How To Write a Bot Movement Controller

BigWorld Technology 2.0. Released 2010.

Software designed and built in Australia by BigWorld.

Level 2, Wentworth Park Grandstand, Wattle St
Glebe NSW 2037, Australia
www.bigworldtech.com

Table of Contents

Chapter 1. What is a Movement Controller?

A *movement controller* is an object that controls the movement of a bot.

More specifically, it must sub-class the `MovementController` class defined in `bigworld/src/server/tools/bots/movement_controller.[ch]pp`, and therefore implement the virtual method `nextStep()`, which supplies position and orientation updates for that bot.

Additionally, you will need to implement a sub-class of `MovementFactory` (defined in the same source files as `MovementController`) to supply `MovementController` objects to bots, either at bot instantiation time, or on change of movement controller, via the virtual method `create()`. For details, see "The `create()` method" on page 7 .

Note

The Patrol movement controller is recommended as a reference companion to this document.

It can be found in `src/server/tools/bots/patrol_graph.[ch]pp`.

This file also contains useful classes other than the movement controller and factory referenced in this document.

Chapter 2. The `MovementFactory`

We will deal with the implementation of the `MovementFactory` first because you cannot test your `MovementController` without a working one, and it also illustrates the parameters that will be available to your `MovementController` when it is instantiated.

2.1. Constructor and global registration

The `MovementFactory` need not be declared in your `MovementController`'s header file, as it will never be instantiated directly. Instead, you will declare a single static instance of the `MovementFactory` following its definition, whose constructor globally registers it as the factory for that controller type.

For example, in `src/server/tools/bots/patrol_graph.cpp`:

```
/* Source abbreviated for illustrative purposes */
namespace
{
    class PatrolFactory : public MovementFactory
    {
    public:
        PatrolFactory() : MovementFactory( "Patrol" )
        {
            /* any required init, usually unnecessary */
        }

        MovementController *create( const std::string & data, float & speed,
 Vector3 & position )
        {
            /* parse data string and create concrete subclass of
 MovementController */
        }
    };

    PatrolFactory s_patrolFactory;
}
```

The constructor for `PatrolFactory` calls the constructor of `MovementFactory` to globally register the object as the factory for generating `MovementController` objects of type `Patrol`.

After the class definition, we declare a single instance of the factory class, which will associate itself with the name `Patrol` from that point on. It is useful to declare the `MovementFactory` class in an anonymous namespace (or declare it static) to avoid polluting the top-level namespace with the name of the declared instance.

2.2. The `create()` method

The first parameter passed to this method (`const std::string & data`) is the parameter passed to the `setDefaultControllerData` and `updateMovement` watchers when controlling bots. The `create()` method needs to parse this string, and then use the parsed data to create an instance of the movement controller.

Additionally, the method receives the speed and position of the bot, which is usually passed to the constructor of the movement controller.

Note

For more details, see the Server Operations Guide's chapter *Stress Testing with Bots*.

Chapter 3. The MovementController

The movement controller needs to maintain a set of state variables that are used to control the bot's movements.

For example, the `GraphTraverser` movement controller in `src/server/tools/bots/patrol_graph.[ch]pp` keeps track of the following information:

- A graph that the bot is traversing.
- The node towards which the bot is heading.
- A position near that node toward which it is actually heading.
- A timer for spending time at a given node.

3.1. The `nextStep()` method

The most important part of the movement controller is the `nextStep()` method:

```
bool nextStep( float & speed, float dTime, Vector3 & pos, Direction3D & dir )
```

The `dTime` parameter is the time elapsed since the last call to `nextStep()`.

Aside from updating any state variables as necessary, the general function of this method is to establish some unit vector the bot is moving down, and to update `pos` and `dir` appropriately, according to the formula below:

pos += unitvec * (speed * dTime);

dir.yaw = unitvec.yaw()

Chapter 4. Useful Utility Classes

Reusing the graph classes defined in `bigworld/src/server/tools/bots/patrol_graph.[ch]pp` may considerably speed your controller implementation.

Among other things, it provides useful methods to randomly select nodes to travel to, and also to generate random positions within a node according to XML-defined graphs. For more details, see the Server Operations Guide's chapter *Stress Testing with Bots* section “Controlling Movement”.