

MechManiaXI Reference Manual

1.0

Generated by Doxygen 1.4.4

Thu Oct 6 16:01:42 2005

Contents

1	Rules and Introduction	1
1.1	The Story So Far	1
1.2	How The Game Is Played	1
1.3	Rules of the Contest	3
1.4	Running a Test Match	4
1.5	XML Files	4
1.6	API Introduction	5
2	MechManiaXI Module Index	7
2.1	MechManiaXI Modules	7
3	MechManiaXI Hierarchical Index	9
3.1	MechManiaXI Class Hierarchy	9
4	MechManiaXI Class Index	11
4.1	MechManiaXI Class List	11
5	MechManiaXI Module Documentation	13
5.1	Client Library	13
6	MechManiaXI Class Documentation	15
6.1	mmxi::Archer Class Reference	15
6.2	mmxi::Fighter Class Reference	17
6.3	mmxi::GameBoard Class Reference	19
6.4	mmxi::GameConstants Struct Reference	23
6.5	mmxi::Incoming Class Reference	26
6.6	mmxi::MoveableUnit Class Reference	28
6.7	mmxi::Player Class Reference	30
6.8	mmxi::Point Class Reference	33
6.9	mmxi::Unit Class Reference	36

6.10 mmxi::Wall Class Reference 44

6.11 mmxi::Wizard Class Reference 45

Chapter 1

Rules and Introduction

1.1 The Story So Far

Since ancient times, man has waged war with each other in pursuit of land, resources, and superiority over fellow man. Swords and magic are commonplace and with each passing year comes sharper swords and more potent magic. Fighters, archers, and wizards lay siege to enemy castles where citizens hide in fear. The entire world was engulfed in constant warfare.

In an effort to put an end to the wars, King Kamin, ruler of the most prosperous country in the land, commissioned his subjects to create a weapon of terrible destruction. The weapon was to be named "Mech", the ancient word for peace. It was the king's intent to use the weapon to force the other kingdoms towards a period of peace and prosperity. However, peace always comes at great cost.

The king was betrayed by his subjects and the plans for the weapon were leaked to enemy lands. Before King Kamin was able to put an end to the fighting, his enemies built mechs of their own. However, they did not intend to use mechs for the benefit of mankind. The weapon intended to bring peace instead brought death, famine and destruction across the land. Cities were set a flame by the terrible power of the mech. Farmland was razed and kingdoms crumbled. Each passing year brought more destruction.

Over ten years have passed since the start of the war. Mechs are no longer in production. Armies have been diminished and war funds are being used to fund the rebuilding of kingdoms. However, the war is not yet over!

Reconnaissance spies have spotted an abandoned mech on the outskirts of your kingdom while surveying an enemy kingdom. The king has ordered your troops to capture the abandoned mech and sneak it in to the enemy's castle. Assault the enemy castle and use the mech to burn your way to victory! Peace is finally at hand, but who will be the one to restore it?

1.2 How The Game Is Played

1.2.1 Objective

At the center of the playing field, there is a mech.

The first team to carry the Mech into the opposing team's castle wins the match. In order to do this, each team has a limited amount of resources with which to build units and walls. Every turn each team receives money, used to purchase units, which appear at their castle. Also, money can

be used by Archer units to construct walls.

1.2.2 Scoring

If no one completes the objective after 4500 turns, a scoring system is used to determine a winner. Scoring is based on kills and on castle proximity. Killing an enemy a certain distance from his castle will earn a minimal number of points. Killing an enemy close to his castle will earn a considerable number of points.

The specific score is determined by the following code, where `this_team` is the team of the newly dead unit and `other_team` is the team scoring points:

```
int castle_dist = abs(dead_unit.x - castles[this_team].x) + abs(dead_unit.y - castles[this_team].y);

int score_multiplier = (max_score_dist - castle_dist)/score_scale;
if(score_multiplier < min_score) score_multiplier = min_score;

score[other_team] += dead_unit.base_score() * score_multiplier;
```

If, at 4500 turns, the score is even, the game will enter sudden death. The instant the score is no longer even, a winner will be declared.

If, at 5500 turns, the score is still even, the winner will be determined randomly.

1.2.3 Units

1.2.3.1 Fighters

Fighters are the standard melee combatant. They can only attack adjacent units. Fighters are able to escape the radius of the Wizard's fireball attack, but take too much damage while approaching Archers.

Fighters are also the only unit which can carry the Mech. They will automatically pick up the Mech if they walk over it.

1.2.3.2 Archers

Archers are a ranged combatant with a fast rate of fire. Their arrows fire with medium range, but fire fast enough to deal significant damage to approaching Fighters. Wizards, however, have a longer range than Archers, and are therefore very dangerous to Archers.

Archers can build walls, which can only be destroyed by Wizards.

1.2.3.3 Wizards

Wizards are an artillery ranged attacker. They cast fireballs which are slow, but have a nine square area of effect and deal massive damage. Fighters are fast enough to avoid fireballs, but Archers have shorter range than Wizards and are unable to escape the blast.

Wizards are the only unit able to destroy walls.

1.2.4 Walls

Walls block all movement and can only be damaged by Wizards. A unit standing on a wall cannot be damaged by Fighters. There is no damage taken for moving off of a wall. However, there is no way to climb walls again after leaving them. An Archer who builds a wall ends up standing on it.

1.3 Rules of the Contest

1.3.1 Basic Rules

The intention of the contest is for groups of three to independently develop competitive programs using the resources provided. Laptop computers may be used to view the online documentation, but any code written on a computer not provided by us is unsupported. There is to be no food or drink inside the lab. Outside libraries may be used, as long as your submission builds and runs on the lab computers.

Any attempt to circumvent the rules of this contest by receiving outside help, gaining unauthorized access to the data of other teams, or any other unsportsmanlike conduct will be grounds for disqualification. All decisions made by tournament judges are final.

1.3.2 Time Limits

The programming session will run twenty-four hours starting from 9 AM Saturday, October 8 and ending 9 PM Sunday, October 9. During this time only 18 hours of lab time may be used to work on an entry. The remaining time must be spent outside of the lab.

The lab will close for one hour during the keynote presentation, and thus teams have five hours outside of the lab to allocate however they desire. The purpose for this rule is to allow commuting teams the opportunity to sleep and all teams a chance to attend the conference. Teams will be ejected when they have exceeded their allowed time in the lab. However, a grace period may be allowed for final submission.

1.3.3 Tournament Format

The format of the tournament will be double-elimination. That is, any team who loses two games will be eliminated. The winning team will be the last team without two losses. The seeding for the tournament will be determined randomly. The tournament will commence at 11:30 AM in 1320 DCL on Sunday, October 9th.

Once your client is started, you will have 20 seconds to connect successfully to the server. If you fail to establish a connection in this time, you will lose the match. If neither client connects, the server will flip a coin to decide the winner.

If your client disconnects from the server at any time, for any reason, you will automatically lose the match.

The server timeout is three seconds. If your client does not send commands for a given turn in three seconds, it will be disconnected.

The judges reserve the right to kill -9 any client, at any time, for any reason. Furthermore, if the judges determine that your code is crashing the server, you will be disqualified from the tournament.

