CrowdAI developer manual

Pablo González de Prado Salas

2018

Contents

1 Preface **2**

2 Developing CrowdAI **2**

2.1 Minecraft 2

2.1.1 New Minecraft world 2

2.1.2 New Minecraft agent controller 3

2.1.3 Using fitness 5

2.2 Simulation parameters 5

2.3 Website 7

3 Publishing CrowdAI **9**

4 Flow of the code **11**

4.1 Starting the path from click to video display 12

4.2 Redirect to waiting page 13

4.3 From RaiseStartEvolutionEvent to evolution ready 14

4.4 From genome evaluation to video files 16

1 Preface

Thank you for your interest in CrowdAI. This project explores collective and interactive evolution of controllers for Minecraft agents, and it was designed with the idea of making a platform that would be easy to adapt and extend. Hopefully this guide will help you with that.

I am Pablo González de Prado Salas[1], and this project was part of my postdoc at ITU with Sebastian Risi[2].

Good luck, and don't feel shy to contact us to let us know what you are doing with the project or to ask for help if something is not clear!

2 Developing CrowdAI

In this section you will find a quick orientation for the most common changes you would want to make in this platform.

Note: VisualStudio should be opened with administrator privileges!

2.1 Minecraft

In general, changes related to Minecraft are part of Project Malmo[3], and you should read their documentation for help. However, here we will identify the relevant classes and provide some tips.

2.1.1 New Minecraft world

The basic template for the Minecraft world is stored in `Evolution/minecraftWorldXML.txt`. This file is read by the `ProgramMalmo` class. The world information is stored in the line:

```
<FlatWorldGenerator generatorString="3;7,45*48;,biome_1"/>
```

[1]gonzalezdepradosalas.weebly.com

[2]http://sebastianrisi.com

[3]https://www.microsoft.com/en-us/research/project/project-malmo/

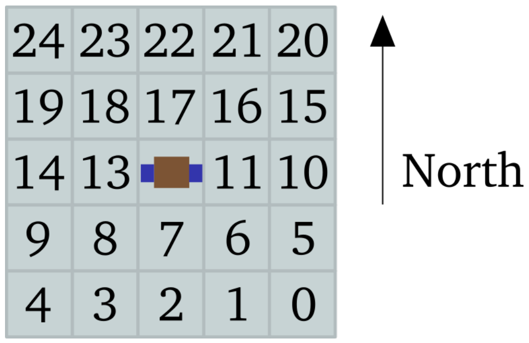

Figure 1: Malmo can return information from tiles relative to the agent. Note that the indexing is not relative to the agent's orientation.

You have on-line tools to help create such strings for simple worlds. (See, for example: https://goo.gl/CnbJNp). In case you want the standard game world use `<DefaultWorldGenerator/>`.

It is likely you will need more complex structures, or procedurally created content (walls, holes, etc.). You may create a dedicated class for this, which `ProgramMalmo` will call. In our case, look at `MazeMaker`. In truth, the only special command you need to know is the new block command (which works on a `MissionSpec` object):

```
<mission.drawBlock(block.xCoord, block.yCoord,
                   block.zCoord, block.type);/>
```

2.1.2 New Minecraft agent controller

Class `ProgramMalmo` is responsible for creating Minecraft simulations. Periodically, (see `ProgramMalmo.MainLoop()` is will create events, which pass information about the Minecraft world in their argument. See `JsonObservations` for the list of available world parameters. These may be extended within Project Malmo capabilities. As an example, we work with custom lists with information of nearby blocks. These have been setup in the Minecraft configuration file Evolution/minecraftWorldXML.txt within the field `<ObservationFromGrid>`, as in Fig. 1. We use three such lattices (for information from tiles with different vertical coordinates).

Because the indices are relative to the world compass and not the agent's orientation, we created a class to process these lattices (`<ManyTiles>`).

`ProgramMalmo` also sends commands to the Minecraft simulation, which controls the agent (`ProgramMalmo.ExecuteCommands()`).

