Building Player Tasks - Documentation & Reference Manual

by William Chilcote

Contents

Primary Principles

Objective progression in the Task System is organized through a hierarchy of objective types. At the highest level of the hierarchy are *Level* objects, which each contain a set of *Tasks*. Each Task can also contain a set of *Steps*.

Levels should contain tasks that are both largely related to each other, and are also unrelated in some way to tasks in other levels. While Tasks and Levels can be organized arbitrarily, an example of an effective use of their structure would be to put Tasks from one primary location into one Level, while the following Level would contain Tasks from another location.

Tasks represent short objectives that the player has to complete. Although Tasks can contain an essentially unlimited number of Steps, they are intended to have only a maximum of a few that can be completed in quick succession. An example of a Task could be “get this item and bring it to this location.” Tasks are what the player is intended to see when looking at their current or completed objectives.

Steps are the foundation of the Task System that all other objective objects are comprised of. Steps represent the specific objective that the player needs to complete in order to progress. Common uses of Steps include “get this many of this item,” “talk to this person,” or “move to this location.”

Steps contain two integer values, one representing current progress, and one representing the target progress needed for completion. Target process is often 1, but can also be greater, such as when a specific number of items is required to complete the Step. When the current progress is equal to the target progress, the Step is considered complete.

Overall progress for a Step is the ratio of current progress to target process, between `0` and `1`. The progress of a Task is the average progress value of its contained Steps, and the progress of a Level is the average progress value of its contained Tasks. When any level of the objective hierarchy is completed, the next one is automatically started. Progress toward any Step cannot begin until all previous Steps have been completed.

Level XML Formatting

The organization of all Tasks and Steps contained in each Level is stored in an XML file. The root node must be named `level`, and have the attributes `name`, and `scene`. The `name` attribute contains a string representing the name of the Level as it will be displayed in the game. The `scene` attribute contains a string representing the name of the scene that will be loaded when this level is selected from the level select page. Note that the scene must be included in the Build Settings scene list to be loaded.

All Tasks in the Level are stored as sibling `task` nodes under the root `level` node. The first child node of each task node should be named `name`, and contain a string representing the name of the Task as it will be displayed in the game. The second child node should be named `description`, and contain a string representing the description of the task as it will be displayed in the game. The third child node should be named `steps`, and should contain a child node for each contained Step within the Task.

Another type of node that is located at the same level as `task` nodes is `move`. This node will teleport the player to the given coordinates (`x` and `y`) when they have finished the task defined in the previous node. If it is at the start of the the `level` child nodes, the move will be made when the player begins the level. This can be used for defining player positions between levels if the same scene is used multiples times, as well as for quick travel between tasks.

The fourth and final child node is the optional `completion` node. This node contains items that can either be given to, or taken from the player upon their completion of the Task. To give the player items, use the `give` node. To take items from the player, use the `take` node. Both nodes should be placed as children of the `completion` node. They both have a required first attribute `name`, that contains a string representing the name of the item to give or take. They also both have a second optional attribute `count`, which represents the number of this item to either give to or take from the player. If the `count` attribute it omitted, its value will default to `1`.

An example of proper formatting can be seen below.

```
<level name=“Example" scene=“Level One”>
  <move x="10" y="20"/>
  <task>
    <name>First Task</name>
    <description>Do this!</description>
    <steps>
      <manual/>
    </steps>
    <completion>
      <give name="Pencil" count="2"/>
    </completion>
  </task>
  <task>
    <name>Second Task</name>
    <description>Do this too!</description>
    <steps>
      <manual/>
      <manual/>
    </steps>
  </task>
</level>
```

Step Types

There are four different types of Steps that can be used in a Task. While at the lowest level, all Steps are of the same Manual type, the other Step types have additional built-in functionalities that allow them to automatically detect when progress has been made toward their completion.

Location

The Location Step type is completed when the player enters a given collider in the scene. To represent the Location Step in the Level XML, the node must be named `location`, and have the attribute `name`. The `name` attribute must contain a string representing the path of the `GameObject` with the attached collider that the player must enter to complete the Step. The format of the path of this GameObject is the same as paths used with `UnityEngine.GameObject.Find`. An example of the XML node for this Step can be seen below.

```
<location name="/Doors/Door 2"/>
```

Manual

The Manual Step type is completed when `Advance` is called on the instance of the `Step` class. As a shortcut, the static method `Tasks.Advance` can be called to call `Advance` on the currently active `Step` instance. Although these functions can technically be called on any Step type to advance its progress, it is recommended to only use it on Manual Step types, as most others evaluate their progress automatically. The XML node for this Step type accepts the optional attribute `count`, containing an integer representing the number of times `Tasks.Advance` must be called in order to complete this task. If the `count` node is omitted, its value will default to `1`. An example of the XML node for this Step can be seen below.

```
<manual count="2"/>
```

Item

Note: This Step type makes heavy use of the Inventory System. For more information, see the included documentation, 'Item Creation & Modifying Player Inventory'.

The Item Step type is completed when the player has the specified number of the specified type of item in their inventory. The XML node for this Step type must be named `item`, and have attribute `name`, containing a string representing the name of the desired item. The node also accepts an optional attribute, `count`, containing an integer representing the number of the specified item that the player needs to possess before the Step can be completed. If the `count` attribute is omitted, its value will default to `1`. As soon as the player has the specified number of the specified item in their inventory, the task will automatically be completed. An example of the XML node for this Step type can be seen below.

```
<item name="Pencil" count="2"/>
```

Dialogue

Note: This Step type makes heavy use of the Dialogue System. For more information, see the included documentation, 'Building NPC Dialogue - Documentation & Reference Manual'.

The Dialogue Step type is completed when the player accesses a portion of dialogue that utilizes the Dialogue System's `advance` node. When reaching the `advance` node in any portion of dialogue, the player's progress will be incremented independent of whether or not they are actually on the desired task. Therefore, the current task should first be checked in the relevant dialogue before advancing player progress. The XML node for this Step type accepts no attributes, and is self-closing as can be seen in the example below.

```
<dialogue/>
```

System Setup

The XML file for each Level desired in the game should be added to custom editor, found in Window > Level XML. Level files should be added to the list in the same order as they are desired to appear chronologically in your game. This editor generates an asset that must be named "`LevelXML.asset`" and be stored in the top level of a "`Pencil`" directory. The objective hierarchy will automatically be generated at runtime. An instance of `PlayerControl` must be attached to the player, as it makes use of the player's Collider to detect Trigger entries to automatically advance the progress of certain location-based Step types.

The `Tasks` class has several static properties that can freely be used in other scripts. `Tasks.allLevels` contains a list of all loaded Level objects, and progress relative to the player. `Tasks.currentLevel` will return the latest `Level` containing any incomplete Steps or Tasks. Similarly, the `Level` class has a property `currentTask` that will return the latest incomplete `Task`, and the `Task` class has a property `currentStep` that will return the latest incomplete `Step`. `Level`, `Task`, and `Step` all have the property `progress` that represents the decimal value of their completion between `0` and `1`.