Finally, if a game runs for 500 turns (approx. 20 seconds) with neither client taking any action, the winner will be determined randomly.

1.3.4 Contest Arena

During the contest there will be a server running battles between random team submissions, which is referred to as the Arena. The Arena exists to allow teams to test their strategies and further encourage interesting competition. It is strongly recommended that teams submit work-in-progress player code to the arena, but it is not mandatory. A small prize will be awarded to the team with the best record in the Arena. The Siebel video wall will display arena battles, and teams will be able to connect to the server running the Arena with a visualization client.

1.4 Running a Test Match

Running a test match can be done in one of two ways. First, the long way.

Start up a server.

```
mmxiserv [mapfile]
```

Open a visualization client.

```
xvis
```

Start up Player 1.

```
myclient
```

Start up Player 2.

```
someotherclient
```

However, if you prefer to save keystrokes (and terminal windows) the server is able to do any or all of this for you.

```
mmxiserv [mapfile] --xvis --p1 myclient --p2 someotherclient
```

1.5 XML Files

In order to associate your team with the PNG files for your flag and logo, the server reads in an XML file. This file must be named `team#.xml` (where `#` is your team number.) It will read from the XML file your team name, team flag, team logo, and team color for use with the 3D visualization software.

Here is an example XML file. The format should be self-explanatory:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<mm1team>
<teamnumber>0</teamnumber>
<teamname>The Test Client</teamname>
```

```
<executable>testclient</executable>
<teamlogo>testlogo.png</teamlogo>
<teamflag>testflag.png</teamflag>
<primary_color>#FFF00</primary_color>
<secondary_color>#800080</secondary_color>
</mm11team>
```

1.6 API Introduction

In order to create a client, you must create a derived version of the **mmxi::Player**(p. 30) class. **mmxi::Player**(p. 30), which is derived from **mmxi::GameBoard**(p. 19), contains all the knowledge of the current game state, command functions like **mmxi::Player::build_fighter()**(p. 30), **mmxi::Player::build_archer()**(p. 30), and **mmxi::Player::build_wizard()**(p. 30), and many helper functions inherited from **mmxi::GameBoard**(p. 19) to help you make sense of the raw state data.

To examine units, and order your own units, you'll interact with the **mmxi::Unit**(p. 36) class hierarchy - in particular, **mmxi::Fighter**(p. 17), **mmxi::Wizard**(p. 45), and **mmxi::Archer**(p. 15). Those classes have information like their current position (in the form of an **mmxi::Point**(p. 33), which is a handy helper class to deal with locations) and command functions like **mmxi::Unit::attack()**(p. 39) or **mmxi::Archer::build_wall()**(p. 15).

Once you have your **mmxi::Player**(p. 30) class, you'll instantiate it and pass it to **mmxi::MMXIMain()**(p. 14), along with your team number, your team name, and a five-integer hash code that uniquely identifies your team. This allows the server to associate you with your XML file. **mmxi::MMXIMain()**(p. 14) will then parse the command-line options, connect to the server, identify itself, and initiate the game.

Chapter 2

MechManiaXI Module Index

2.1 MechManiaXI Modules

Here is a list of all modules:

Client Library	13
--------------------------	----

Chapter 3

MechManiaXI Hierarchical Index

3.1 MechManiaXI Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

mmxi::GameBoard	19
mmxi::Player	30
mmxi::GameConstants	23
mmxi::Point	33
mmxi::Unit	36
mmxi::Incoming	26
mmxi::MoveableUnit	28
mmxi::Archer	15
mmxi::Fighter	17
mmxi::Wizard	45
mmxi::Wall	44

Chapter 4

MechManiaXI Class Index

4.1 MechManiaXI Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

mmxi::Archer (Class representing an Archer (p.15) unit)	15
mmxi::Fighter (Class representing a Fighter (p.17) unit)	17
mmxi::GameBoard (A class used to track our current knowledge of the game board)	19
mmxi::GameConstants (Contains all the numerical constants (the "rules") of a game)	23
mmxi::Incoming (Class representing an Incoming! message)	26
mmxi::MoveableUnit (This means Fighters, Archers, and Wizards (the Big Three.))	28
mmxi::Player (The superclass for all game clients)	30
mmxi::Point (Represents a point on the game board)	33
mmxi::Unit (Superclass for the Unit (p.36) hierarchy)	36
mmxi::Wall (Class representing a Wall (p.44))	44
mmxi::Wizard (Class representing a Wizard (p.45) unit)	45

Chapter 5

MechManiaXI Module Documentation

5.1 Client Library

Classes

- class **mmxi::Point**
Represents a point on the game board.
- class **mmxi::Unit**
*Superclass for the **Unit**(p. 36) hierarchy.*
- class **mmxi::MoveableUnit**
*This means **Fighters**, **Archers**, and **Wizards** (the Big Three.).*
- class **mmxi::Fighter**
*Class representing a **Fighter**(p. 17) unit.*
- class **mmxi::Archer**
*Class representing an **Archer**(p. 15) unit.*
- class **mmxi::Wizard**
*Class representing a **Wizard**(p. 45) unit.*
- class **mmxi::Wall**
*Class representing a **Wall**(p. 44).*
- class **mmxi::Incoming**
Class representing an Incoming! message.
- class **mmxi::GameBoard**
A class used to track our current knowledge of the game board.
- class **mmxi::GameBoard**

A class used to track our current knowledge of the game board.

- class **mmxi::Player**

The superclass for all game clients.

Functions

- void **mmxi::SendOrder** (const COrder &ord)

Queues up an Order to send to the server.

- template<class T> const Unit::TTalker & **mmxi::operator**<< (const Unit::TTalker &lhs, const T &rhs)

- int **MMXIMain** (int argc, char *argv[], **mmxi::Player** *thePlayer, unsigned int teamnum, const char *teamname, unsigned int hash0, unsigned int hash1, unsigned int hash2, unsigned int hash3, unsigned int hash4)

Starts up the client.

5.1.1 Function Documentation

5.1.1.1 int MMXIMain (int argc, char * argv[], mmxi::Player * thePlayer, unsigned int teamnum, const char * teamname, unsigned int hash0, unsigned int hash1, unsigned int hash2, unsigned int hash3, unsigned int hash4)

Starts up the client.

MMXIMain(p. 14) will connect to the server, join any games that might be going on, keep the Player's game board updated, call thePlayer->Turn() once per turn, and send thePlayer's Orders to the server.

5.1.1.2 void mmxi::SendOrder (const COrder & ord)

Queues up an Order to send to the server.

The Order passed to **SendOrder**(p. 14) is placed in a buffer, which is sent to the server once Turn() exits.

Chapter 6

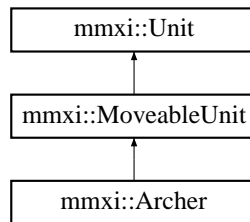
MechManiaXI Class Documentation

6.1 mmxi::Archer Class Reference

Class representing an **Archer**(p. 15) unit.