The agent controller is responsible for taking these world updates (inputs) and updating the list of commands in `ProgramMalmo` (outputs). You can determine which class will do this in
`OnlineMalmoPseudoEvaluator.Evaluate`. In our case, we are using a `MazeControllerNoTurn` class. Any controller that is created here must receive a reference to its corresponding `ProgramMalmo` object, and will subscribe to the would update events. In the standard version each user has a single `ProgramMalmo` object that will be used for all Minecraft simulations, but in the parallel version each genome gets a unique `ProgramMalmo`/controller pair.

This code is a minimal example of how the controller works

```
void WhenObservationsEvent(
        object sender, ObservationEventArgs malmoInfo)
{
    brain.InputSignalArray[0] = malmoInfo.DamageTaken;
    brain.InputSignalArray[1] = malmoInfo.XPos;
    brain.Activate();
    if (brain.OutputSignalArray[0] > actionThreshold)
        programMalmo.AddCommand("move 1");
}
```

Here "brain" is a phenome, or processed version of the genome. It is the neural network responsible for taking inputs and producing outputs. The responsibility of the controller is to translate world information into neural network inputs, and neural network outputs into Minecraft commands.

Important! If you change the number of inputs or outputs used by the neural network, you must update the properties `InputCount` and `OutputCount` in `MyExperiment`. Note that `InputCount` is the number of inputs **not** considering bias (so if you are only using one input for, say, the agent's x-coordinate, here you would write "1", not "2").

2.1.3 Using fitness

`ProgramMalmo` is called from `OnlineMalmoPseudoEvaluator` which is used to evaluate genomes. In our case this means taking a genome and producing a video, but it is also possible to update the genome's fitness. Notice this class has access to `ProgramMalmo` and thus to the Minecraft world and the simulation details, which may be used to produce a fitness value (pass zero otherwise). If you want to use this fitness value, remember to check that it is, in fact, used by the `IGenomeListEvaluator` used. For instance, we have deactivated fitness update in `SerialGenomeListEvaluator.EvaluateOne` (so that creating a video won't overwrite any fitness values, in case they are used).

The `IGenomeListEvaluator` that is used by evolution is determined in `SimpleNeatExperiment.CreateEvolutionAlgorithm` (the method is overloaded, this information is in the main version, the one taking a genome factory and a genome list).

2.2 Simulation parameters

There are many parameters driving a NEAT simulation. Here we will go through all of them, which are inconveniently scattered across different classes.

NeatEvolutionAlgorithmParameters

This class controls the parameters relevant for the production of offspring. Here we control the elitism proportion, asexual, sexual and inter-species reproduction rates. In general, these are not the first parameters to change.

NeatGenomeParameters

This class controls the parameters relevant for genome mutations. Here we can adjust the initial connectivity of the network (proportion of possible connections that will be randomly wired).

Very relevant: here we adjust the relative probabilities for the different types of mutations. If add node is too high, for example, genomes will quickly grow in size, which can be a problem both for the search algorithm and for performance.

AuxState for nodes is not used in this version.

`ConnectionMutationInfoList` methods return the different types of mutations for connection weights. We can add different types of mutation with relative likelihoods of being selected! We have Gaussian perturbations (we define the width of the Gaussian curve, the "sigma") and connection resets. We can also have a fixed number of mutations (say, "mutate five connections") or establish a proportion ("mutate 30% of all connections").

Note that during evolution we may load different mutation schemes (see, for example, how `CreateParametersForBigChanges` is used in `NeatGenomeFactory`).

Evolution/Malmo.config.txt

This file determines the **population size**, the **number of species**, the type of network (feedforward or cyclic, and the number of activation iterations in cyclic networks).

ComplexityThreshold and ComplexityRegulationStrategy determine when the algorithm will try to start reducing the complexity of the genomes, using special parameter schemes from `NeatEvolutionAlgorithmParameters` and `NeatGenomeParameters` (see `CreateSimplifyingParameters` methods in both classes).

