Malmo Platform Tutorial

To begin:

From the root of your Malmo deployment:

1. Launch Minecraft:

```
cd Minecraft
launchClient.bat   (on Windows)
./launchClient.sh  (on Linux or MacOSX)
```

(NB: If you see a line saying something like "Building 95%", ignore it – you don't need to wait for this to complete.)

2. Open a terminal/command prompt and navigate to Python_Examples

1. Standing around in fields

Start by running tutorial_1.py. This is barest skeleton of a mission – the agent does nothing but stand in a field.

When you run it you should see something like this:

```
c:\Malmo\Python_Examples>python tutorial_1.py
DEBUG: Sending MissionInit to 127.0.0.1 : 10000
DEBUG: Looking for client, received reply from 127.0.0.1: MalmoOK
Waiting for the mission to start .....
Mission running ..............................................................
Mission ended
```

Notice the countdown in the bottom left corner of the Minecraft window – the default mission ends after ten seconds.

2. Get moving

While a mission is running, you can send commands to the agent to control it. Try adding this just before the main mission loop:

```
agent_host.sendCommand("turn -0.5")
agent_host.sendCommand("move 1")
agent_host.sendCommand("jump 1")
```

When you run the mission again you should see the agent moving.

> TIP: Pressing F5 in the Minecraft window will give you alternative views on the player – this can be helpful for seeing what's going on.
> Pressing F3 will display Minecraft's debug information – you should be able to see the position and orientation changing.

By default, the agent is being controlled by the *ContinuousMovementCommands* – these are:

- move [-1,1]
  - "move 1" is full speed ahead; "move -0.5" moves backwards at half speed, etc.
- strafe [-1,1]
  - "strafe -1" moves left at full speed; "strafe 1" moves right at full speed, etc.

- pitch [-1,1]
    - "pitch 1" starts tipping camera upwards at full speed, "pitch -0.1" starts looking down slowly, etc.
- turn [-1,1]
    - "turn -1" starts turning full speed left, etc.
- jump 1/0
    - "jump 1" starts jumping; "jump 0" stops.
- crouch 1/0
- attack 1/0
- use 1/0
- hotbar.0 – hotbar.8 1/0 [we'll use these later]

> TIP: Minecraft has a day/night cycle which takes around 20 minutes, so after ten minutes the world will be shrouded in darkness. To return to the light of day, click on the Minecraft window and type:
>
> `/time set 1000`
>
> (This corresponds to the start of the Minecraft day; 13000 is sunset, the start of the Minecraft night.)

Try experimenting with combinations of these commands, both in our outside of the main mission loop. For example, what would happen if you replaced the previous commands with this?

```
agent_host.sendCommand("pitch -1")
time.sleep(1)
agent_host.sendCommand("attack 1")
```

3. Introducing the Mission XML

This line:

```
my_mission = MalmoPython.MissionSpec()
```

is doing some work behind the scenes to create a default Mission XML string. It's this XML that is sent to Minecraft to specify the mission.

To see the XML that is being sent, you can call:

```
print my_mission.getAsXML(True)
```

It should produce something like this:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Mission xmlns="http://ProjectMalmo.microsoft.com" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://ProjectMalmo.microsoft.com Mission.xsd">
  <About>
    <Summary/>
  </About>
  <ServerSection>
    <ServerHandlers>
      <FlatWorldGenerator generatorString="3;7,220*1,5*3,2;3;,biome_1"/>
      <ServerQuitFromTimeUp description="" timeLimitMs="10000"/>
      <ServerQuitWhenAnyAgentFinishes description=""/>
    </ServerHandlers>
  </ServerSection>
  <AgentSection mode="Survival">
    <Name>Cristina</Name>
    <AgentStart/>
    <AgentHandlers>
      <ObservationFromFullStats/>
```

