



User Guide

v1.2

Quest Machine
Copyright © Pixel Crushers. All rights reserved.

Demo level layout by Walker Boys Studio.

Deepest thanks to all the beta testers who helped improve Quest Machine so much!

Contents

| | |
|--|----|
| Chapter 1: Welcome to Quest Machine..... | 4 |
| Chapter 2: Getting Started..... | 5 |
| Chapter 3: Quest Editor..... | 18 |
| Chapter 4: Quest Scene Setup & Management..... | 35 |
| Chapter 5: Quest UIs..... | 43 |
| Chapter 6: Quest Generation..... | 47 |
| Chapter 7: Scripting..... | 60 |
| Chapter 8: Third Party Integration..... | 66 |
| Chapter 9: Multiplayer & NPC Questers..... | 67 |
| Appendix 1: Localization and Text Tables..... | 68 |
| Appendix 2: Save System..... | 68 |



Chapter 1: Welcome to Quest Machine

Welcome to Quest Machine!

Quest Machine is a quest system for Unity. You can use it to add hand-written and procedurally-generated quests (also known as missions or objectives) to your projects. Quest Machine works great in 2D, 3D, VR, and AR.

Quest Machine lets you:

- Write hand-written quests
- Dynamically generate new quests based on the current world state
- Design and manage quest-related user interfaces
- Tie gameplay to quests
- Save & load quests

All without requiring any scripting!

It also has a well-organized API and ample code hooks to make it easy for programmers to extend its functionality.

In This Manual

The rest of this manual contains these chapters:

- **Chapter 2: Getting Started** – *Installation, demo scene, and quick start*
- **Chapter 3: Quest Editor** – *How to write quests*
- **Chapter 4: Quest Scene Setup & Management** – *How to set up scenes and control quests*
- **Chapter 5: Quest UIs** – *How the UIs work and how to customize them*
- **Chapter 6: Quest Generation** – *How to procedurally generate quests*
- **Chapter 7: Scripting** – *Programming reference*
- **Chapter 8: Third Party Integration** – *How to integrate with other assets*
- **Chapter 9: Multiplayer & NPC Questers** – *Considerations for multiplayer games*

How to Get Help

It's dangerous to go alone!

We're here to help! If you get stuck or have any questions, please contact us any time at support@pixelcrushers.com or visit <https://pixelcrushers.com>.

We do our very best to reply to all emails within 24 hours. If you haven't received a reply within 24 hours, please check your spam folder.

Chapter 2: Getting Started

Setup

Quest Machine requires Unity 2018.4+. To get started, open the Asset Store window and import Quest Machine. This will import these folders into your project:

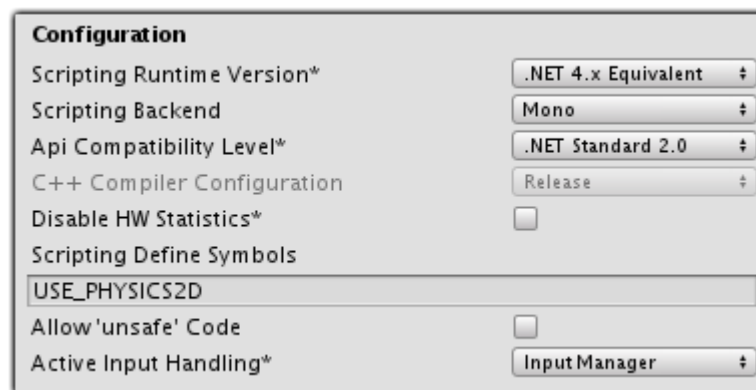
- **Plugins ► Pixel Crushers ► Quest Machine:** The Quest Machine product
- **Plugins ► Pixel Crushers ► Common:** Core scripts shared by all Pixel Crushers products

The first time you import Quest Machine, it will ask if you want to add standard input definitions to detect when the player is using a joystick. Click **Yes**.

Enabling 2D Physics Support

Quest Machine supports 2D physics as well as 3D physics. In Unity projects, the 2D physics package (Physics2D) can be enabled or disabled, so Quest Machine's code doesn't assume that Physics2D is available in your project. To tell Quest Machine that Physics2D is available, select menu item **Tools ► Pixel Crushers ► Common ► Misc ► Enable Physics2D Support...** or tick the USE_PHYSICS2D checkbox in the Welcome Window.

If you want to manually enable Physics2D support instead, select **Edit ► Project Settings ► Player**, and add the scripting symbol USE_PHYSICS2D as shown below:



Quest Machine's example scene is in 2D. You will need to enable the Physics2D package and enable Quest Machine support for it.

Video Tutorials

If you prefer video, you can watch Quest Machine's [Video Tutorial Series](#). This manual contains additional information not covered in the video tutorials. If you have a question, refer to the manual or contact us.

Demo Scene

The demo scene is in Plugins ► Pixel Crushers ► Quest Machine ► Demo. Play it to see how Quest Machine works. Remember to [set the USE_PHYSICS2D symbol](#).



How to Play

- Use the arrow keys, WASD keys, or joystick to move.
- Click the left mouse button or joystick button A to swing your sword.
- Click the right mouse button or joystick button B to interact with things in the game world.
- Press the joystick Back button to open the quest journal and the Start button to open the menu.

Demo Quests

The demo scene contains these quests:

- **Pesky Rabbits** (Johann the Farmer): A single-time hand-written quest.
- **Harvest Carrots** (Johann the Farmer): A repeatable hand-written quest.
- **Food Delivery** (Johann the Farmer): A longer hand-written quest with a prerequisite.
- **Coin Race**: (Captain Molly): A timed quest.
- Plus some number of procedurally-generated quests depending on the state of the game world. Sir Goodwin and Captain Molly are both concerned about the safety of the village. Sir Goodwin is not above slayings his enemies, but Captain Molly will usually seek a less violent alternative. Unless you've changed the parameters of the spawners in the demo scene, these quests will usually be generated:
 - **Kill *n* Orcs**: (Sir Goodwin): Generated if he detects Orcs in the Forest.
 - **Polymorph *n* Orcs**: (Captain Molly): Generated if she detects Orcs in the Forest.

The next section presents an overview of how Quest Machine works. If you want to jump straight into making quests, you can skip to the [Quick Start](#) section.

Overview

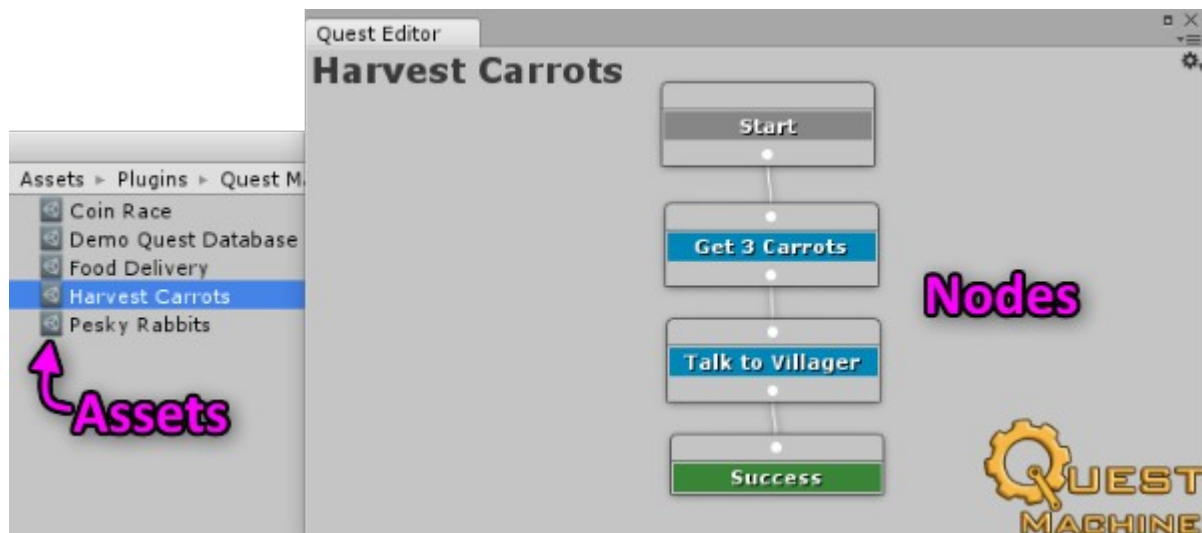
This section presents a bird's eye view of Quest Machine.

