carry. The `maxTreasures` = `floor(health / 250)`.

- `treasures` : `List of Treasure`

The list of the treasures that is currently being carried by the hero.

Treasure Treasure gives you a certain amount of plasma per turn as long as your hero unit has it in its possession. There are 20 treasures on the map.

- `plasmaPerTurn` : `Unsigned Integer`

The amount of plasma that is gained per turn when the treasure is in your possession. This is always between 1 and 5. 10 of the treasures give 1 per turn. 4 of the treasures give 2 per turn. 2 of the treasures give 3 per turn. 2 of the treasures give 4 per turn. 2 of the treasures give 5 per turn.

- `location` : `Location`

The location of the treasure. This will be the location of the hero if it is in the possession of a hero.

Player The player object is used as a reference to the owner of units.

- **name** : **String**
The name of the team.
- **plasma** : **Unsigned Integer**
This is the ammount of liquid asset you possess. This does not include your units as assets. Units will be cashed in for 50 percent of their initial value at the end of the game.

Unit These are the units that your hero spawns. There are a maximum of 100 units per team (including the hero). The base cost of a unit is 100. When you kill a unit you get 33 percent of that unit's cost (minus it's base and unless it's the hero). Unit's are cashed in (at 50 percent) at the end of the game to count towards your assets.

- **owner** : **Player**
- **maxHealth** : **Unsigned Integer**
The hero unit has a maximum health of 1000. For other units, the maximum health is specified at creation.
- **health** : **Unsigned Integer**
The current health that the unit has. This will always be between 0 and maxHealth.
- **visibilityRange** : **Unsigned Integer**
- **attackRange** : **Unsigned Integer**
- **attackDamage** : **Integer**
The attack damage of a particular unit.
- **location** : **Location**

6.3 Actions

Move Move a unit along the specified path. You must specify a series of single steps that each take a turn to process. For example, a path (1,1), (1,2), (2,2), (2,3), (3,3) will take five turns to process.

- **unit** : **Unit**
- **path** : **List of Location**

Drop Drop a treasure. Since your hero can only hold so many items at a time, you want to prioritize which treasures he holds. A treasure that gives you 5 plasma per turn is worth more than a treasure that only gives you 1 plasma per turn.

- **treasure** : **Treasure**

Attack Attack a unit. This command will fail if you are not within range of the unit you are attempting to attack, or if you can no longer see the unit you are attacking when the action gets processed.

- **unit** : **Unit**
- **target** : **Unit**

Summon Summon a new unit. The new unit is created at the current location of your hero unit. If you no longer have a hero unit, you will not be able to summon any more units. The costs of each attribute are specified as a formula in terms of x , where x is the desired value of the field. For example, if you wanted a unit with 20 health, x would be 20, and the cost would be 100 plasma. Additionally, there is a base cost of 100 plasma for each unit. Therefore, the minimum cost of a unit would be 100 plasma plus 5 plasma for one unit of health.

- **health** : **Unsigned Integer**
Health costs $5 * x$, where x is the desired maximum health for the new unit.

- `visibilityRange` : `Unsigned Integer`
Cost for visibility range is $2 * x^4$.
- `attackRange` : `Unsigned Integer`
Cost for attack range is $10 * x^3 + 50$.
- `attackDamage` : `Integer`
There are two formulas for attack damage, one for negative attack damage (for a medic) and one for positive attack damage. Cost for positive attack damage is $2 * x^2$. Cost for negative attack damage is $10 * x^2$.

PickUp Signal the hero to pick up a treasure. Note that your hero must be in the same location as the treasure for this command to work. Additionally, this requires your hero to have room to hold another treasure. If this is not the case, or if your hero is dead this command will fail.

- `treasure` : `Treasure`

7 C++ and Java Specific API

Although the Java and C++ clients are both different, the general architecture is the same.

7.1 Sending Actions and Getting State

There is a function `sendActions` that sends actions and then blocks on game state. This function is locked internally so that only one thread can be sending or waiting at a time. You are welcome to change this and create your own thread interleaving, however this is not necessary and could potentially result in you DOSing the server (which will disqualify you).

7.2 Queuing Actions

Actions are queued by calling the action functions in the game class. The function call queues an action that is then serialized to XML when `sendActions` is called.

7.3 Persistent Pointers to Game State

If you need to keep a pointer to a `MechMania` object (such as a player), you should not store this information as a pointer. Instead, each object has a unique ID to identify it that will be persistent from turn to turn. Store this as an unsigned int, and then restore it (as demonstrated in the C++ sample client).