```
#include <board.h>
```

Inheritance diagram for mmxi::Archer::



Public Member Functions

- **Archer** (CUnit unit)
Constructor.
- virtual **Archer * get_archer** ()
Downcast to an Archer(p. 15).
- virtual float **range** ()
Returns this Unit's attack range.
- virtual void **build_wall** ()
Only works if this Unit(p. 36) is an Archer(p. 15).

6.1.1 Detailed Description

Class representing an **Archer**(p. 15) unit.

Archers have a low-powered, but fast, ranged attack. They are particularly effective against Fighters. They also have the ability to build walls.

6.1.2 Member Function Documentation

6.1.2.1 virtual Archer* mmxi::Archer::get_archer () [inline, virtual]

Downcast to an **Archer**(p. 15).

So far, client code has not been tested for compatibility with RTTI. We are providing these simple downcast functions for convenience. They will return NULL if the cast is no good, and this otherwise.

Reimplemented from **mmxi::Unit** (p. 40).

6.1.2.2 virtual float mmxi::Archer::range () [inline, virtual]

Returns this Unit's attack range.

If the distance to some enemy unit is less than this range, it is likely that an attack will succeed - although, remember, the targeted unit can always move to somewhere where your attack will not succeed. This is particularly a problem for Wizards.

Returns 0 if called on a Castle or a **Wall**(p. 44).

Reimplemented from **mmxi::Unit** (p. 42).

The documentation for this class was generated from the following file:

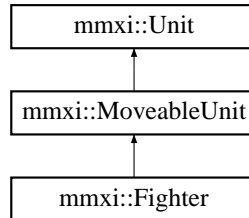
- board.h

6.2 mmxi::Fighter Class Reference

Class representing a **Fighter**(p.17) unit.

```
#include <board.h>
```

Inheritance diagram for mmxi::Fighter::



Public Member Functions

- **Fighter** (CUnit &unit)
Constructor.
- virtual **Fighter** * **get_fighter** ()
*Downcast to a **Fighter**(p.17).*
- virtual float **range** ()
Returns this Unit's attack range.

6.2.1 Detailed Description

Class representing a **Fighter**(p.17) unit.

Fighters have a moderately powerful attack that can be used against adjacent squares. They also have the ability to carry the Mech, which they automatically pick up by walking over it.

6.2.2 Member Function Documentation

6.2.2.1 virtual **Fighter*** mmxi::Fighter::get_fighter () [inline, virtual]

Downcast to a **Fighter**(p.17).

So far, client code has not been tested for compatibility with RTTI. We are providing these simple downcast functions for convenience. They will return NULL if the cast is no good, and this otherwise.

Reimplemented from **mmxi::Unit** (p.40).

6.2.2.2 virtual float mmxi::Fighter::range () [inline, virtual]

Returns this Unit's attack range.

If the distance to some enemy unit is less than this range, it is likely that an attack will succeed - although, remember, the targeted unit can always move to somewhere where your attack will not succeed. This is particularly a problem for Wizards.

Returns 0 if called on a Castle or a **Wall**(p. 44).

Reimplemented from **mmxi::Unit** (p. 42).

The documentation for this class was generated from the following file:

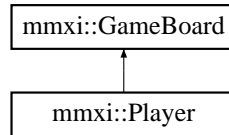
- board.h

6.3 mmxi::GameBoard Class Reference

A class used to track our current knowledge of the game board.

```
#include <board.h>
```

Inheritance diagram for mmxi::GameBoard::



Public Member Functions

- **bool building ()**
Are we in fact building anything?
- **template<class T, class U> T closest_from (U st, const std::vector< T > &ed)**
Templated generic closest-thing finder. Called by all the others.
- **Unit * closest_friendly (Unit *in)**
Returns the closest friendly unit to in.
- **Unit * closest_enemy (Unit *in)**
Returns the closest enemy unit to in.
- **Wall * closest_wall (Unit *in)**
Returns the closest wall to in.
- **Wall * closest_friendly_wall (Unit *in)**
Returns the closest friendly-built wall to in.
- **Wall * closest_enemy_wall (Unit *in)**
Returns the closest enemy-built wall to in.
- **Wall * closest_neutral_wall (Unit *in)**
Returns the closest map wall to in.
- **Fighter * closest_friendly_fighter (Unit *in)**
Returns the closest friendly fighter to in.
- **Archer * closest_friendly_archer (Unit *in)**
Returns the closest friendly archer to in.
- **Wizard * closest_friendly_wizard (Unit *in)**
Returns the closest friendly wizard to in.
- **Fighter * closest_enemy_fighter (Unit *in)**

Returns the closest enemy fighter to in.

- **Archer * closest_enemy_archer (Unit *in)**
Returns the closest enemy archer to in.
- **Wizard * closest_enemy_wizard (Unit *in)**
Returns the closest enemy wizard to in.
- **bool is_unit (int x, int y)**
Returns true if there's a unit at the given location.
- **bool is_unit (const Point &p)**
Returns true if there's a unit at the given location.
- **bool is_wall (int x, int y)**
Returns true if there's a wall at the given location.
- **bool is_wall (const Point &p)**
Returns true if there's a wall at the given location.
- **bool is_wall (const Unit *un)**
Returns true if there's a wall at the given Unit's location.

Public Attributes

- **std::vector< Unit * > all**
All units.
- **std::vector< Unit * > friendly**
All friendly, moveable units.
- **std::vector< Unit * > enemy**
All enemy, moveable units. (Anything that might attack you.).
- **std::vector< Unit * > neutral**
All neutral units. (Currently, the Mech.).
- **std::vector< Fighter * > my_fighters**
All friendly (that is, controllable) fighters.
- **std::vector< Archer * > my_archers**
All friendly (that is, controllable) archers.
- **std::vector< Wizard * > my_wizards**
All friendly (that is, controllable) wizards.
- **std::vector< Fighter * > enemy_fighters**
All enemy fighters.

- `std::vector< Archer * > enemy_archers`
All enemy archers.
- `std::vector< Wizard * > enemy_wizards`
All enemy wizards.
- `std::vector< Wall * > walls`
All walls on the game board.
- `std::vector< Wall * > my_walls`
All walls built by you.
- `std::vector< Wall * > enemy_walls`
All walls built by someone not you.
- `std::vector< Wall * > neutral_walls`
All walls that the game started out with.
- `std::vector< Incoming * > incoming`
Incoming(p. 26) attacks/explosions you might want to respond to.
- `std::vector< Unit * > my_new_dead`
*Units of yours that died *this turn.*.*
- `std::vector< Unit * > enemy_new_dead`
*Enemy units that died *this turn.*.*
- `std::vector< Unit * > dead`
Dead units go here.
- `Unit * castles [2]`
Your castle is castles[0]. The enemy castle is castles[1].
- `Unit * mech`
The all-important Mech.
- `int turn`
The current turn.
- `int money`
How much money we have.
- `int score [2]`
The current scores. Your score is score[0]. Your opponent's score is score[1].
- `COrder constructing`
What we're building right now.

6.3.1 Detailed Description

A class used to track our current knowledge of the game board.

The documentation for this class was generated from the following file:

- board.h

6.4 mmxi::GameConstants Struct Reference

Contains all the numerical constants (the "rules") of a game.