Quest Machine consists of these main parts:

- **Quest Asset:** Contains the complete definition for a quest.
- **Quest Editor:** Lets you create hand-written quests and view runtime quest states.
- **Quest Execution:** Manages quests and user interfaces at runtime.
- **Quest Generation:** Dynamically generates new quests based on the state of the game world.

Quests

Quest Machine's fundamental asset is the *quest*. A quest is a task that a character may complete, usually to gain a reward. This often involves several subtasks, called *quest nodes*, each with their own completion requirements.



Internally, quests are ScriptableObjects. Hand written quests are stored as asset files in your project. Procedurally generated quests are stored in memory.

Quests consist of:

- **Quest Info:** Global information about the quest.
- **Start Data:** Defines how the quest starts.
- **States:** Information about the overall states that the entire quest may be in.
- **Counters:** Numeric variables that the quest can use to keep track of progress.
- **Nodes:** Each node consists of:
 - **Main Node Data:** Information about this node.
 - **Node States:** Information about the states that this node may be in.
 - **Conditions:** Requirements that must be met for the quest to progress to the next node.

Quest Editor



Use the Quest Editor to design hand-written quests and view the states of hand-written and procedurally-generated quests at runtime. For detailed information about the Quest Editor, see [Chapter 3: Quest Editor](#).

Quest UIs



Quest Machine manages these quest-related UIs:

- **Dialogue:** Used when talking with NPCs to accept and complete quests.
- **Journal:** Shows the player's active and completed quests.
- **HUD:** Shows tracking information for active quests.
- **Alert:** Shows pop-up messages.
- **Indicators:** Typically shown over NPCs' heads to indicate quest-related activity.

For detailed information about quest UIs, see [Chapter 4: Quest UIs](#).

Quest Generator



Quest Machine's quest generation features let you annotate your game world with information used to procedurally generate quests. For detailed information about quest generation, see [Chapter 6: Quest Generation](#).

Message System

Quest Machine objects use a *message system* to efficiently communicate activity such as changes in counter values and quest states.



You don't have to write any code to use the message system. Quest Machine includes inspectors to send and listen for messages.

Localization

All text fields are localizable. For information about localization, see [Appendix 1: Localization and Text Tables](#).

Extending Quest Machine with Custom Scripts

While Quest Machine's default capabilities are quite powerful and flexible, you can create your own conditions, actions, reward systems, UI content types, and more. This is covered in [Chapter 7: Scripting](#).

Quick Start

We're here to help! If you get stuck at any point, please [contact us](#).

This section jumps right into using Quest Machine to create quests. You can find more detailed steps in the chapters following this Quick Start section.

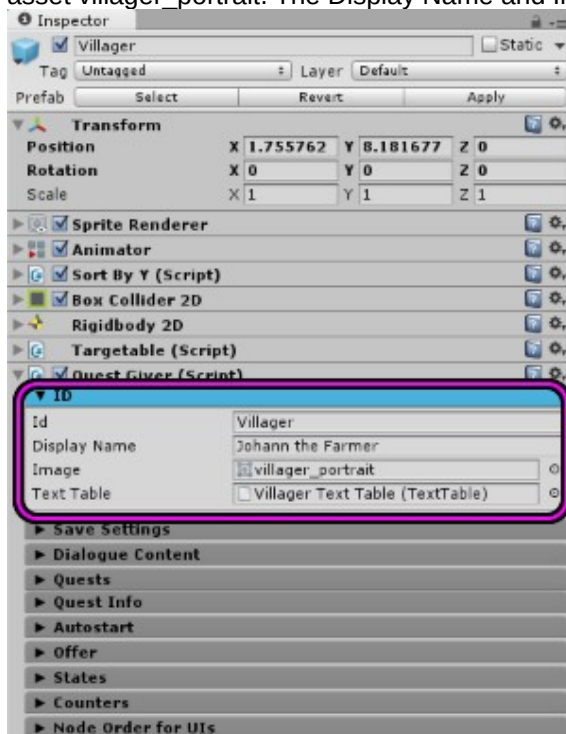
Open Quick Start Scene

Navigate to the folder Pixel Crushers ► Quest Machine ► Demo and open the Quick Start scene. This scene is set up like the Demo scene except without any quests. We've already added the Quest Machine and Input Device Manager prefabs, as well as a gameplay menu and the player. This quick start jumps straight to writing a quest. (Scene setup is covered in the [Scene Setup](#) chapter.)

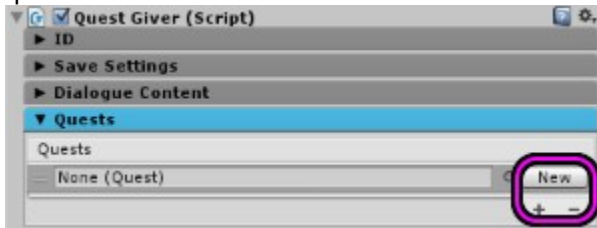
Write A Hand-Written Quest

Orcs stole Johann the Farmer's money and hid it in crates and barrels around the village. Johann needs to ask the player to retrieve his money so he can buy seeds for next year's crops. In this tutorial, we'll create a quest to help Johann. The quest will count the number of coins that the player has picked up. When the player has picked up the required amount, the quest will end. To set up the quest, we'll specify text content in a few areas (dialogue, journal, and HUD), add a simple condition that counts coins, and add it to a new quest database asset. Quest Machine has wizards to simplify these steps, but we'll do it manually to provide a deeper introduction to the editor. Then we'll configure Johann to offer the quest.

1. In the Hierarchy, select the GameObject named **Villager**.
2. (Add a **Quest Giver** component.) This step has already been done in the Quick Start scene.
3. In the ID section, set **ID** to "Villager", **Display Name** to "Johann the Farmer", and **Image** to the asset villager_portrait. The Display Name and Image are shown in runtime UIs.

