

CE318: Games Console Programming

Lab 8: Designing behaviours with Behaviour Trees

2nd December 2011

Behaviour Trees

- First, go to the next pages to download the following files:
 - Go to <http://www.gamebrainsai.com> to download the Behaviour Tree editor. You will need to register and accept a confirmation email to be able to download the free version.
 - Go to the course webpage to download the provided code for this lab.
- Open the Behaviour tree editor (`gamebrains.bt_library/BT Editor/bin/BTEditor.exe`).
 - From the BT Editor, select *File* → *Open* → *OpenTree*, and open the *XML* file that contains the base behaviour that you will use to start this lab. It is located in the files you downloaded for the lab, in `3DMazeShooter/3DMazeShooterContent/enemyBehaviour.xml`.
 - You should see now a behaviour tree depicted in the center of the application. You can interact with the nodes clicking on them, or right-clicking to open the contextual menu. By doing this, you can add and delete nodes, change the order of them among its siblings, etc.
 - In this lab, you will only need to create Actions, Conditions, Selectors and/or Sequence nodes. The tool allows you to create also filters, but those nodes are not supported by the *C#* library (included in the code for this lab). It is up to you if you want to enhance the library to support this nodes, but they won't be needed for this exercise.
 - * **Note:** In this editor, Selectors and Sequences are not depicted with the usual notation ('?' and '→' respectively). Instead, Selector are nodes coloured in **Blue** while Sequences are **Green**.
 - In order to add a new Action or Condition, you must do right click on a Selector or a Sequence node, and select the option 'Add Action' or 'Add Condition' accordingly. Then, it is important that you give it a name and an operation. You must do that by writing it in the right panel, where the properties of the select node appear. Please, pay special attention to the field **Operation**, as this one is used by the *C#* library to read and create the tree in memory.
 - Remember to save the tree when you make modifications. The *C#* library is able to read this tree in the format saved by this tool. If you keep the same name of the file, you won't even have to modify the code to read the proper behaviour tree file.
- These are the steps you have to follow every time you create a new leaf node in the tree:
 - Every new Action and Condition that you create in the tree must have a correspondent class in the library. A suggested place to locate these classes is where the class `XMLReader.cs` is (in `3DMazeShooter/MapGameLibrary/level/entities/leafNodes/`). The class should have the same name as the one you wrote in the **Operation** field in the BT Editor (this is not mandatory, but doing this will help you to have an easy-to-understand code).
 - These classes have to inherit from the class `BTLeafNode`, and override the method `step`. You will have to write in this method the logic you want for this node. In this method, you have to:
 - * Make a call to `base.step()` at the beginning of the function.
 - * Assign to the variable `m_nodeStatus` the result of the node. This can be either `BTConstants.NODE_STATUS_SUCCESS` or `BTConstants.NODE_STATUS_FAILURE`.
 - * **Only** in the case your node is a condition, you must end the method reporting the result to this node's parent. You can do this by doing the call `m_parent.update(m_nodeStatus);`.

- For examples of Actions and Conditions, you are encouraged to see the classes `IsFollowingPath` and `FollowPath`, in `3DMazeShooter/MapGameLibrary/level/entities/Leaf nodes/`.
- Finally, you must include your new class in the importer, located in the class `XMLReader` (also in `3DMazeShooter/MapGameLibrary/level/entities/Leaf nodes/`). In this class, in the method `readNode()`, the `Operation` field is read and used to create the appropriate class.
- The behaviour tree provided as an example makes the enemy to wander around the level. The objective of this lab is to enhance this behaviour so it performs the following actions:
 - If the enemy sees the flag, it has to calculate a path (using A*) to its location and pick it up (the enemy will pick it up automatically when they collide).
 - If the enemy sees the player, it should shoot him. You can use the function `Attack()`, in `class Enemy`, that targets and shoots bullets to the enemy.
 - You are also asked to decide which one of these two behaviours (attack or pick up the flag) has a higher priority, what must be specified in the design of your behaviour tree.
 - If no enemy or flag is visible, then the Enemy has to wander around the level as it does in the sample behaviour provided (get a random node in the level, calculate the path to that position and follow it). Obviously, it has to stop wandering (and interrupt the current followed path) if the enemy or the flag are seen.
 - Other considerations about the gameplay are up to you. For instance, some ideas are: what happens when the flag is picked up? It could be re-spawned in a different location, or dropped by the carrier when it is shot. You also could actually kill the enemy or the ship when it reaches a certain damage, and then re-spawn it somewhere in the map.
- The overall idea is that the new leaf nodes you have to create are going to use functions from `class Enemy`, like `getFlagPath()`, `Attack()` and `IsFlagVisible()`.
- Finally, some hints about some important classes provided for this lab:
 - `Enemy` is the class that encapsulates the enemy's behaviour. It has a reference to a `BehaviourTree` object. It creates the tree in the constructor of the class and calls the method `BehaviourTree.execute()` from `Update()`, to actually execute the behaviour tree.
 - `Ship` is the class for the player.
 - `Level` encapsulates all the logic for the level creation, as well as some other useful functions like collision detection and ray tracing.