```
#include <constants.h>
```

Public Member Functions

- void **set_type** (GameType type)

Public Attributes

- int **fighter_hp**
Hit points of Fighters.
- int **archer_hp**
Hit points of Archers.
- int **wizard_hp**
Hit points of Wizardss.
- double **fighter_speed**
Movement speed of Fighters. Note that lower speeds are faster.
- double **archer_speed**
Movement speed of Archers. Note that lower speeds are faster.
- double **wizard_speed**
Movement speed of Wizards. Note that lower speeds are faster.
- int **starting_money**
How much money each team starts with.
- int **move_time**
Basic cost for moving horizontally or vertically.
- int **diag_move_time**
Basic cost for moving diagonally.
- int **fighter_cost**
*Cost of building a **Fighter**(p. 17).*
- int **archer_cost**
*Cost of building an **Archer**(p. 15).*
- int **wizard_cost**
*Cost of building a **Wizard**(p. 45).*
- int **wall_hp**
Hit points of Walls.

- int **wall_cost**
Cost of building a wall.
- int **wall_build_time**
Time it takes to build a wall.
- int **fighter_build_time**
*Time it takes to build a **Fighter**(p. 17).*
- int **wizard_build_time**
*Time it takes to build a **Wizard**(p. 45).*
- int **archer_build_time**
*Time it takes to build an **Archer**(p. 15).*
- double **fighter_range**
*Attack range of **Fighters**.*
- double **archer_range**
*Attack range of **Archers**.*
- double **wizard_range**
*Attack range of **Wizards**.*
- int **fighter_charge**
*Time it takes a **Fighter**(p. 17) to attack.*
- int **archer_charge**
*Time it takes an **Archer**(p. 15) to attack.*
- int **wizard_charge**
*Time it takes a **Wizard**(p. 45) to attack.*
- int **fighter_damage**
*How much damage a **Fighter**(p. 17) does.*
- int **archer_damage**
*How much damage an **Archer**(p. 15) does.*
- int **wizard_damage**
*How much damage a **Wizard**(p. 45) does to the square it hits directly.*
- int **wizard_splash_damage**
*How much damage a **Wizard**(p. 45) does to horizontally or vertically adjacent squares.*
- int **wizard_outer_splash_damage**
*How much damage a **Wizard**(p. 45) does to diagonally adjacent squares.*
- `GameType` **current_game_type**
Enum describing the current game type.

6.4.1 Detailed Description

Contains all the numerical constants (the "rules") of a game.

The documentation for this struct was generated from the following file:

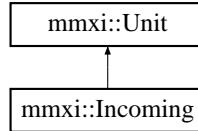
- constants.h

6.5 mmxi::Incoming Class Reference

Class representing an Incoming! message.

```
#include <board.h>
```

Inheritance diagram for mmxi::Incoming::



Public Member Functions

- **Incoming** (CUnit unit)
Constructor.
- virtual **Incoming** * **get_incoming** ()
Downcast to an Incoming(p. 26).
- virtual bool **in_range** (const **Unit** *target)
Checks range to some Unit(p. 36).

6.5.1 Detailed Description

Class representing an Incoming! message.

Incoming! messages are subclasses of **Unit**(p. 36) mostly for bookkeeping convenience. They appear only in the incoming roster, nowhere else, and are generated when a **Wizard**(p. 45) starts spellcasting. (They may also be generated by some other events. Depends on the map.)

In any case, if you have a **Unit**(p. 36) within range of an **Incoming**(p. 26), you may just want to get him the hell out of there.

6.5.2 Member Function Documentation

6.5.2.1 virtual Incoming* mmxi::Incoming::get_incoming () [inline, virtual]

Downcast to an **Incoming**(p. 26).

So far, client code has not been tested for compatibility with RTTI. We are providing these simple downcast functions for convenience. They will return NULL if the cast is no good, and this otherwise.

Reimplemented from **mmxi::Unit** (p. 40).

6.5.2.2 virtual bool mmxi::Incoming::in_range (const Unit * target) [inline, virtual]

Checks range to some **Unit**(p. 36).

If `in_range` returns true, you may assume that calling `attack()` (p. 39) on the same `Unit` (p. 36) will succeed. (Unless, of course, that `Unit` (p. 36) moves out of range or does something else nefarious.

Reimplemented from `mmxi::Unit` (p. 41).

The documentation for this class was generated from the following file:

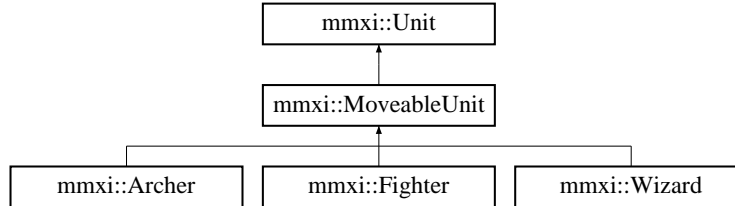
- `board.h`

6.6 mmxi::MoveableUnit Class Reference

This means Fighters, Archers, and Wizards (the Big Three.).

```
#include <board.h>
```

Inheritance diagram for mmxi::MoveableUnit::



Public Member Functions

- **MoveableUnit** (CUnit &unit)
Constructor.
- virtual void **move** (const **Point** &destination)
Begin moving to a new square.
- virtual void **move** (int xdest, int ydest)
Begin moving toward an x,y point.
- virtual void **attack** (const **Unit** *target)
Attack a unit.
- virtual void **dance** ()
Dance.

6.6.1 Detailed Description

This means Fighters, Archers, and Wizards (the Big Three.).

6.6.2 Member Function Documentation

6.6.2.1 virtual void mmxi::MoveableUnit::attack (const Unit * target) [virtual]

Attack a unit.

Attacks all have charge times, which will be arbitrated by the server. If you attempt to attack a unit not in range, or not alive, or nonexistent (the ID you send does not correspond to any units on the board) the server will generate a warning.

If a target unit moves out of range during charge time, the attack will fail.

Castles and Walls cannot attack, and attempting to attack with one will cause your client to generate a warning.

Reimplemented from `mmxi::Unit` (p. 39).

6.6.2.2 virtual void mmxi::MoveableUnit::dance () [virtual]

Dance.

Yes, dance. The `Unit`(p. 36) will boogie down. (Note that `idle()`(p. 41) still returns true if the `Unit`(p. 36) is dancing; therefore, dancing will probably not foul up your AI.

Reimplemented from `mmxi::Unit` (p. 39).

6.6.2.3 virtual void mmxi::MoveableUnit::move (int *xdest*, int *ydest*) [virtual]

Begin moving toward an x,y point.

Not recommended; provided for convenience. The `Point`(p. 33) class is generally superior to x,y coordinate pairs, and is recommended for most purposes (read: all that do not involve mucking around in the protocol, which is also not recommended).

Reimplemented from `mmxi::Unit` (p. 42).

6.6.2.4 virtual void mmxi::MoveableUnit::move (const Point & *destination*) [virtual]

Begin moving to a new square.

Units may only be moved to one of the eight adjacent squares.

The time it takes to move depends on the type of unit and will be arbitrated by the server. If the server determines that your move is valid, on the next turn you may examine this unit's Order to determine how long the move will take.

We will add other helper functions for determining move cost later.

Attempting to move a Castle or a `Wall`(p. 44) is meaningless, and will cause your client to generate a warning.

Attempting to move a dead unit or a unit not yours is a non-fatal error, and will cause the server to generate a warning.

Reimplemented from `mmxi::Unit` (p. 42).

The documentation for this class was generated from the following file:

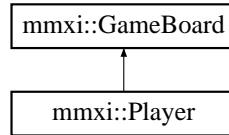
- board.h

6.7 mmxi::Player Class Reference

The superclass for all game clients.