8 Python Specific API

The python client runs on twisted, and therefore is event driven. All of the code goes in a turn function that is a member of `Game`. As shown in the sample code, the best way of implementing this is to create a subclass of `Game` and add the function `turn`.

9 Perl Specific API

The Perl client is fairly simple. You need to write a loop around the `send_actions` method, which returns when your actions complete and updates the internal game state. The id value returned in the data structure can be used between iterations to refer to the same thing – as long as it still exists in the game. All of the data in the turn and state formats are available, and the exact data structure returned is documented in `perldoc` (run `'perldoc'` on `MechMania.pm`).

10 Protocol

We use a protocol with three layers. At the bottom, we are using a single TCP/IP socket per client that is open for the duration of the game. If for some reason this socket closes, the client is disconnected for the rest of the game.

At the second level, we use D. J. Bernstein's NetStrings (see <http://cr.yip.to/proto/netstrings.txt>). The first message sent from the client to the server contains only the player's name.

At the third level, there are two XML formats that we use. One is used for sending the state from the server to the player, and the other is used for sending actions from the player to the server. The XML schemas for these protocols are included below.

10.1 State Protocol

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:simpleType name="location">
    <xs:restriction base="xs:string">
      <xs:pattern value="(\d+,\d+)" />
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="boolean">
    <xs:restriction base="xs:string">
      <xs:enumeration value="true" />
      <xs:enumeration value="false" />
    </xs:restriction>
  </xs:simpleType>
  <xs:complexType name="Hero">
    <xs:complexContent>
      <xs:extension base="Unit">
        <xs:sequence>
          <xs:element name="maxTreasures" type="xs:unsignedInt" />
          <xs:element name="treasures" type="xs:IDREF" maxOccurs="unbounded" />
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="Treasure">
    <xs:sequence>
      <xs:element name="plasmaPerTurn" type="xs:unsignedInt" />
      <xs:element name="location" type="location" />
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID" />
  </xs:complexType>
  <xs:complexType name="Player">
    <xs:sequence>
      <xs:element name="name" type="xs:string" />
      <xs:element name="plasma" type="xs:unsignedInt" />
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID" />
  </xs:complexType>
  <xs:complexType name="Unit">
    <xs:sequence>
```

```

    <xs:element name="owner" type="xs:IDREF" />
    <xs:element name="health" type="xs:unsignedInt" />
    <xs:element name="visibilityRange" type="xs:unsignedInt" />
    <xs:element name="attackRange" type="xs:unsignedInt" />
    <xs:element name="attackDamage" type="xs:int" />
    <xs:element name="location" type="location" />
  </xs:sequence>
  <xs:attribute name="id" type="xs:ID" />
</xs:complexType>
<xs:element name="state">
  <xs:complexType>
    <xs:sequence>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="hero" type="Hero" />
        <xs:element name="treasure" type="Treasure" />
        <xs:element name="player" type="Player" />
        <xs:element name="unit" type="Unit" />
      </xs:choice>
    </xs:sequence>
    <xs:attribute name="me" type="xs:IDREF" />
  </xs:complexType>
</xs:element>
</xs:schema>

```

10.2 Turn Protocol

```

<?xml version="1.0" encoding="ISO-8859-1"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:simpleType name="location">
    <xs:restriction base="xs:string">
      <xs:pattern value="(\d+,\d+)" />
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="boolean">
    <xs:restriction base="xs:string">
      <xs:enumeration value="true" />
      <xs:enumeration value="false" />
    </xs:restriction>
  </xs:simpleType>

  <xs:element name="turn">
    <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:sequence>
          <xs:element name="move">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="unit" type="xs:int" />
                <xs:element name="path" type="location" maxOccurs="unbounded"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>

```

```

<xs:element name="drop">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="treasure" type="xs:int" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="attack">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="unit" type="xs:int" />
      <xs:element name="target" type="xs:int" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="summon">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="health" type="xs:unsignedInt" />
      <xs:element name="visibilityRange" type="xs:unsignedInt" />
      <xs:element name="attackRange" type="xs:unsignedInt" />
      <xs:element name="attackDamage" type="xs:int" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="pickUp">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="treasure" type="xs:int" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:sequence>
</xs:choice>
</xs:complexType>
</xs:element>
</xs:schema>

```