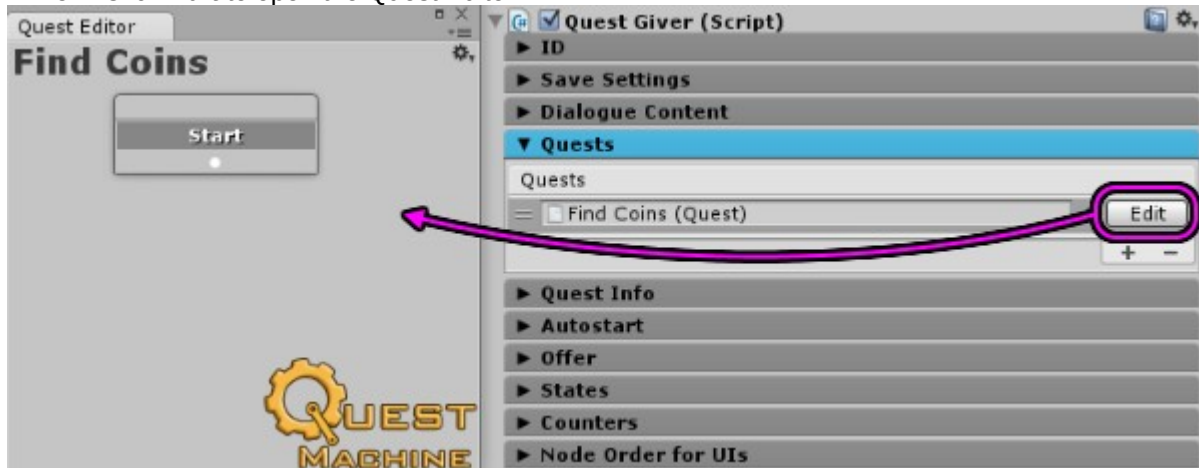


4. In the Quests section, click “+” to add a new slot, then click **New** to create a new quest in the slot.

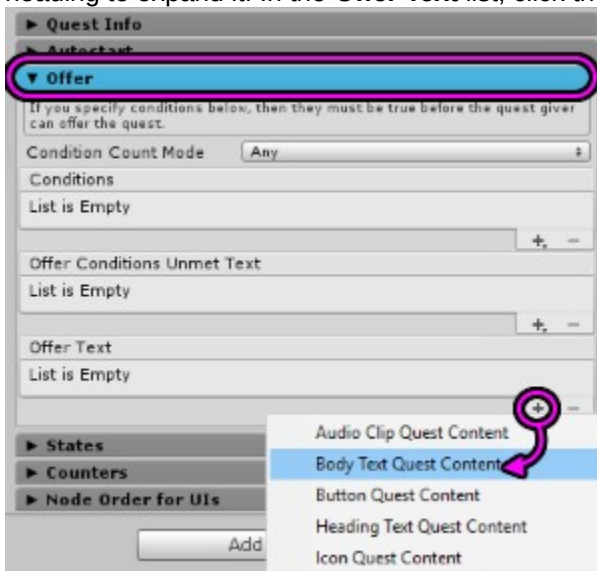


Save the new quest as “Find Coins”.

5. Click **Edit** to open the Quest Editor:



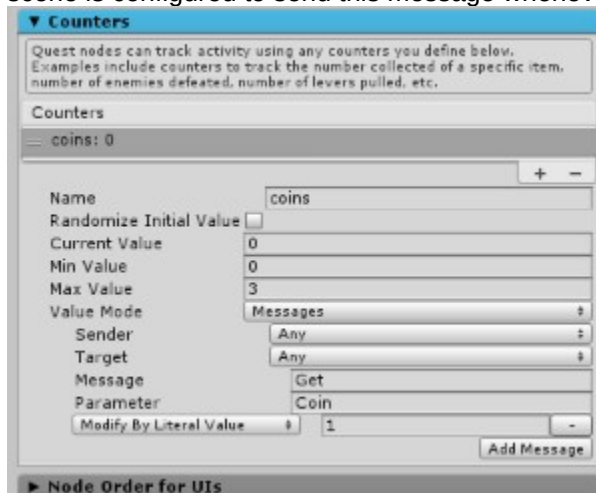
6. In the Quest Editor, click on empty canvas to inspect the quest's main properties. Click the **Offer** heading to expand it. In the **Offer Text** list, click the “+” and select **Body Text Quest Content**:



7. This will add a Body Text element to the **Offer Text** list. In Quest Machine, most text elements have three fields: a regular string, String Asset, and Text Table. The latter two allow you to assign text from assets instead of adding text directly into the quest; they're covered in more detail in the Quest Editor chapter. For now we'll use the regular string field. Enter the quest text:



8. We'll define a counter to keep track of how many coins the player has found.
 - a) In the **Counters** section, click "+" to add a new counter.
 - b) Set **Name** to "coins".
 - c) Set **Max Value** to 3.
 - d) We'll let the Message System tell us when the player picks up a coin. Set **Value Mode** to **Messages** and click the **Add Message** button. Then set **Message** to "Get" and **Parameter** to "Coin". (Messages are case-sensitive, so use the exact spelling and capitalization shown.) When the quest receives this message, it will modify the value of coins by +1. The quick start scene is configured to send this message whenever the player picks up a coin.

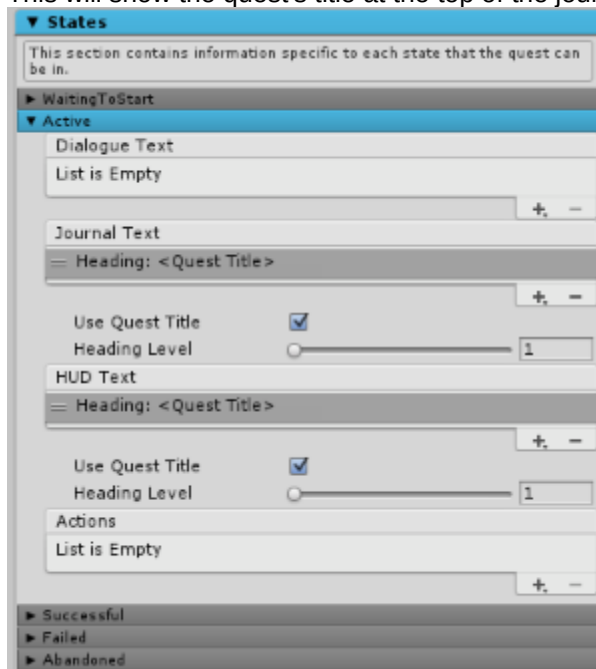


Note: We use messages as a simple way to count coins in this demo. In the demo scene, crates and barrels are configured to send a "Get" "Coin" message when broken. If you use a third party inventory system after completing this tutorial, you can use or create an integration script to check it instead of using messages.

9. In the Quest Editor, right-click on the **Start** node and select **New Node** → **Condition**. Then right-click on the **Condition 1** node and select **New Node** → **Success**:



10. Now let's add the quest text. When Quest Machine puts together the content for a UI, it first uses the content defined for the current main quest state. Then it adds the content defined for each node's state. Click on blank canvas area to inspect the main quest settings. Expand **States** → **Active**. In the **Journal Text** section, add a **Heading Text Quest Content**. Tick **Use Quest Title**. This will show the quest's title at the top of the journal entry. Do the same for **HUD Text**.



▼ States

This section contains information specific to each state that the quest can be in.