```
#include <player.h>
```

Inheritance diagram for mmxi::Player::



Public Member Functions

- **COrder MoveOrder** (const CUnit *unit, int x, int y)
Generates and fills out a movement Order.
- void **build** (UnitType type)
 Cancels the previous build (returning your money) if it exists.
- void **build_fighter** ()
 Builds a fighter.
- void **build_archer** ()
 Builds an archer.
- void **build_wizard** ()
 Builds a wizard.
- virtual void **Turn** ()=0
 The core decision-making function, called once per turn.
- virtual **Fighter * MakeFriendlyFighter** (CUnit &in)
 Factory functions, used to allow subclassed units.
- virtual **Archer * MakeFriendlyArcher** (CUnit &in)
 Factory functions, used to allow subclassed units.
- virtual **Wizard * MakeFriendlyWizard** (CUnit &in)
 Factory functions, used to allow subclassed units.
- virtual **Fighter * MakeEnemyFighter** (CUnit &in)
 Factory functions, used to allow subclassed units.
- virtual **Archer * MakeEnemyArcher** (CUnit &in)
 Factory functions, used to allow subclassed units.
- virtual **Wizard * MakeEnemyWizard** (CUnit &in)
 Factory functions, used to allow subclassed units.

6.7.1 Detailed Description

The superclass for all game clients.

A new client is made by inheriting from this class and implementing the Turn function. Each turn, the clientlib will communicate with the server and update the **GameBoard**(p. 19) member variable. It will then call the Turn function.

6.7.2 Member Function Documentation

6.7.2.1 virtual Archer* mmxi::Player::MakeEnemyArcher (CUnit & in) [inline, virtual]

Factory functions, used to allow subclassed units.

This function is called whenever a new **Archer**(p. 15) on the enemy team is created. If you would like to write your client by subclassing the **Unit**(p. 36) classes and having them maintain state, you may override **MakeEnemyArcher**()(p. 31).

6.7.2.2 virtual Fighter* mmxi::Player::MakeEnemyFighter (CUnit & in) [inline, virtual]

Factory functions, used to allow subclassed units.

This function is called whenever a new **Fighter**(p. 17) on the enemy team is created. If you would like to write your client by subclassing the **Unit**(p. 36) classes and having them maintain state, you may override **MakeEnemyFighter**()(p. 31).

6.7.2.3 virtual Wizard* mmxi::Player::MakeEnemyWizard (CUnit & in) [inline, virtual]

Factory functions, used to allow subclassed units.

This function is called whenever a new **Wizard**(p. 45) on the enemy team is created. If you would like to write your client by subclassing the **Unit**(p. 36) classes and having them maintain state, you may override **MakeEnemyWizard**()(p. 31).

6.7.2.4 virtual Archer* mmxi::Player::MakeFriendlyArcher (CUnit & in) [inline, virtual]

Factory functions, used to allow subclassed units.

This function is called whenever a new **Archer**(p. 15) on your team is created. If you would like to write your client by subclassing the **Unit**(p. 36) classes and having them maintain state, you may override **MakeFriendlyArcher**()(p. 31).

6.7.2.5 virtual Fighter* mmxi::Player::MakeFriendlyFighter (CUnit & in) [inline, virtual]

Factory functions, used to allow subclassed units.

This function is called whenever a new **Fighter**(p. 17) on your team is created. If you would like to write your client by subclassing the **Unit**(p. 36) classes and having them maintain state, you may override **MakeFriendlyFighter**()(p. 31).

6.7.2.6 virtual Wizard* mmxi::Player::MakeFriendlyWizard (CUnit & in)
[inline, virtual]

Factory functions, used to allow subclassed units.

This function is called whenever a new **Wizard**(p. 45) on your team is created. If you would like to write your client by subclassing the **Unit**(p. 36) classes and having them maintain state, you may override **MakeFriendlyWizard**()(p. 32).

6.7.2.7 virtual void mmxi::Player::Turn () [pure virtual]

The core decision-making function, called once per turn.

This function will be called once per turn. It should examine the **GameBoard**(p. 19), decide on its moves, and use **Send()** to issue its decisions in the form of **Orders** to its units.

The documentation for this class was generated from the following file:

- player.h

6.8 mmxi::Point Class Reference

Represents a point on the game board.

```
#include <board.h>
```

Public Member Functions

- **Point** ()
Creates a point at the center of the board.
- **Point** (int xin, int yin)
*Creates a **Point**(p. 33) at (x,y).*
- int **get_x** () const
Returns the x-value.
- int **get_y** () const
Returns the y-value.
- bool **in_board** ()
*Returns true if this **Point**(p. 33) is within the playing field.*
- int **manhat_dist** (const **Point** &pt)
Returns the Manhattan distance to another point.
- float **distsquared** (const **Point** &pt)
Returns the distance to another point squared.
- float **distance** (const **Point** &pt)
Returns the distance to another point.
- **Point** **closest_point_to** (const **Point** &dest)
Returns the neighbor closest to another point.
- **Point** **farthest_point_from** (const **Point** &dest)
Returns the neighbor farthest from another point.
- **Point** **neighbor** (int n)
*Returns the numbered neighbor of this **Point**(p. 33).*
- const bool **operator==** (const **Point** &rhs)
Compare points.
- const bool **operator!=** (const **Point** &rhs)
Compare points.
- const **Point** & **operator+=** (const **Point** &rhs)
Add points.

- **const Point & operator-=** (const **Point** &rhs)
Subtract points.
- **const Point operator+** (const **Point** &rhs)
Add points.
- **const Point operator-** (const **Point** &rhs)
Subtract points.
- **int move_cost** (const **Point** &rhs)
*-1 if that point is not adjacent to this **Point**(p. 33).*

6.8.1 Detailed Description

Represents a point on the game board.

Has working assignment and copy. Can be assigned, copied, passed, spindled, and mutilated as desired.

6.8.2 Member Function Documentation

6.8.2.1 Point mmxi::Point::closest_point_to (const Point & dest)

Returns the neighbor closest to another point.

Observe **Unit::move_toward()**(p. 37) for a sample use of **closest_point_to()**(p. 34). Returns the point that is closest to the point you pass in while still being only one square from this one (and thus still a valid move.)

6.8.2.2 float mmxi::Point::distance (const Point & pt) [inline]

Returns the distance to another point.

Despite what I said in **distsquared()**(p. 34), not particularly inefficient. Avoid 2^n or $n!$ algorithms and you'll probably be fine.

6.8.2.3 float mmxi::Point::distsquared (const Point & pt) [inline]

Returns the distance to another point squared.

More efficient than **distance()**(p. 34) - if you only need to compare two distances, use **distsquared()**(p. 34) for efficiency.

6.8.2.4 Point mmxi::Point::farthest_point_from (const Point & dest)

Returns the neighbor farthest from another point.

Similar to **closest_point_to()**(p. 34), except it performs the opposite function.

6.8.2.5 `int mmxi::Point::manhat_dist (const Point & pt)` [inline]

Returns the Manhattan distance to another point.

Handy when you want to find out how far away in walking terms a point is.

6.8.2.6 `Point mmxi::Point::neighbor (int n)`

Returns the numbered neighbor of this **Point**(p.33).

A square on a grid has eight neighbors. MMXI numbers them as follows:

7	0	1
6		2
5	4	3

This function will return one of those neighbors, if called with a value from 0 to 7. This allows you to iterate through the neighbors of a **Point**(p.33) easily.

If called with a value outside the range 0-7, **neighbor()**(p.35) returns this - i.e. the center point.

The documentation for this class was generated from the following file:

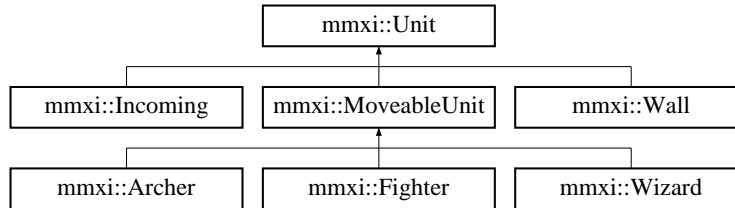
- board.h

6.9 mmxi::Unit Class Reference

Superclass for the **Unit**(p. 36) hierarchy.