MyExperiment

In this class we only set the number of input and output nodes used by the networks, as well as the class used to evaluate genomes (meaning assigning a fitness value or, in our case, producing a video file).

However, this class inherits from `SimpleNeatExperiment` where many important decisions are made. Most important is the method `CreateEvolutionAlgorithm` (the one taking a genome factory and a genome list). Here we decide:

- Distance metric (to compare genomes)

- Speciation strategy (to cluster genomes)
- Complexity regulation strategy. This is how it will be decided that genomes are “too complex”, for example, by counting connections. When too complex, the “simplifying” parameters are used, trying to reduce genome complexity while retaining functionality.
- The evolution algorithm and genome-to-phenome decoders may also be selected here (although these would not be minor changes).
- Genome evaluator, that take a list of genomes to update their fitness (in our case, to produce videos). Note some evaluators may take other evaluators as parameters. For example, `SelectiveGenomeListEvaluator` takes a genome list and prunes out genomes that have been already evaluated, and then returns that list to another genome evaluator. Remember that the final evaluator, that takes only one genome, has been set in `MyExperiment`.

PopulationReadWrite

Folder structure is determined here. It is important to set it up correctly! It also determines whether we are working on debug or release (because release creates a different tree of folders and we have to take that into account to move files around!) Here we also choose the path for error messages and user activity data (logs).

As a rule, you will need to check and possibly update any absolute paths, while relative paths will probably be Ok (as long as you don’t alter the folder tree).

2.3 Website

This project has been written using ASP.NET_MVC[4,5], which allows for natural integration of C#-based projects in web services.

[4]https://en.wikipedia.org/wiki/ASP.NET_MVC
[5]https://www.asp.net/mvc

Website elements can be found in project INM. The folder "views" has the HTML code for the different pages in the website. To add a new page, simply right click on the desired folder and select Add→Views.... Note that all views are associated to an action in a "controller" (it helps to be consistent with the folder naming). These actions determine what happens when a user navigates to that page. For example, a basic action for a page called About (for example websiteName/Home/About) would be, within the `HomeController`,

```
public ActionResult About()
{
    \\maybe do stuff here? Let's increment a variable:
    ++exampleVariable;
    return View();
}
```

You may use more advanced front-end programming if you wish. The folder Scripts contains many javaScript files.

You may want to change how inactive users are handled. Look at section 4.3.

The last piece of the MVC puzzle is the "model". From these classes we can create objects needed in the user-server communication and are handled in databases. We created the "candidate" class with information about the controllers users evolve and publish (the controller name and description, the parent ID, the path to the video file, etc.). These objects may be passed to views. See these two examples:

First example, that returns all candidates in the database:

```
// GET: Pictures
public ActionResult ShowSavedCandidates()
{
    // This takes all pictures from the database
    var allPictures = db.Candidates.ToList();
    return View(allPictures);
}
```

The second example only passes three objects, corresponding to the three active candidates in a user's evolutionary process:

```
// This variable is defined in the controller class
private List<Candidate> candidates;

// Shows the candidate videos after each generation
public ActionResult Index()
{
    string userName = HttpContext.UserIdentity();
    // This resets the candidate objects that will be used
    PrepareCandidateModels();
    if (!CheckIfUserExists(userName))
    {
     return RedirectToAction("UnexpectedError");
    }
    // This fills the candidates with the right information
    UpdateCandidatePaths(userName);
    // The candidates are passed to View
    return View(candidates);
}
```

3 Publishing CrowdAI

Publishing CrowdAI may be harder than expected, at least for non experts. Most of the trouble derives from Project Malmo and its dependencies. So your first step should be to carefully install Project Malmo, which you probably already did in order to develop your project.

Some project properties that may be relevant are:
Properties/Application: target framework, .NET 4.5.2.
Properties/Build: platform Active, (Any CPU); platform target, x64.