► WaitingToStart

▼ **Active**

Dialogue Text

List is Empty

Journal Text

— Heading: <Quest Title>

Use Quest Title ☒

Heading Level

HUD Text

— Heading: <Quest Title>

Use Quest Title ☒

Heading Level

Actions

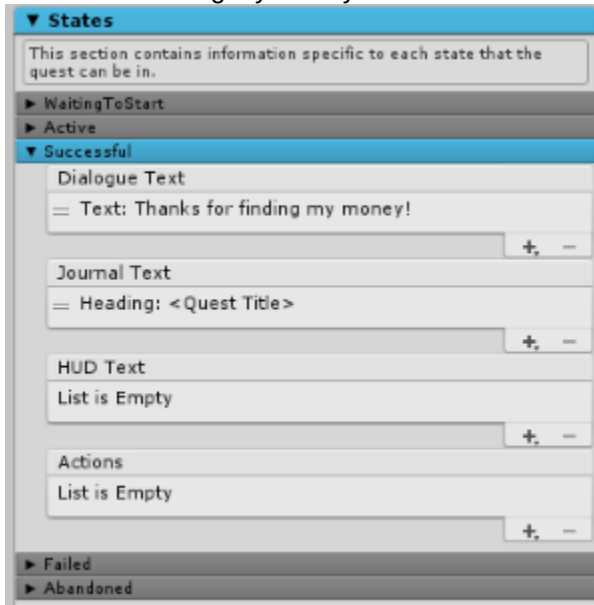
List is Empty

► Successful

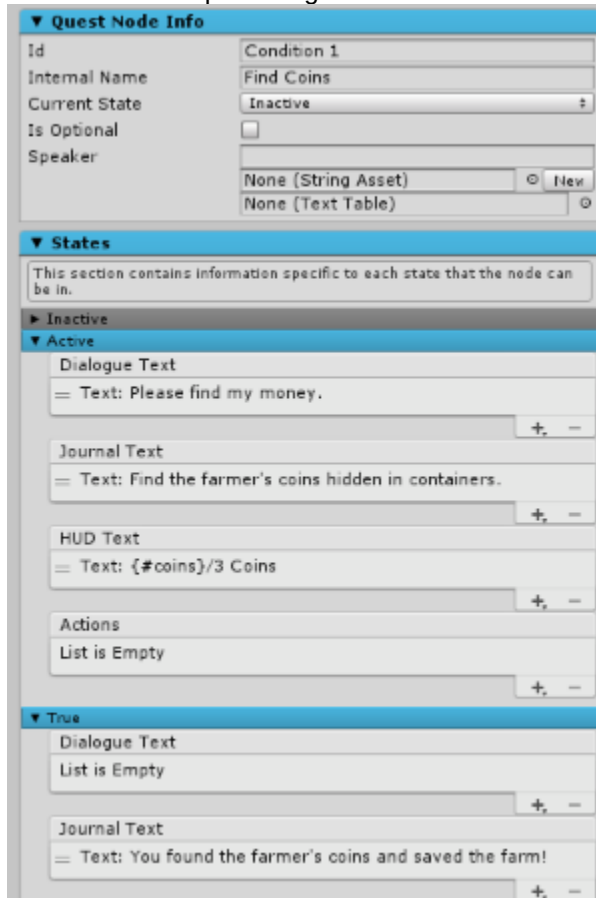
► Failed

► Abandoned

11. In the **Successful** state (which the player can see after completing the quest), set the **Dialogue Text** to something congratulatory, and add a quest heading to the **Journal Text**. In the screenshot below, we clicked the “+” button and selected **Body Text Quest Content** to add body text. If the player talks to the quest giver after completing this quest, the quest giver will say “Thanks for finding my money!”



12. Click on the **Condition 1** node. In **Quest Node Info**, set **Internal Name** to “Find Coins”. This is only visible in the editor, but it reminds you what the purpose of the node is. Then in the **States** section set the **Active** and **True** content to the body text shown below. (When you click on a list item, it will open a larger text area in which you can enter the text.) *Active* content is shown when this node is active. *True* content is shown when this node’s conditions have become true. Note that we’re using a special tag “{#coins}” to reference the value of the coins counter in the HUD Text . Special tags like these are covered in detail in the Quest Editor chapter.



▼ Quest Node Info

Id: Condition 1

Internal Name: Find Coins

Current State: Inactive

Is Optional: ☐

Speaker: None (String Asset) None (Text Table)

▼ States

This section contains information specific to each state that the node can be in.

▶ Inactive

▼ Active

Dialogue Text
= Text: Please find my money.

Journal Text
= Text: Find the farmer's coins hidden in containers.

HUD Text
= Text: {#coins}/3 Coins

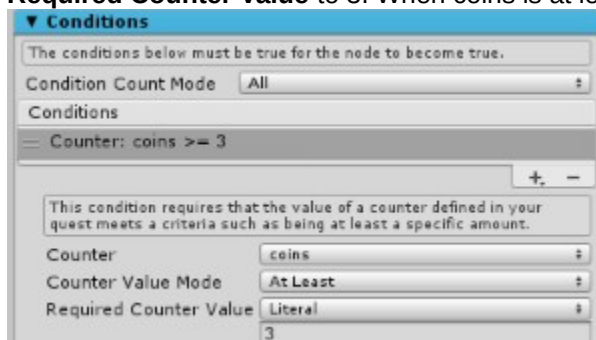
Actions
List is Empty

▼ True

Dialogue Text
List is Empty

Journal Text
= Text: You found the farmer's coins and saved the farm!

13. In the node’s **Conditions** section, add a **Counter Quest Condition**. Since our quest only has one counter, the **Counter** dropdown defaults to “coins”. The counter needs to be at least 3, so set **Required Counter Value** to 3. When coins is at least 3, this node’s state will become true.



▼ Conditions

The conditions below must be true for the node to become true.

Condition Count Mode: All

Conditions

= Counter: coins >= 3

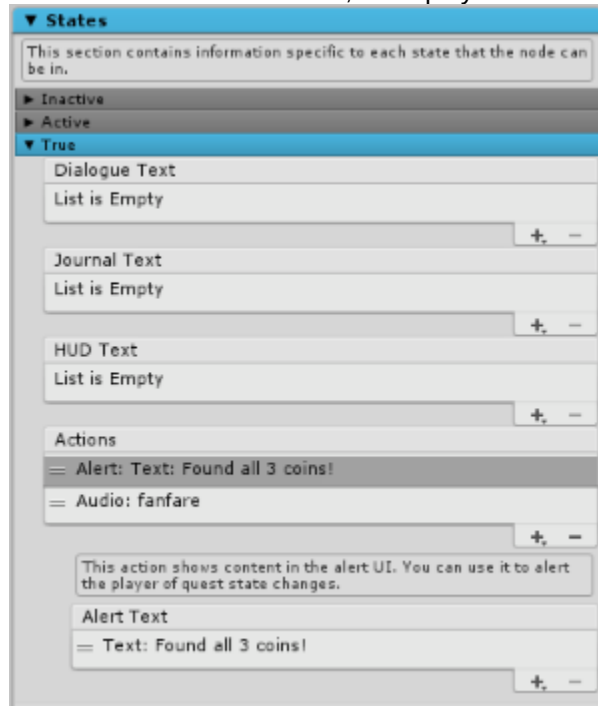
This condition requires that the value of a counter defined in your quest meets a criteria such as being at least a specific amount.

Counter: coins

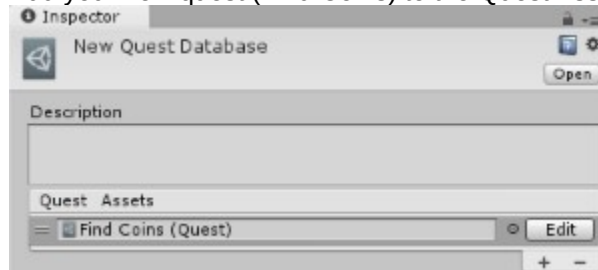
Counter Value Mode: At Least

Required Counter Value: 3

14. Click on the **Success** node. Success and Failure nodes become true as soon as they go active. They set the main quest state to *Success* or *Failure*. Let's show an alert message when the player finds the third coin.
 - a) In the **States** → **True** section, add an **Alert Quest Action** to the **Actions** list.
 - b) In the **Alert Text** list, add a **Body Text Quest Content** set to "Found all 3 coins!"
 - c) Add an **Audio Quest Action** to the **Actions** list, and assign the fanfare audio clip. When the Success node becomes true, it will play a fanfare to congratulate the player.



15. We're almost done! The last step is to add this quest to a *quest database*. The database gives Quest Machine access to your quests even if the quest giver is no longer present in the scene.
 - a) Right-click in the Project view and select **Create** → **Pixel Crushers** → **Quest Machine** → **Quest Database**.
 - b) Add your new quest (Find Coins) to the Quest Assets list.



- c) In the hierarchy, inspect **Quest Machine**. Add your new quest database to the **Quest Databases** list.

Now test your first quest!

Chapter 3: Quest Editor

This chapter describes how to use the Quest Editor. To open the Quest Editor, use menu item **Tools** → **Pixel Crushers** → **Quest Machine** → **Quest Editor**. The Quest Editor window works in conjunction with the Inspector view.



Navigating the Quest Editor

These inputs manipulate the canvas:

| Input | Alternate | Function |
|-------------|-----------|--------------|
| LMB | | Select |
| RMB | Ctrl+LMB | Context menu |
| MMB drag | Alt + LMB | Pan canvas |
| Mouse wheel | Gear menu | Zoom |

Gear Menu

The gear menu in the upper right has these menu options:

- **Pan** → **Top Left**: Moves to the top left of the canvas.
- **Zoom** → various zoom options: Changes the zoom level.
- **Text** → **Tags to Text Table**: Populates a text table with any tags referenced in the quest.