```
#include <board.h>
```

Inheritance diagram for mmxi::Unit::



Public Member Functions

- **Unit** (CUnit &unit)
Constructor.
- int **get_hp** () const
*Returns the hit points of this **Unit**(p. 36).*
- int **get_id** () const
Returns this Unit's unique ID number.
- int **get_team** () const
Returns this Unit's team ID.
- bool **is_alive** () const
*Returns true if this **Unit**(p. 36) is living.*
- UnitType **get_type** () const
*Returns the type of this **Unit**(p. 36), in the form of a UnitType enum.*
- bool **idle** () const
Returns true if unit is not doing anything.
- int **eta** () const
Returns the number of turns until completion of the Unit's current action.
- virtual **Fighter** * **get_fighter** ()
*Downcast to a **Fighter**(p. 17).*
- virtual **Archer** * **get_archer** ()
*Downcast to an **Archer**(p. 15).*
- virtual **Wizard** * **get_wizard** ()
*Downcast to a **Wizard**(p. 45).*

- virtual **Incoming** * **get_incoming** ()
Downcast to an Incoming(p. 26).
- virtual **Wall** * **get_wall** ()
Downcast to a Wall(p. 44).
- virtual void **move** (const **Point** &destination)
Begin moving to a new square.
- virtual void **move** (int xdest, int ydest)
Begin moving toward an x,y point.
- void **move_direction** (int direction)
Move in one of the eight numbered directions.
- void **move_toward** (const **Point** &target)
Move toward a point on the map.
- void **move_toward** (const **Unit** *target)
Move toward some other unit.
- void **move_away_from** (const **Point** &target)
Move away from a point on the map.
- void **move_away_from** (const **Unit** *target)
Move away from some other unit.
- virtual void **attack** (const **Unit** *target)
Attack a unit.
- virtual void **attack** (const **Point** &target)
Attack a location.
- virtual void **build_wall** ()
Only works if this Unit(p. 36) is an Archer(p. 15).
- virtual bool **in_range** (const **Unit** *target)
Checks range to some Unit(p. 36).
- **Point** **get_loc** () const
Gets this Unit's location as a Point(p. 33) structure.
- virtual float **range** ()
Returns this Unit's attack range.
- bool **has_mech** ()
You may want to pay attention to this.
- virtual void **dance** ()

Dance.

- void **trashtalk** (const char *)
Trashtalk.
- TTalker **trashtalk** ()
- int **get_x** () const
Returns the current x-position of this Unit(p. 36).
- int **get_y** () const
Returns the current y-position of this Unit(p. 36).
- CUnit & **get_cunit** ()
Returns a reference to the underlying CUnit structure.
- COrder & **get_orders** ()
Returns a reference to the unit's current COrder.

Public Attributes

- int **info** [8]
Space provided for user-defined data.
- void * **infop**
Space provided for user-defined data.

Protected Attributes

- CUnit **un**
The underlying CUnit structure received from the server.

Classes

- class **TTalker**

6.9.1 Detailed Description

Superclass for the **Unit**(p. 36) hierarchy.

Unit(p. 36) is the principal data structure you will be examining to decide what to do. Units will be created and maintained for you by the client library.

Do not create your own Units; they will not represent meaningful data. If you would like to hold on to a **Unit**(p. 36), simply copy the pointer. Units are never deleted by the client library, so you need have no fear of such pointers being invalidated.

Of course, placing a **Unit*** somewhere will not prevent that **Unit**(p. 36) from being killed; the server will reject any orders given to a dead **Unit**(p. 36). It is not a fatal error; however, you will

need to ensure that unit death does not break your AI, as your Units will almost certainly be killed during the course of a game.

NEVER NEVER NEVER delete a **Unit**(p.36) yourself. It is very likely to cause the client library to segfault. (Remember, if your executable segfaults, you lose by forfeit.)

6.9.2 Constructor & Destructor Documentation

6.9.2.1 mmxi::Unit::Unit (CUnit & *unit*) [inline]

Constructor.

To stop you accidentally creating null units, which will result in considerable segfault risk, the **Unit**(p.36) class has no default constructor. You may create NULL units using this constructor, although it is not recommended. Assignment operator and copy constructors are provided.

6.9.3 Member Function Documentation

6.9.3.1 virtual void mmxi::Unit::attack (const Point & *target*) [inline, virtual]

Attack a location.

You may choose to, instead of attacking a specific **Unit**(p.36), attack a square on the map. Whatever is in that square when your attack lands will be hurt.

6.9.3.2 virtual void mmxi::Unit::attack (const Unit * *target*) [inline, virtual]

Attack a unit.

Attacks all have charge times, which will be arbitrated by the server. If you attempt to attack a unit not in range, or not alive, or nonexistent (the ID you send does not correspond to any units on the board) the server will generate a warning.

If a target unit moves out of range during charge time, the attack will fail.

Castles and Walls cannot attack, and attempting to attack with one will cause your client to generate a warning.

Reimplemented in **mmxi::MoveableUnit** (p.28).

6.9.3.3 virtual void mmxi::Unit::dance () [inline, virtual]

Dance.

Yes, dance. The **Unit**(p.36) will boogie down. (Note that **idle**(p.41) still returns true if the **Unit**(p.36) is dancing; therefore, dancing will probably not foul up your AI.

Reimplemented in **mmxi::MoveableUnit** (p.29).

6.9.3.4 int mmxi::Unit::eta () const [inline]

Returns the number of turns until completion of the Unit's current action.

Again, you do not have this knowledge for units not on your team. The result of this function on a unit not yours is undefined.

6.9.3.5 `virtual Archer* mmxi::Unit::get_archer () [inline, virtual]`

Downcast to an **Archer**(p.15).

So far, client code has not been tested for compatibility with RTTI. We are providing these simple downcast functions for convenience. They will return NULL if the cast is no good, and this otherwise.

Reimplemented in **mmxi::Archer** (p.16).

6.9.3.6 `CUnit& mmxi::Unit::get_cunit () [inline]`

Returns a reference to the underlying **CUnit** structure.

If you want to go fooling around in the protocol, be our guest. We won't help you if you screw it up.

6.9.3.7 `virtual Fighter* mmxi::Unit::get_fighter () [inline, virtual]`

Downcast to a **Fighter**(p.17).

So far, client code has not been tested for compatibility with RTTI. We are providing these simple downcast functions for convenience. They will return NULL if the cast is no good, and this otherwise.