- I followed this procedure: https://goo.gl/GTbR6Y

Table 1: Application Pool parameters

.NET version	v4.0
Enable 32-bit applications	false
Managed pipeline mode	integrated
Process model identity	myUserName (superuser)

- Using Visual Studio, we published the project to a folder, and then we setup the website using IIS: https://goo.gl/ysvt2e
- Table 1 shows some important parameters that were used. Having an Application Pool with superuser privileges is not a great security measure. The alternative is to find and grant privileges for all the processes that Malmo requires outside of its own folder, but this may be hard to do, given the several dependencies associated with Malmo.
- Don't forget folder permissions. We also added IIS_IUSRS to the list of user names. https://goo.gl/fNeY51
- Don't forget to update the folder paths in the `PopulationReadWrite` class (see Sec. 2.2).
- In the website folder you will find a Web.config file. In our case, Visual Studio creates an incorrect file, with several references to the local database (LocalDb)\MSSQLLocalDB. These need to be fixed, in our case to USERWIN\SQLEXPRESS.
- SQL may be troublesome. It is better that someone with some experience publishes the project. If you can't have that, maybe these links will help:

  https://goo.gl/Hdh4dC

  https://goo.gl/Hdh4dC

  https://goo.gl/Vms7DJ

  https://goo.gl/3qTJ1g

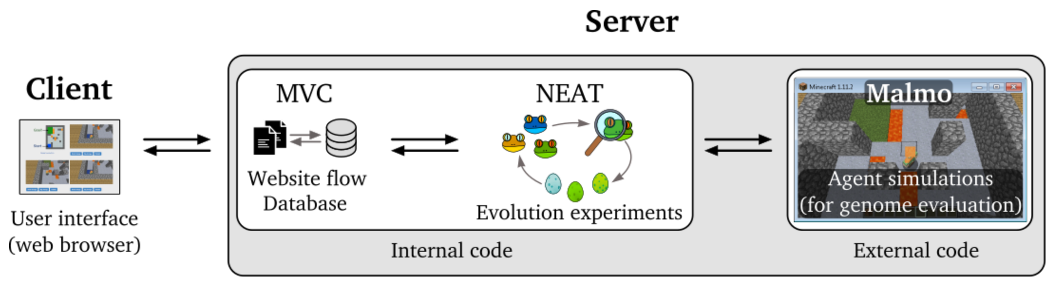

Figure 2: Webservice structure.

4 Flow of the code

CrowdAI is a web service, meaning that there is a server (backend) that interacts with the users' browsers (frontend). The service is implemented using ASP.NET-MVC.

The main project in the solution (in Visual Studio a "solution" may be composed by smaller "projects") is responsible for the coordination between client (user side) and server. This project is awkwardly named INM (like the solution, from "Interactive neuroevolution in Minecraft").

When an evolutionary process needs to be started or continued, then project INM communicates with project Evolution, where this happens. Project Evolution contains our version of SharpNEAT for this.

Genomes are evaluated by humans (interactive evolution). For this humans need videos of Minecraft agents controlled by the different genomes. In order to create this videos the project Evolution includes a Project Malmo interface, which creates Minecraft simulations using external code (provided via .dll files).

It is relevant to note that project INM has a reference to project Evolution, but not the other way around (which would cause a forbidden dependence loop).

Thus we can identify the following different areas:

- Client (user interface)
- Client-evolution coordination
- Evolution

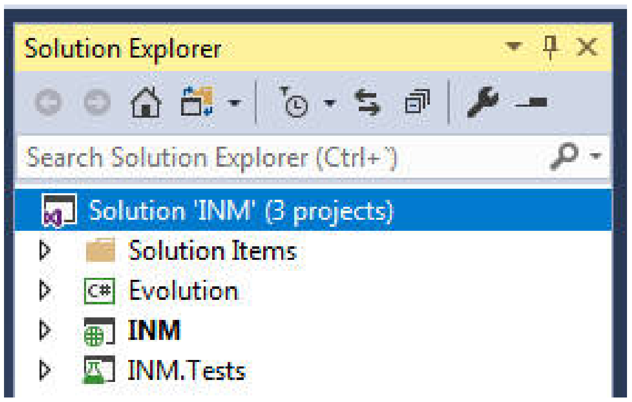
Solution Explorer
Search Solution Explorer (Ctrl+`)
Solution 'INM' (3 projects)
Solution Items
Evolution
INM
INM.Tests

Figure 3: Visual Studio terminology: the "solution" is composed by different "projects".