Context Menu

If you right-click on blank canvas or a quest node, it will open a context menu. At runtime, the context menu also allows you to set the states of quest nodes.

Creating a New Quest

To create a new quest, right-click in the Project view and select **Create** → **Pixel Crushers** → **Quest Machine** → **Quest**.

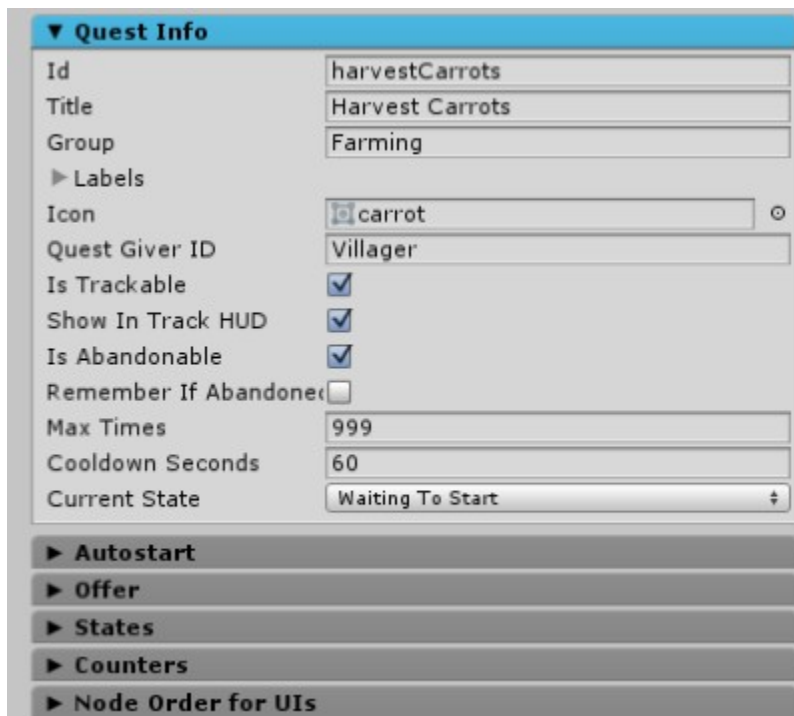
You can also create a new quest by clicking on the Quest Editor's **New quest asset...** button (if no quest is currently being edited in the window) or through the **Quest Giver** inspector.

Editing Quest Data

To edit a quest's main data, click on blank canvas (that is, not on a node). The inspector will show several collapsible sections of main data. To expand or collapse a section, click on its header.

Quest Info

The Quest Info section contains global information about the quest:



The screenshot shows the 'Quest Info' inspector panel with the following fields and values:

- Id:** harvestCarrots
- Title:** Harvest Carrots
- Group:** Farming
- Labels:** (collapsible section)
- Icon:** carrot (with a small icon preview)
- Quest Giver ID:** Villager
- Is Trackable:** ☒
- Show In Track HUD:** ☒
- Is Abandonable:** ☒
- Remember If Abandoned:** ☐
- Max Times:** 999
- Cooldown Seconds:** 60
- Current State:** Waiting To Start

Below the main fields are several collapsible sections:

- ▶ Autostart
- ▶ Offer
- ▶ States
- ▶ Counters
- ▶ Node Order for UIs

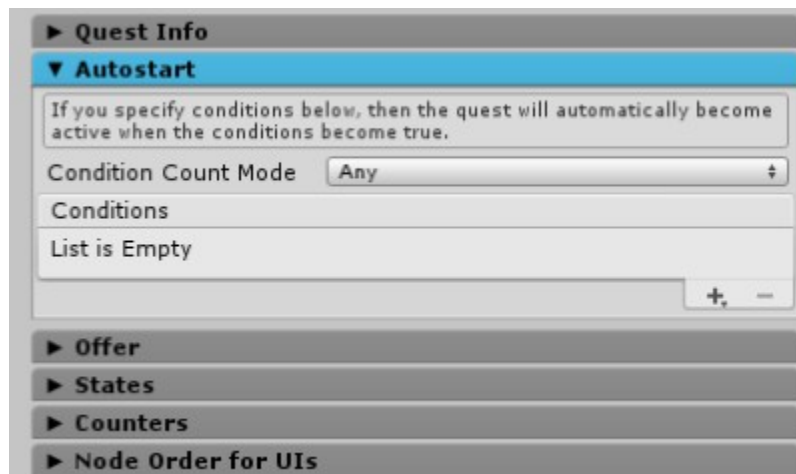
| Field | Description |
|---------------|--|
| ID | Assign a unique identifier that Quest Machine will use internally. |
| Title | The title displayed to the player in UIs. |
| Group | An optional heading under which to group the quest in the journal and HUD. |
| Labels | Optional labels which you can use to filter or sort quests. |
| Icon | An optional icon to show in UIs. |

| | |
|------------------------------|--|
| Quest Giver ID | Typically assigned automatically at runtime, the ID of the quest giver. |
| Is Trackable | Can the player turn HUD tracking on and off? |
| Show In Track HUD | Should the HUD show this quest? |
| Is Abandonable | Can the player abandon this quest? |
| Remember if Abandoned | Should the quest remain in the player's journal when abandoned? |
| Max Times | Number of times the quest giver can offer this quest. |
| Cooldown Seconds | Seconds that must pass before the quest giver can offer the quest again. |
| Current State | The current state of the quest. |

In Quest Machine, most text elements have three fields: a regular string, String Asset, and Text Table. The latter two allow you to assign text from assets instead of adding text directly into the quest; they're covered in more detail in the *Localization* section later in this chapter.

Autostart

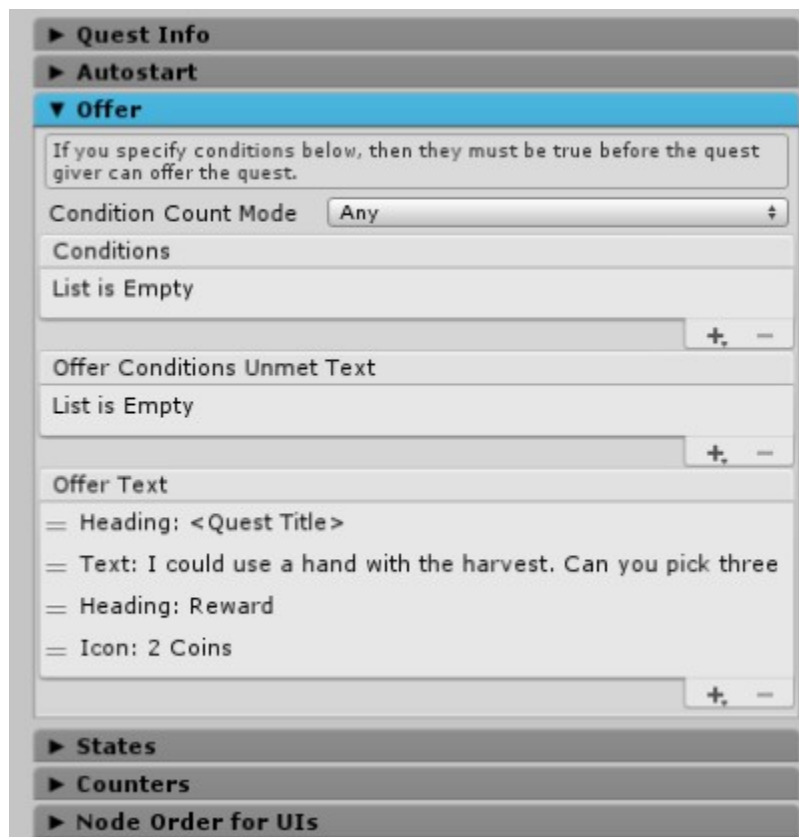
The Autostart section defines conditions that automatically start the quest. For example, if the player enters a specific trigger, it could be configured to send a message using the Message System. If the quest receives this message, it will automatically start the quest.