Reimplemented in **mmxi::Fighter** (p.17).

6.9.3.8 `int mmxi::Unit::get_hp () const [inline]`

Returns the hit points of this **Unit**(p.36).

When this function is called on Castles, which do not have hit points, the result is undefined.

6.9.3.9 `int mmxi::Unit::get_id () const [inline]`

Returns this Unit's unique ID number.

Used to identify Units to the server, and specify targets of orders. You will probably not need to look at this number as you write your client.

6.9.3.10 `virtual Incoming* mmxi::Unit::get_incoming () [inline, virtual]`

Downcast to an **Incoming**(p.26).

So far, client code has not been tested for compatibility with RTTI. We are providing these simple downcast functions for convenience. They will return NULL if the cast is no good, and this otherwise.

Reimplemented in **mmxi::Incoming** (p.26).

6.9.3.11 `Point mmxi::Unit::get_loc () const [inline]`

Gets this Unit's location as a **Point**(p.33) structure.

It is recommended that you use this function rather than `get_x()`(p. 38) and `get_y()`(p. 38), as the `Point`(p. 33) provides several useful helper functions.

6.9.3.12 `COrder& mmxi::Unit::get_orders ()` [inline]

Returns a reference to the unit's current `COrder`.

Again, if you want to fool around the underlying protocol, you may, but I think it's a waste of time. We've tried to provide you with everything you need.

6.9.3.13 `int mmxi::Unit::get_team () const` [inline]

Returns this Unit's team ID.

If `get_team()`(p. 41) returns 1, you control this unit. If it returns 2, your opponent controls this unit. Otherwise, it is a neutral unit (such as the Mech.)

6.9.3.14 `UnitType mmxi::Unit::get_type () const` [inline]

Returns the type of this `Unit`(p. 36), in the form of a `UnitType` enum.

Your Castles and Walls are maintained as Units, to keep things simple for us.

6.9.3.15 `virtual Wall* mmxi::Unit::get_wall ()` [inline, virtual]

Downcast to a `Wall`(p. 44).

So far, client code has not been tested for compatibility with RTTI. We are providing these simple downcast functions for convenience. They will return NULL if the cast is no good, and this otherwise.

Reimplemented in `mmxi::Wall` (p. 44).

6.9.3.16 `virtual Wizard* mmxi::Unit::get_wizard ()` [inline, virtual]

Downcast to a `Wizard`(p. 45).

So far, client code has not been tested for compatibility with RTTI. We are providing these simple downcast functions for convenience. They will return NULL if the cast is no good, and this otherwise.

Reimplemented in `mmxi::Wizard` (p. 45).

6.9.3.17 `bool mmxi::Unit::idle () const` [inline]

Returns true if unit is not doing anything.

Note that you only have knowledge of what units are doing for your own units. The return value `idle()`(p. 41) on units that are not on your team is undefined.

6.9.3.18 `virtual bool mmxi::Unit::in_range (const Unit * target)` [inline, virtual]

Checks range to some `Unit`(p. 36).

If `in_range` returns true, you may assume that calling `attack()` (p. 39) on the same `Unit` (p. 36) will succeed. (Unless, of course, that `Unit` (p. 36) moves out of range or does something else nefarious.

Reimplemented in `mmxi::Incoming` (p. 26).

6.9.3.19 `bool mmxi::Unit::is_alive () const [inline]`

Returns true if this `Unit` (p. 36) is living.

You will not have to call this unless you are either looking in the more general members of your `GameBoard` (p. 19) (like `GameBoard::all` (p. 20)) as dead Units will automatically be evicted from most rosters.

It is a non-fatal error to attempt to command a dead unit.

6.9.3.20 `virtual void mmxi::Unit::move (int xdest, int ydest) [inline, virtual]`

Begin moving toward an x,y point.

Not recommended; provided for convenience. The `Point` (p. 33) class is generally superior to x,y coordinate pairs, and is recommended for most purposes (read: all that do not involve mucking around in the protocol, which is also not recommended).

Reimplemented in `mmxi::MoveableUnit` (p. 29).

6.9.3.21 `virtual void mmxi::Unit::move (const Point & destination) [inline, virtual]`

Begin moving to a new square.

Units may only be moved to one of the eight adjacent squares.

The time it takes to move depends on the type of unit and will be arbitrated by the server. If the server determines that your move is valid, on the next turn you may examine this unit's Order to determine how long the move will take.

We will add other helper functions for determining move cost later.

Attempting to move a Castle or a `Wall` (p. 44) is meaningless, and will cause your client to generate a warning.

Attempting to move a dead unit or a unit not yours is a non-fatal error, and will cause the server to generate a warning.

Reimplemented in `mmxi::MoveableUnit` (p. 29).

6.9.3.22 `void mmxi::Unit::move_direction (int direction) [inline]`

Move in one of the eight numbered directions.

Simplest way to move; see `Point::neighbor()` (p. 35) for an explanation of the numbering of the eight cardinal directions.

6.9.3.23 `virtual float mmxi::Unit::range () [inline, virtual]`

Returns this Unit's attack range.

If the distance to some enemy unit is less than this range, it is likely that an attack will succeed - although, remember, the targeted unit can always move to somewhere where your attack will not succeed. This is particularly a problem for Wizards.

Returns 0 if called on a Castle or a **Wall**(p. 44).

Reimplemented in **mmxi::Fighter** (p. 17), **mmxi::Archer** (p. 16), and **mmxi::Wizard** (p. 45).

6.9.3.24 void mmxi::Unit::trashtalk (const char *)

Trashtalk.

Will cause a speech bubble to pop up above your **Unit**(p. 36) in the prettyvis, or a message in xvis. Remember that you have a limited character budget for trashtalking. Please don't flood the server.

6.9.4 Member Data Documentation

6.9.4.1 int mmxi::Unit::info[8]

Space provided for user-defined data.

If you need to store some user data, but don't want to subclass your units, you can do it here. Initialized to 0 in construction.

6.9.4.2 void* mmxi::Unit::infop

Space provided for user-defined data.

If you need to store some user data, but don't want to subclass your units, you can do it here. Initialized to NULL in construction.

The documentation for this class was generated from the following file:

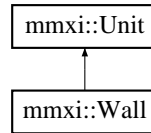
- board.h

6.10 mmxi::Wall Class Reference

Class representing a **Wall**(p. 44).