- Minecraft simulations

We will now describe what happens, step by step, from the moment the user clicks on "evolve" until they are presented with videos to choose, and what happens after they do choose a candidate to continue the process. We will also address how multiple users are treated to avoid conflicts.

4.1 Starting the path from click to video display

Many methods in the coordination project INM will require the userName. This is done by giving users a cookie with a number, which will be their ID for a month. See the classes `ExtensionHelper` and `CookiesCounter` for more details.

When a user clicks the button "Evolve", the method `StartEvolution()` in `CandidatesController` is triggered:

```
public ActionResult StartEvolution() {
    string userName = HttpContext.UserIdentity();
    WriteLineForDebug("Requesting to start evolution!", userName);
    if (EventsController.RaiseStartEvolutionEvent(userName))
    {
        return RedirectToAction("WaitingForEvolutionSetup");
    }
    else
    {
        return RedirectToAction("EvolutionBusy");
    }
}
```

This method calls
`EventsController.RaiseStartEvolutionEvent(userName)`
and returns
`RedirectToAction("WaitingForEvolutionSetup")`.

This sets two things in motion. On the one hand, it starts the process for a new evolution process. On the other hand, it redirects the user to a new page waiting until the process is done. We will cover both parts, which are related.

4.2 Redirect to waiting page

It is not straightforward to redirect the user to a new website page. Instead, we load the waiting page, and this page redirects automatically to a new page. The URL for the new page is returned by an "asynchronous" action:

```
<script type="text/javascript">
    window.location =
        "@Url.Action("WaitDuringEvolutionSetup", "Auxiliary")";
</script>
```

This action, in the class `AuxiliaryController`, will redirect to the new page when a process is done (in this case, when the evolution setup is ready).

The code in this class is a bit involved, but it is easy to understand that `WaitDuringEvolutionSetupCompleted(bool success)` returns the new URL, and this method is called if `WaitForSetup()` returns `true`. `WaitForSetup()` returns `false` after a predefined waiting time, or `true` if `IsUsersEvolutionReady` succeeds.

`IsUsersEvolutionReady`, like all the information relative to the user's status, is found in the `User` class, handled by `ActiveUsersList` (in project Evolution/UsersInfo). In the next section, we will see when and how this information gets updated for our new user.

If the process is successful, the URL will be for a new waiting page, in this case waiting for the candidate videos. This page works identically (automatically redirecting to a new page via an asynchronous action). The final result, if no problems are found, is the main evolution page, with three videos for the user to see. But many things need to happen first.

4.3 From RaiseStartEvolutionEvent to evolution ready

The flow of the code takes us now to the class `EventsController`. The first thing the method `RaiseStartEvolutionEvent` will do is to check if there are simulation slots available. `ActiveUsersList` contains users that are already active. If the new user is not there, `ActiveUsersList` first removes inactive users. Actions like "evolve" update the idle time of the given user, if a user has spent too much time without actions, they will be removed from the active users list at this point. If not too many users are active, then `RaiseStartEvolutionEvent` continues.

Then StartEvolutionEvent is launched, and this is received in `EvolutionCoordination.OnStartEvolution`, which calls `StartOrRestartEvolution`. When a user resets evolution or branches, the code will also get to this point.

Depending on whether the user has an evolutionary algorithm (again, `ActiveUsersList` has this information) we go to `LaunchEvolution` or `ReLaunchEvolution`. Let us look closer at the `LaunchEvolution` path (they are quite similar).