Conditions are covered in the [Conditions](#) section later in this chapter.

Offer

The Offer section defines the conditions that must be true before the quest giver can offer this quest. It also contains the text to display if the conditions aren't met or if the giver is offering the quest.

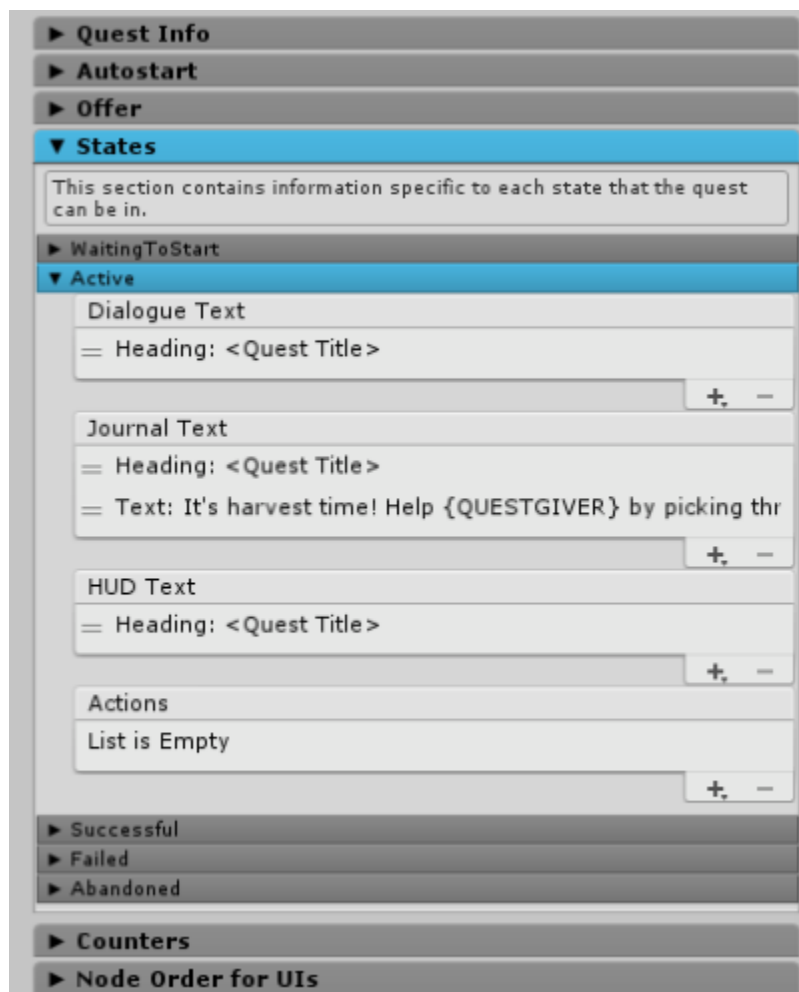


The screenshot shows the 'Offer' section of the Quest Machine editor. It includes a 'Condition Count Mode' dropdown set to 'Any', a 'Conditions' list (currently empty), an 'Offer Conditions Unmet Text' list (currently empty), and an 'Offer Text' list. The 'Offer Text' list contains four items: 'Heading: <Quest Title>', 'Text: I could use a hand with the harvest. Can you pick three', 'Heading: Reward', and 'Icon: 2 Coins'. Below the 'Offer' section are tabs for 'States', 'Counters', and 'Node Order for UIs'.

Conditions and UI Content are covered in the [Conditions](#) and [UI Content](#) sections later in this chapter.

States

The States section lets you define the text and actions for each state that the quest can be in.



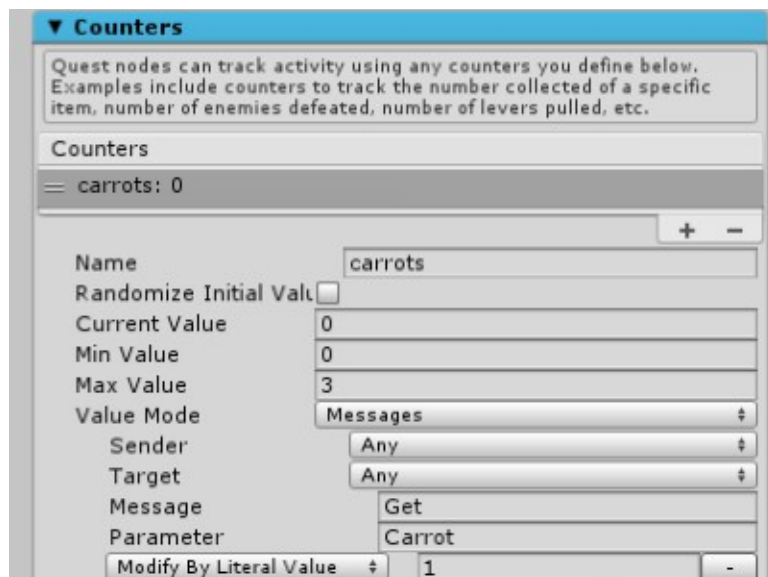
A quest can be in these states:

| State | Description |
|-----------------------|-------------------------------------|
| WaitingToStart | The quest hasn't started yet. |
| Active | The quest is active for the player. |
| Successful | A success quest node was reached. |
| Failed | A failure quest node was reached. |
| Abandoned | Player abandoned the quest. |

UI Content and Actions are covered in the [UI Content](#) and [Actions](#) sections later in this chapter.

Counters

The Counters section lets you define counters, integer variables that your quest can use to track activity.



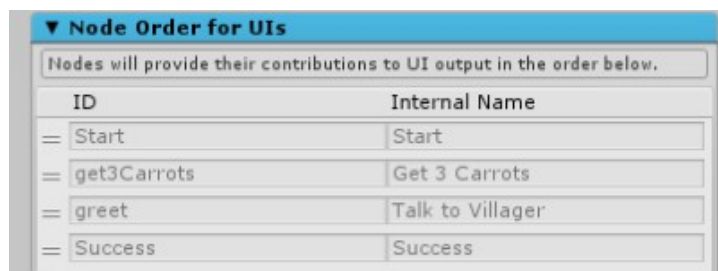
Counter values can change manually, upon receipt of messages from the Message System, or using a Data Synchronizer component. The Value Mode dropdown specifies how the value changes.

| | |
|--------------------------|---|
| Manually | Change via script or Set Counter Value action. |
| Message System | Configure the counter to listen for messages from the Message System. |
| Data Synchronizer | Configure a script to handle data synchronization messages, covered in the Scripting chapter. |

Note: Counters are usually paired with a condition node that waits for the counter to reach a required value. You can add the counter and condition node manually, as covered in the [Quick Start](#) tutorial, or use the [Counter Requirement Wizard](#) to automate the steps.

Node Order for UIs

The Node Order for UIs section lets you specify the order in which quest nodes contribute to UI content.



Editing Quest Node Data

Every quest starts with a **Start** node. In general, all quests should end in a **Success** or **Failure** node.

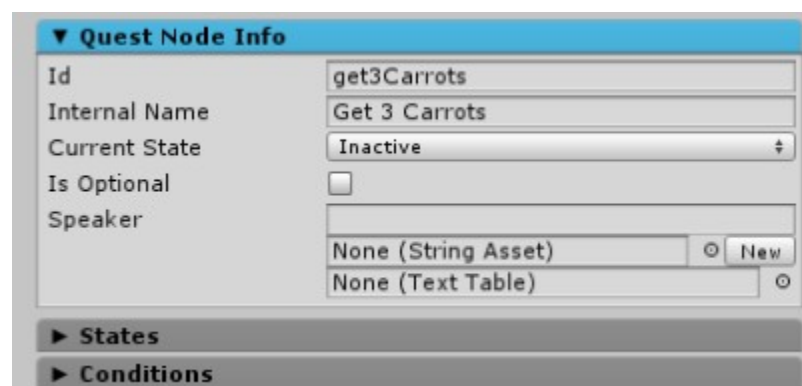
To add a quest node, right-click on a node and select **New Node** → *node type*.

| | |
|--------------------|--|
| Passthrough | When active, it immediately become true. |
| Condition | When active, it waits until specified conditions are true. |
| Success | When active, it immediately becomes true and sets the quest state to Successful. |
| Failure | When active, it immediately becomes true and sets the quest state to Failure. |

To inspect a quest node, click on it. The quest node inspector has several sections.