```
#include <board.h>
```

Inheritance diagram for mmxi::Wall::



Public Member Functions

- **Wall** (CUnit unit)
Constructor.
- virtual **Wall** * **get_wall** ()
Downcast to a Wall(p. 44).

6.10.1 Detailed Description

Class representing a **Wall**(p. 44).

Walls go into their own roster (the **Wall**(p. 44) roster.) You won't find them in the friendly or enemy rosters. Only Wizards can damage walls.

6.10.2 Member Function Documentation

6.10.2.1 virtual Wall* mmxi::Wall::get_wall () [inline, virtual]

Downcast to a **Wall**(p. 44).

So far, client code has not been tested for compatibility with RTTI. We are providing these simple downcast functions for convenience. They will return NULL if the cast is no good, and this otherwise.

Reimplemented from **mmxi::Unit** (p. 41).

The documentation for this class was generated from the following file:

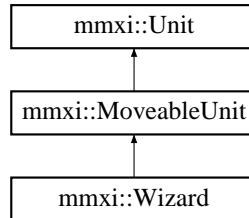
- board.h

6.11 mmxi::Wizard Class Reference

Class representing a **Wizard**(p. 45) unit.

```
#include <board.h>
```

Inheritance diagram for mmxi::Wizard::



Public Member Functions

- **Wizard** (CUnit unit)
Constructor.
- virtual **Wizard** * **get_wizard** ()
Downcast to a Wizard(p. 45).
- virtual float **range** ()
Returns this Unit's attack range.

6.11.1 Detailed Description

Class representing a **Wizard**(p. 45) unit.

Wizards have an incredibly powerful, but slow, attack that pulverizes anything, including walls, in the square it hits and does considerable splash damage to surrounding squares.

6.11.2 Member Function Documentation

6.11.2.1 virtual Wizard* mmxi::Wizard::get_wizard () [inline, virtual]

Downcast to a **Wizard**(p. 45).

So far, client code has not been tested for compatibility with RTTI. We are providing these simple downcast functions for convenience. They will return NULL if the cast is no good, and this otherwise.

Reimplemented from **mmxi::Unit** (p. 41).

6.11.2.2 virtual float mmxi::Wizard::range () [inline, virtual]

Returns this Unit's attack range.

If the distance to some enemy unit is less than this range, it is likely that an attack will succeed - although, remember, the targeted unit can always move to somewhere where your attack will not succeed. This is particularly a problem for Wizards.

Returns 0 if called on a Castle or a **Wall**(p. 44).

Reimplemented from **mmxi::Unit** (p. 42).

The documentation for this class was generated from the following file:

- board.h

Index

- attack
 - mmxi::MoveableUnit, 28
 - mmxi::Unit, 39
- Client Library, 13
- clientlib
 - MMXIMain, 14
 - SendOrder, 14
- closest_point_to
 - mmxi::Point, 34
- dance
 - mmxi::MoveableUnit, 29
 - mmxi::Unit, 39
- distance
 - mmxi::Point, 34
- distsquared
 - mmxi::Point, 34
- eta
 - mmxi::Unit, 39
- farthest_point_from
 - mmxi::Point, 34
- get_archer
 - mmxi::Archer, 16
 - mmxi::Unit, 39
- get_cunit
 - mmxi::Unit, 40
- get_fighter
 - mmxi::Fighter, 17
 - mmxi::Unit, 40
- get_hp
 - mmxi::Unit, 40
- get_id
 - mmxi::Unit, 40
- get_incoming
 - mmxi::Incoming, 26
 - mmxi::Unit, 40
- get_loc
 - mmxi::Unit, 40
- get_orders
 - mmxi::Unit, 41
- get_team
 - mmxi::Unit, 41
- get_type
 - mmxi::Unit, 41
- get_wall
 - mmxi::Unit, 41
 - mmxi::Wall, 44
- get_wizard
 - mmxi::Unit, 41
 - mmxi::Wizard, 45
- idle
 - mmxi::Unit, 41
- in_range
 - mmxi::Incoming, 26
 - mmxi::Unit, 41
- info
 - mmxi::Unit, 43
- infop
 - mmxi::Unit, 43
- is_alive
 - mmxi::Unit, 42
- MakeEnemyArcher
 - mmxi::Player, 31
- MakeEnemyFighter
 - mmxi::Player, 31
- MakeEnemyWizard
 - mmxi::Player, 31
- MakeFriendlyArcher
 - mmxi::Player, 31
- MakeFriendlyFighter
 - mmxi::Player, 31
- MakeFriendlyWizard
 - mmxi::Player, 31
- manhat_dist
 - mmxi::Point, 34
- mmxi::Archer, 15
 - get_archer, 16
 - range, 16
- mmxi::Fighter, 17
 - get_fighter, 17
 - range, 17
- mmxi::GameBoard, 19
- mmxi::GameConstants, 23
- mmxi::Incoming, 26
 - get_incoming, 26

- in_range, 26
- mmxi::MoveableUnit, 28
- mmxi::MoveableUnit
 - attack, 28
 - dance, 29
 - move, 29
- mmxi::Player, 30
 - MakeEnemyArcher, 31
 - MakeEnemyFighter, 31
 - MakeEnemyWizard, 31
 - MakeFriendlyArcher, 31
 - MakeFriendlyFighter, 31
 - MakeFriendlyWizard, 31
 - Turn, 32
- mmxi::Point, 33
 - closest_point_to, 34
 - distance, 34
 - distsquared, 34
 - farthest_point_from, 34
 - manhat_dist, 34
 - neighbor, 35
- mmxi::Unit, 36
 - attack, 39
 - dance, 39
 - eta, 39
 - get_archer, 39
 - get_cunit, 40
 - get_fighter, 40
 - get_hp, 40
 - get_id, 40
 - get_incoming, 40
 - get_loc, 40
 - get_orders, 41
 - get_team, 41
 - get_type, 41
 - get_wall, 41
 - get_wizard, 41
 - idle, 41
 - in_range, 41
 - info, 43
 - infop, 43
 - is_alive, 42
 - move, 42
 - move_direction, 42
 - range, 42
 - trashtalk, 43
 - Unit, 39
- mmxi::Wall, 44
 - get_wall, 44
- mmxi::Wizard, 45
 - get_wizard, 45
 - range, 45
- MMXIMain
 - clientlib, 14
- move
 - mmxi::MoveableUnit, 29
 - mmxi::Unit, 42
- move_direction
 - mmxi::Unit, 42
- neighbor
 - mmxi::Point, 35
- range
 - mmxi::Archer, 16
 - mmxi::Fighter, 17
 - mmxi::Unit, 42
 - mmxi::Wizard, 45
- SendOrder
 - clientlib, 14
- trashtalk
 - mmxi::Unit, 43
- Turn
 - mmxi::Player, 32
- Unit
 - mmxi::Unit, 39