Now the process is very much plain NEAT for a while, but important things happen:

- `MyExperiment` **is created**. This inherits from `SimpleNeatExperiment` and decides the experiment parameters (notice that `EvolutionCoordinator` is passing the path for a Malmo.config.xml file to MyExperiment creator), as well as how genomes will be decoded into phenomes and, crucially, how they will be evaluated (in our case, using `OnlineMalmoPseudoEvaluator`, which in turn decides which controller will run Minecraft agents).

  Most importantly: in `MyExperiment` you should change the number of input and output neurons. In Malmo.config.xml you change the population size and number of species.

- `SimpleNeatExperiment` will create the `NeatEvolutionAlgorithm`, using for this the path for a possible saved population file (this used, for example, when branching, and this file is reset for evolution reset).

  Note: when the evolution algorithm is created it registers the user in `ActiveUsersList` (where there will be a reference linking the user with the evolution algorithm).

- `EvolutionCoordination` is subscribed to events in the evolution algorithm (used mainly to save the population after each generation).

- **A first module is added if not present**. Genomes in this project are compatible with a modular structure. When they are created they only have a basic scaffolding. Here they are provided with a standard module that connects all inputs and outputs (more technically: allows these connections).

When `Initialize` is complete, `LaunchEvolution` continues by marking the evolution as active in `ActiveUsersList`. Then evolution is properly started in `StartEvolution`, which calls `MakeEvolutionReady` in the evolutionary algorithm.

Here we can find the class `ProcessNewGeneration`. This class takes the population and assigns to each genome and index that will let us know where to display the video in the website (there are three positions). Currently, the video that was chosen at one iteration will be displayed in the same place in the next one, but there are other options, like all random or the one selected always on top, etc. Genomes are set as not evaluated so that we make sure that all videos are generated (except for the champion, because we already have that video, which only needs a new file name).

The user status in `ActiveUsersList` is updated, letting `EventsListenerController` know that evolution is ready.

This connects with the other process we have been following (see 4.2), where the user had been redirected to the waiting page and was waiting precisely for this update. Remember from the end of the previous subsection that this will take the user to the new waiting page, "waiting for candidate videos". `MakeEvolutionReady` now asks the evaluator to "evaluate" the genomes, which results in the creation of the videos. The user information is again updated letting the user, finally, get to the main evolution page.

4.4 From genome evaluation to video files

Different evaluators may be chosen in `SimpleNeatExperiment`. Some may parallelize evaluation of genomes. Others may skip re-evaluation of surviving genomes from previous generations (elites). In our case all, in the end, call `Evaluate` in `OnlineMalmoPseudoEvaluator`. This method takes a phenome (or "brain", the processed version of a genome) and creates for it an object from the class `ProgramMalmo`. `ProgramMalmo` will create the Minecraft simulation, and is able to pass commands to that simulation (so as to control it). Before starting the simulation (`ProgramMalmo.RunMalmo`) an agent controller is created. This takes the phenome and the `ProgramMalmo`, so that only the brain from one specific genome will interact with this particular Minecraft simulation.

Different Minecraft simulations are identified by a port. Trying to create a new simulation in a used port will result in an error, and this specially delicate if we allow for parallel evaluation of the genomes. `userToPort` dictionary helps to handle this.

`UpdateUserToPortDictionary` assigns a free port to the new user. This port is the first in the range assigned for the user.

`ProgramMalmo.RunMalmo` takes also an offset value to know exactly which port in the user's range should be used for the given simulation (this is currently assigned in `ParallelGenomeListEvaluator`). The port is then used to initialize a `clientPool`, which is how `ProgramMalmo` identifies the correct Minecraft simulation to use.

We will only describe now the steps that are more relevant for the user. For more details, go to the Project Malmo[6] documentation.

`ProgramMalmo` can create events that will be received by the associated agent controller. Now, it creates an event that will have the phenome used by the controller reset (meaning the activation values of the neurones are set to zero).