Quest Node Info

The Quest Node Info section lets you specify high level information about the node.



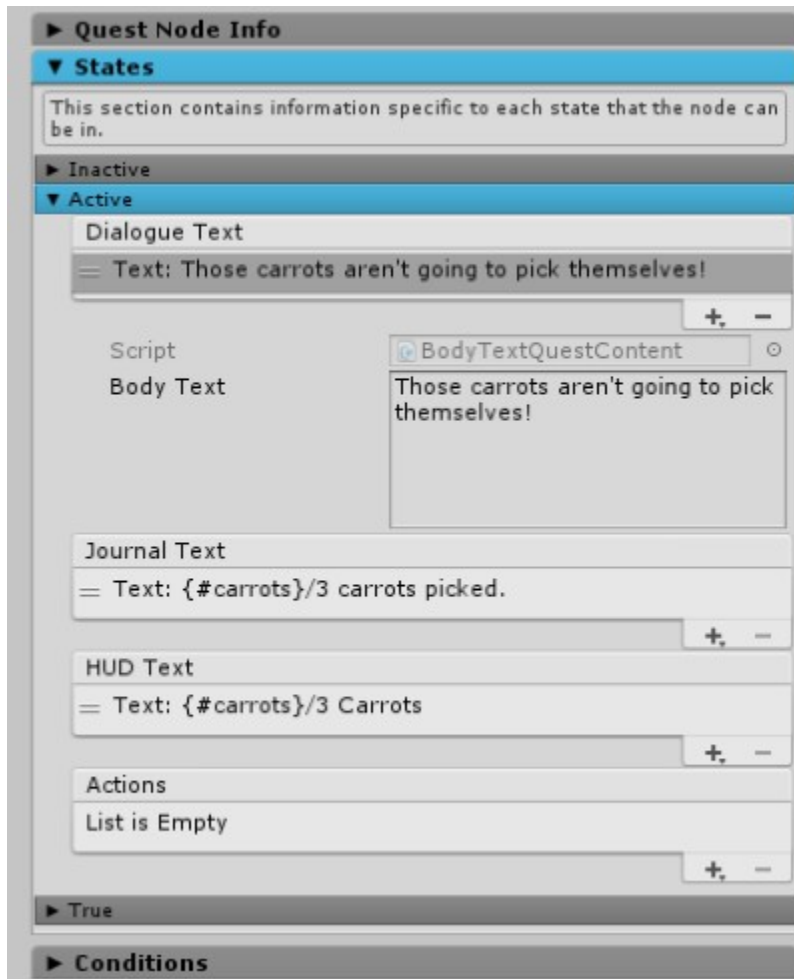
The screenshot shows the 'Quest Node Info' panel in a software editor. It contains the following fields and controls:

- Id:** A text field containing 'get3Carrots'.
- Internal Name:** A text field containing 'Get 3 Carrots'.
- Current State:** A dropdown menu showing 'Inactive'.
- Is Optional:** An unchecked checkbox.
- Speaker:** A section with two dropdown menus. The first shows 'None (String Asset)' with a 'New' button. The second shows 'None (Text Table)'.
- States:** A collapsed section indicated by a right-pointing arrow.
- Conditions:** A collapsed section indicated by a right-pointing arrow.

| | |
|----------------------|---|
| ID | An optional unique identifier to reference this quest node in scripts. |
| Internal Name | Shown only in the quest editor for your reference. |
| Current State | The current state of this node. |
| Is Optional | If a child node's conditions require that all parent nodes are true, don't require that this parent node is true if ticked. |
| Speaker | Leave blank if the quest giver is the speaker. Otherwise this is ID of an NPC. |

States

The States section lets you define the UI Content and Actions for each state that this node can be in.



A quest node can be in the **Inactive**, **Active**, or **True** state.

Dialogue Text is shown in the dialogue UI when speaking with the quest giver or the speaker assigned in the node's Speaker field.

Journal Text is shown in the quest journal UI.

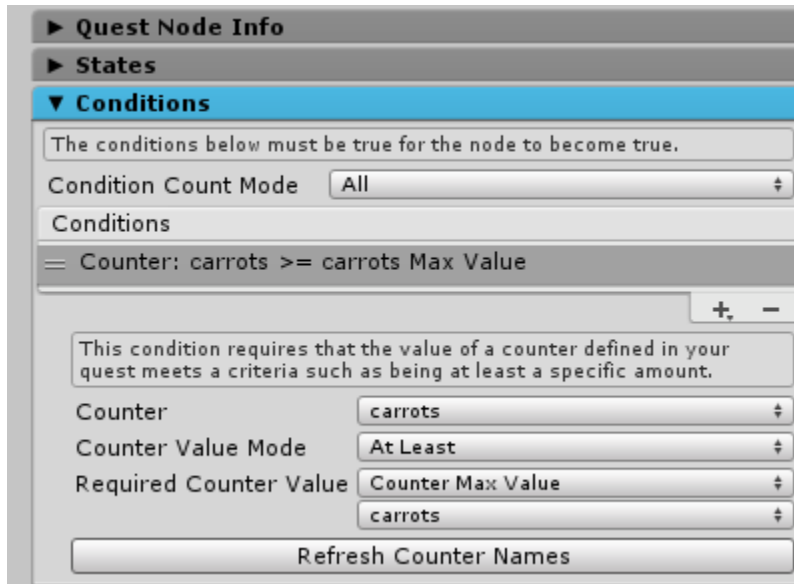
HUD Text is shown in the HUD.

Actions are activities that happen when the node enters the state.

UI Content and Actions are covered in the [UI Content](#) and [Actions](#) sections later in this chapter.

Conditions

For Condition nodes, the Conditions section lets you specify the conditions that must be true for the node to change from the Active state to the True state.



The screenshot shows the 'Conditions' panel in the Quest Machine editor. It has a title bar 'Quest Node Info' and a sub-header 'States'. The 'Conditions' section is expanded, showing a list of conditions. The first condition is 'Counter: carrots >= carrots Max Value'. Below the list, there is a description: 'This condition requires that the value of a counter defined in your quest meets a criteria such as being at least a specific amount.' The configuration fields are: 'Counter' (carrots), 'Counter Value Mode' (At Least), 'Required Counter Value' (Counter Max Value), and 'carrots'. A 'Refresh Counter Names' button is at the bottom.

Condition Count Mode specifies how many conditions in the Conditions list must be true.

Conditions are covered in the [Conditions](#) section later in this chapter.

UI Content Types

Quest Machine includes the following built-in UI content types. You can also define your own content types (see the [Scripting](#) chapter).

Audio Clip

Plays an audio clip. Example: Play a trumpet when the player completes a quest.

Body Text

Displays regular body text. Most of your quest content will probably be Body Text. In all text, you can use special [Tags](#). You can use the [Reference Window](#) for access to commonly-used tags.

Button

Displays a button that performs [actions](#) when clicked. Example: Configure reward buttons to allow the player to choose a reward. For rewards, you will typically set a group number for the reward buttons. When the player clicks one button to choose the reward, all buttons with the same group number will be made non-interactable. See the Demo's Pesky Rabbits quest for an example.

Heading Text

Displays heading text.

Icon

Displays an icon with count and caption. Example: Show the targets of a quest, or the rewards offered upon completion.

Conditions

Quest Machine includes the following built-in condition types. You can also define your own condition types (see the [Scripting](#) chapter).