```
      <ContinuousMovementCommands turnSpeedDegs="180"/>
    </AgentHandlers>
  </AgentSection>
</Mission>
```

The MissionSpec object provides a basic API for manipulating this XML, but we'll edit it directly for the following examples.

Open tutorial_2.py and you'll see the XML is being passed directly to the MissionSpec constructor. We've also filled in a few of the blanks in the default mission, and upped the time limit to 30 seconds. Try running it – we're back to standing in a field.

> TIP: Rather than wait for the mission to end, you can interrupt the Python process by pressing Ctrl-C. Minecraft should detect this and prepare itself for the next mission. You can check this has happened by switching on the Minecraft diagnostics – from the main menu click on "Mods", select "Microsoft MalmoPlatform" in the list on the left, hit "Config", and then click on the "debugDisplayLevel" button until "Show all diagnostics" appears. (You can also do this while a game is running by pressing <Escape> and selecting "Mod Options…")

4. Controlling our environment

Firstly, this field is pretty boring. Let's jazz it up – change the FlatWorldGenerator generatorString to the more interesting one at the top of the python file and rerun.

> Note: The mission should take longer to start this time. This is because the world requirements have changed. The platform tries to reuse worlds as an optimisation because world creation is very expensive, so a new world is only built when necessary. This means that certain changes to the Minecraft environment may persist between missions – something to be aware of.

Generator strings can be created using online tools – eg http://chunkbase.com/apps/superflat-generator

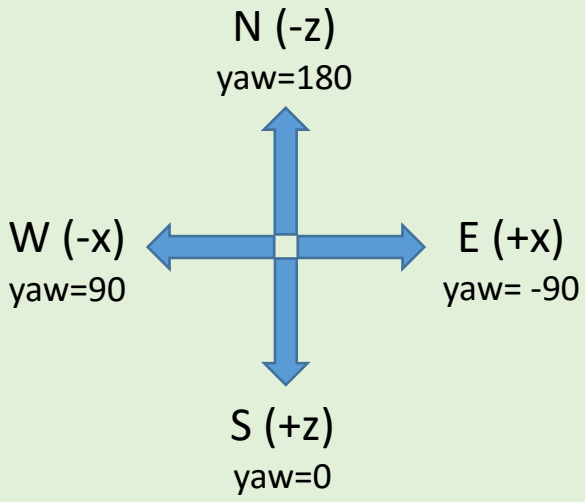
Now let's set the time to permanent twilight – add this to the top of **ServerSection**, just before **ServerHandlers**:

```
<ServerInitialConditions>
    <Time>
        <StartTime>12000</StartTime>
        <AllowPassageOfTime>false</AllowPassageOfTime>
    </Time>
</ServerInitialConditions>
```

Now we have a nice sunset, but we can't see it – we'd like our agent to start off by facing the sun, so expand the **AgentStart** node to this:

```
<AgentStart>
    <Placement x="0" y="56" z="0" yaw="90"/>
</AgentStart>
```

TIP: Coordinates in Minecraft work as follows:
(The y-axis corresponds to height)



Only one thing could possibly improve on the beauty of Minecraft's sunset: snow. Try adding this to the **ServerInitialConditions** section, after the **Time** block we added earlier:

```
<Weather>rain</Weather>
```

(Minecraft only has one category for snow and rain – which one you get depends on the *biome* – cold biomes snow, warm biomes rain. The biome is defined by the generatorString.)

TIP: Minecraft weather is random, and affects things like visibility, light levels, the player's traction, etc. To ensure controlled conditions for an experiment, use `<Weather>clear</Weather>`.

5. Decorating

It may be pink and snow-covered now, but we're still just standing around in a vast flat field. (The FlatWorldGenerator does exactly what its name suggests). It's time to add some features. Try adding this code to the **ServerHandlers** block, right after the **FlatWorldGenerator** line:

```
<DrawingDecorator>
    <DrawSphere x="-27" y="70" z="0" radius="30" type="air"/>
</DrawingDecorator>
```

If you run the script now, your agent should find himself on the lip of a vast rainbow-coloured crater. The DrawingDecorator allows us to draw primitive shapes out of Minecraft blocks. The available primitives are:

```
<DrawCuboid x1, y1, z1, x2, y2, z2, type/>
<DrawLine x1, y1, z1, x2, y2, z2, type/>
<DrawBlock x, y, z, type/>
<DrawSphere x, y, z, radius, type/>
<DrawItem x, y, z, type/>
```

For a list of the block and item types available, see *Types.xsd*.
Try experimenting with these drawing commands yourself.

TIP: The platform disables mouse control of Minecraft by default, in order to prevent accidental mouse movements from affecting experiments. To explore the Minecraft world you have decorated, you can toggle between human and platform mouse control – click on the Minecraft window and press <Enter> to switch. Once you have control, explore the world using the mouse, and the <W>, <S>, <A> and <D> keys (assuming you have the default Minecraft setup.) The platform will automatically re-take control at the start of each mission.

The ability to create XML dynamically from within a Python script gives us a great deal of power. Run tutorial_3.py to see an example.

6. The Inventory

To progress beyond simple navigation tasks we need to make use of the inventory. The agent can be equipped at the start of each mission by adding an **Inventory** section to the **AgentStart** node, after the **Placement** node, for example:

```
<Inventory>
    <InventoryItem slot="0" type="diamond_pickaxe"/>
</Inventory>
```

There are 40 inventory slots in Minecraft, numbered 0-39:

- 0-8 are the "hotbar" slots – they are displayed on the HUD and accessed with the hotbar keys, and can be selected by the agent using the "hotbar.x" command provided by the *ContinuousMovementCommands* (remember section 2).
  Note: The slots are 0-indexed but the key commands are 1-indexed, so to select slot 0, the command sequence would be:
  - agent_host.sendCommand("hotbar.9 1") # press the key
  - agent_host.sendCommand("hotbar.9 0") # release the key
- 9-35 are the three rows of items visible in the player's inventory menu (press <E> within the game to view this)
- 36-39 are reserved for the four armour slots (eg for diamond_helmet, etc)

For the purposes of this worksheet, we'll ignore slots 9-39, but for further reading look at the *ObservationFromFullInventory*, *ObservationFromHotBar* and *InventoryCommands* mission handlers.

CHALLENGE TIME:

Run tutorial_4.py (it should look fairly familiar).
In the ground-level centre of the Menger sponge is a diamond block. Using what we know already, can you get your agent there before the time runs out?
(It can be done with seven lines of code.)

7. Quit Producers

We added a new concept in tutorial_4.py – *AgentQuitFromReachingPosition*
This is a *QuitProducer* – it provides a way for the platform to decide when the mission has ended.
(The countdown timer we've been using – *ServerQuitFromTimeUp* – is another example.)

The mission ends when the agent comes within a certain tolerance (set to 0.5 in this example) of the specified position. Multiple end points can be specified this way.

An alternative way to do this would be using *AgentQuitFromTouchingBlockType*:

```
<AgentQuitFromTouchingBlockType>
    <Block type="diamond_block" />
</AgentQuitFromTouchingBlockType>
```

The mission also ends when the agent dies. Try adding your solution from tutorial_4.py to tutorial_5.py (or just un-comment the solution we've provided). What happens?

8. Observation Producers

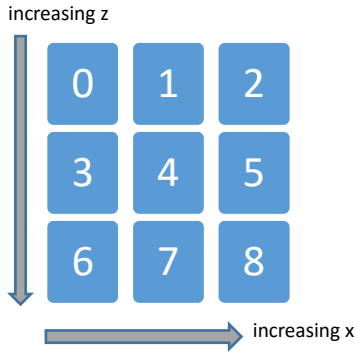
How can we help our agent escape their fiery death?
One way is by using an observation producer. These spit out information throughout the mission lifetime. We'll use *ObservationFromGrid* – look in the XML in tutorial_5.py and you will see the following has been added:

```
<ObservationFromGrid>
    <Grid name="floor3x3">
      <min x="-1" y="-1" z="-1"/>
      <max x="1" y="-1" z="1"/>
    </Grid>
</ObservationFromGrid>
```

Observations are returned as JSON and are accessed via *agent_host.getWorldState().observations*
ObservationFromGrid returns a flattened array of the names of the blocks surrounding the player. The above code asks for the platform to provide the 3x3 grid of blocks directly under the player's feet, and to return it in a JSON array named "floor3x3". A typical output might be:

floor3x3: ['lava', 'obsidian', 'obsidian', 'lava', 'obsidian', 'obsidian', 'lava', 'obsidian', 'obsidian']

The grid is ordered by x, then z, then y – this diagram might help (the numbers are the index of the cell in the flattened array).



For an agent facing west (towards negative x), for example, the square directly in front of him would be at position 3.

CHALLENGE TIME:

Add code, where marked, in tutorial_5.py, to help your agent get to the diamond block without catching fire.

9. Rewards and discrete actions

So far we've been using continuous actions. Much of the traditional Reinforcement Learning literature assumes a discrete action space. The platform allows for this, with the *DiscreteMovementCommands*.

Take a look at tutorial_6.xml – it's the raw mission XML file for a traditional cliff-walking experiment. There are a few new features to note:

- <DiscreteMovementCommands/> - allows the agent to move instantly one block at a time in any direction
- <RewardForTouchingBlockType> - sets up a positive *reward* for touching for touching a lapis lazuli block, and a negative reward for touching lava
- <RewardForSendingCommand> - sets up a reward of -1 for each command the agent sends to Minecraft

The rewards are returned to the agent in the same way as the observations; each "world tick" (20hz) a reward value will be returned. (It will be 0 if nothing has happened since the last one.)

Now look at tutorial_6.py, which uses this XML file. There are several new concepts here too:

- Loading the XML from file:

```
mission_file = './tutorial_6.xml'
with open(mission_file, 'r') as f:
    print "Loading mission from %s" % mission_file
    mission_xml = f.read()
    my_mission = MalmoPython.MissionSpec(mission_xml, True)
```

- Combining raw XML with the API:
  Having loaded the raw XML, it's possible to modify it using the API, as here:

```
# add 20% holes for interest
for x in range(1,4):
        for z in range(1,13):
            if random.random()<0.1:
                my_mission.drawBlock( x,45,z,"lava")
```

  Although we've not covered the API much in this tutorial, this combination of files and API is a very flexible and powerful approach.

- Running the mission in a loop.
  Up to now we've only been running one mission per script. Tutorial_6.py runs the mission repeatedly:

```
…
for i in range(num_repeats):
…
```

CHALLENGE TIME:

Now run tutorial_6.py. A simple agent walks around a grid at random, often dying when it falls in the lava. Your job is to help the poor player survive for longer and to reach the blue goal block.

In the *TabQAgent* class are two functions *updateQTable* and *updateQTableFromTerminatingState* that need to be filled out – see lines 40 and 52.

Try replacing these lines:

```
new_q = old_q
```

with this:

```
new_q = reward
```

The reward structure is specified in the XML, as we've seen: when the agent falls in lava it gets a reward of -100 (and the mission ends). When they reach a blue block they get a reward of +100 (and the mission ends). Additionally they get a reward of -1 every time they take a step.

Now run the agent again with the changes. Notice that the map is updated with the reward values and the agent falls in the lava less often. (Their policy is to take a random move 1% of the time, instead of choosing the best one. This is called an 'epsilon-greedy' policy, with epsilon=0.01.)

Let the agent run around until it finally stumbles upon the blue block, and gets a larger reward. One of the dots turns green on the previous square to indicate that moving in that direction was a good idea.

Your challenge is to write code in those two functions to solve missions like this one as quickly as possible, simply by updating the 'action values' or 'q values' in a more intelligent manner.

10. Advanced decorators

The previous python script randomly removed blocks from the floor to make a more interesting challenge. You may have noticed that occasionally this resulted in a path that was impassable for the discrete agent.

The python script could be made more intelligent to prevent this from happening, but the platform includes some more advanced decorators which will do similar things for you.

The *MazeDecorator* was designed for precisely these sorts of cliff-walking tasks, but can also be modified in a number of different ways. In essence, it creates a grid with a start and end square and a guaranteed path across (for either 8-connected or 4-connected grids.)

CHALLENGE TIME:

Try playing tutorial_7.py manually (hit <Enter> at the start of each mission to gain mouse control).

Can you construct an agent that will solve the mazes?

Take a look at the *MazeRunner.py* sample for some more complicated maze options (and for an agent that can solve them, albeit by cheating!)

The *ClassRoomDecorator* is a whole sub-game ecosystem, designed to create larger-scale challenges. Try playing tutorial_8.py (press <Enter> to gain mouse control, Ctrl-C on the Python process when you get bored.)
The ClassRoomDecorator is also highly customisable – check out the sample_missions/classroom folder for some examples.

11. Now without cheating

ObservationFromGrid is handy, but it arguably constitutes cheating. The goal is to be able to create agents that can make decisions based purely on what they *see*. The platform will return video to the agent – simply add a *VideoProducer* to the **AgentHandlers** section of the XML:

```
<VideoProducer want_depth="true">
    <Width>320</Width>
    <Height>240</Height>
</VideoProducer>
```

Take a look at the *depth_map_runner.py* sample for a relatively simple example of an agent that attempts to navigate from the video (though using the depth map is still arguably cheating.)

BONUS CHALLENGE TIME:

Is it possible to use simple image processing to work out how to steer an agent through a MazeDecorator maze? Eg set the time to permanent daylight, choose high-contrast coloured blocks for path and gap, tilt the camera down to look at the floor, and examine the centre pixels of each video frame?

12. Into the world

So far we've only used the Flatworld generator, but Minecraft wouldn't have captured the imaginations of an entire generation if flat worlds were all it had to offer…
Go back to tutorial_2.py and replace the Flatworld generator line with this:

```
<DefaultWorldGenerator/>
```

Run it and enjoy!

BONUS CHALLENGE TIME:

Add some code to tutorial_2.py to solve general AI.