Then the "mission" is started. This will load a "minecraftWorldXML.txt" file, which contains lots of information. See the current file as an example:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Mission xmlns="http://ProjectMalmo.microsoft.com" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
  <About>
    <Summary>Great mission description!</Summary>
  </About>
  <ServerSection>
    <ServerInitialConditions>
      <Time>
        <StartTime>1000</StartTime>
        <AllowPassageOfTime>false</AllowPassageOfTime>
      </Time>
    </ServerInitialConditions>
    <ServerHandlers>
      <FlatWorldGenerator generatorString="3;7,45*48;,biome_1"/>
      <ServerQuitFromTimeUp timeLimitMs="3000"/>
      <ServerQuitWhenAnyAgentFinishes/>
```

[6]https://www.microsoft.com/en-us/research/project/project-malmo/

```
  </ServerHandlers>
</ServerSection>
<AgentSection mode="Creative">
  <Name>Paco</Name>
  <AgentStart>
    <Placement x="0.5" y="46.0" z="0.5" pitch="60" yaw="0"/>
    <Inventory>
      <InventoryItem slot="0" type="dirt"/>
    </Inventory>
  </AgentStart>
  <AgentHandlers>
    <ChatCommands/>
    <ObservationFromFullStats/>
    <ObservationFromGrid>
      <Grid name="level0">
        <min x="-1" y="0" z="-1"/>
        <max x="1" y="0" z="1"/>
      </Grid>
      <Grid name="levelSub1">
        <min x="-1" y="-1" z="-1"/>
        <max x="1" y="-1" z="1"/>
      </Grid>
      <Grid name="levelSub2">
        <min x="-1" y="-2" z="-1"/>
        <max x="1" y="-2" z="1"/>
      </Grid>
    </ObservationFromGrid>
    <DiscreteMovementCommands/>
    <VideoProducer
     viewpoint="1">
        <Width>432</Width>
        <Height>240</Height>
    </VideoProducer>
  </AgentHandlers>
  </AgentSection>
</Mission>
```

Here we are giving a summary of the mission. Then the time of day is set, and we block the pass of time. The basic terrain is generated and a maximum simulation time is setup. Creative mode is chose, and the agent, Paco, is placed in a specific place with a given inventory. Chat commands are allowed and will be used to give orders to the agent, which will produce "full stats" as feedback. On top of that, we create three grids, which will return the block types of all blocks in them. Finally, discrete movement is set (only one-block movement and 90-degree rotation).

Note we can change more things after loading. `mission.timeLimitInSeconds(10);`, for example, sets a new time limit. Here `AddProceduralDecoration` is called, and will add new blocks to the terrain. See `MazeMaker` for more details.

Starting the mission has some potential for conflicts with multiple users. We tried to minimize this by using try/catch structures.

The mission loop asks the agent to get feedback. These are given in Json format, and are translated into normal types (stored in the `JsonObservations` class). An observations event is created. Any commands received are sent to the agent (and then the list of commands is reset). The thread sleeps for an interval and the process is repeated. A video of the simulation is being created in the meantime.

The agent controller class receives these "observation events", and takes the world feedback as parameters. This information is used to update the inputs of the neural network, which is then "activated" (information flows from input to outputs) and the outputs are translated into commands for the agent, which are passed to `ProgramMalmo`. See this minimal example:

```
void WhenObservationsEvent(
        object sender, ObservationEventArgs malmoInfo)
{
    brain.InputSignalArray[0] = malmoInfo.DamageTaken;
    brain.InputSignalArray[1] = malmoInfo.XPos;
    brain.Activate();
    if (brain.OutputSignalArray[0] > actionThreshold)
        programMalmo.AddCommand("move 1");
}
```

When `ProgramMalmo` ends, the "evaluation" of the genome finishes. `OnlineMalmoPseudoEvaluator`, which has access to `ProgramMalmo`, may return now a fitness value, which the evaluator (e.g., `SerialGenomeListEvaluator`) uses to update the genome's fitness (in our case, in the method `EvaluateOne`).

Before this happens, `ProgramMalmo` asks to create the candidate video, and this is done by the important static class `PopulationReadWrite`. This process involves extracting the compress save files and then renaming and moving the video file to the correct folder for this user's candidate videos.