Counter

A counter value must meet at a requirement such as being at least a specific amount. Example: Configure a counter to increment when the player kills an Orc. Then configure a counter condition to require that the counter is at least 5.

Message

A specific message must be received from the Message System. You can use the [Reference Window](#) for access to commonly-used messages.

Parents

Some number of parent nodes must be in the true state.

Quest State

A quest must be in a specific state. To check this quest, leave the Required Quest ID field blank. Example: Use to require that the player complete another quest first.

Quest Node State

A quest node must be in a specific state. To check this quest node, leave the Required Quest Node ID field blank.

Timer

When the node becomes active, this condition starts a countdown using a counter defined in the quest. Every second, the counter value decreases by one. If it reaches zero, the condition becomes true. Example: Use for timed quests.



Actions

Quest Machine includes the following built-in action types. You can also define your own action types (see the [Scripting](#) chapter).

Activate GameObject

Activates a GameObject in the scene.

Alert

Displays UI content in the alert UI.

Animator

Controls an animator to crossfade to an animation state or change a parameter value.

Audio

Plays an audio clip.

Control Spawner

Starts, stops, or despawns a Spawner.

Give Quest To Quester

Gives a quest to a quester (i.e., a GameObject with a Quest Journal component, such as the player).

Instantiate Prefab

Instantiates a prefab.

Message

Sends a message to the Message System. You can use the [Reference Window](#) for access to commonly-used messages.

Scene Event

Executes a UnityEvent in a scene. Allows you to hook up behavior in the inspector.

Set Counter Value

Modifies a counter value.

Set Indicator

Sets an NPC's quest indicator state for this quest.

Set Quest State

Sets a quest's state. Leave the Quest ID field blank to set this quest's state.

Set Quest Node State

Sets a quest node's state. Leave the Quest Node ID field blank to set this quest node's state.

UnityEvent

Invokes a UnityEvent on a non-scene object such as a ScriptableObject asset.

Tags

Tags are special markup codes in quest text. At runtime, UIs replace tags with their current values. These tags are built in:

| | |
|--------------------|--|
| {QUESTGIVER} | Quest giver's display name. |
| {QUESTGIVERID} | Quest giver's ID. |
| {QUESTER} | Quester's (player's) display name. |
| {QUESTERID} | Quester's (player's) ID. |
| {#counter} | Current value of a counter. |
| {<#counter} | Counter's minimum value. |
| {>#counter} | Counter's maximum value. |
| {:counter} | Counter's value formatted as a time (useful for Timer conditions). |
| {TARGETENTITY} | Name of quest target (typically used in message conditions). |
| {TARGETDESCRIPTOR} | Display name and count of procedurally-generated quest target. |
| {DOMAIN} | Location of procedurally-generated quest target. |

You can enter these tags manually or use the [Reference Window](#) to copy and paste them, which is especially useful for counter tags.

The {#counter} tags reference a counter in the current quest. If you want to reference a counter in a different quest, use the format {#questID:counter}.

All other tags are treated as lookup values in the speaker's [Text Table](#). If the speaker doesn't have a text table, or if the text table doesn't have a value for the tag, the UI shows the tag itself (without braces).

GameObject IDs

Quest Machine often identifies GameObjects by ID. The following components will associate an ID with a GameObject:

| Component | Typical Use |
|------------------|--|
| Quest Journal | Player |
| Quest Giver | NPCs that give quests |
| Quest Entity | Entities that can be involved in procedurally-generated quests |
| Quest Machine ID | Other GameObjects that need to be identified by ID |

Messages

Quest Machine uses a Message System for communication between its various parts. Messages consist of these parts:

| | |
|-----------|---|
| Message | A string such as "Greeted" |
| Parameter | A string such as "Villager" |
| Values | Any number of optional values to pass along with the message. |

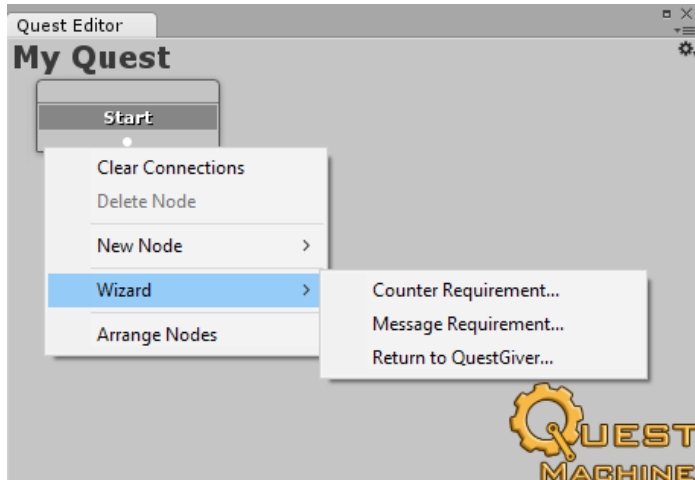
These are commonly-used messages:

| | |
|----------------------------|---|
| Greet | Sent <i>before</i> starting dialogue. Parameter is quest giver's ID. |
| Greeted | Sent <i>after</i> starting dialogue. Parameter is quest giver's ID. |
| Discuss Quest | Sent <i>before</i> starting discussion of a quest. Parameter is quest ID. Value is quest giver's ID. |
| Discussed Quest | Sent <i>after</i> starting discussion of a quest. Parameter is quest ID. Value is quest giver's ID. |
| Quest State Changed | Sent when a quest state changes. Parameter is quest ID. Value is quest node ID or blank. Value 2 is state. |
| Start Spawner | Sent to start a spawner. Parameter is spawner name. |
| Stop Spawner | Sent to stop a spawner. Parameter is spawner name. |
| Despawn Spawner | Sent to stop a spawner and despawn all spawned objects. Parameter is spawner name. |
| Quest Alert | Shows a message in the alert ID. Parameter is quest ID. Value is message. |
| Refresh UIs | Tells all open Quest Machine UIs to refresh their content. |
| Refresh Indicator | Tells an entity to refresh its overhead indicators. Parameter is entity ID. |
| Set Indicator State | Tells an entity to set an indicator. Parameter is quest ID. Value is state. |
| Quest Track Toggle Changed | Toggles quest tracking. Parameter is quest ID. Value is true/false. |
| Quest Abandoned | Notifies that a quest was abandoned. Parameter is quest ID. |
| Quest Counter Changed | Notifies that a quest counter changed. Parameter is quest ID. First value is counter name. Second value is counter value. |
| Set Quest Counter | Sets a quest counter. Parameter is quest ID. First value is counter name. Second value is counter value. |
| Increment Quest Counter | Increments a quest counter. Parameter is quest ID. First value is counter name. Second value is amount to increment. |

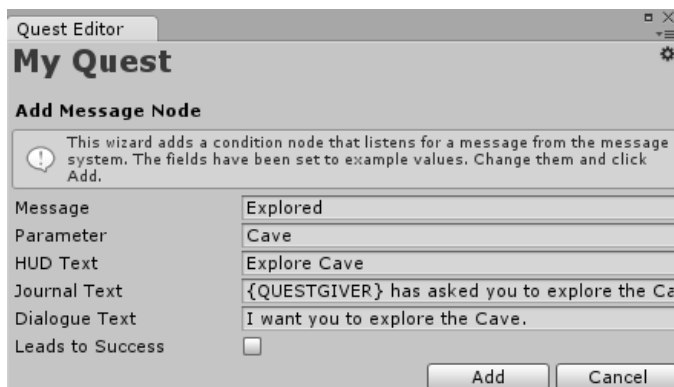
You can enter these messages manually or use the Reference Window to copy and paste them without having to type them in manually. In C# code, you can call the corresponding methods in the QuestMachineMessages class to send the messages.

Quest Editor Wizards

The Quest Editor provides some wizards to automate common steps. To open a wizard, right-click on the node you'd like to link from and select the wizard, or click the gear menu in the upper right.

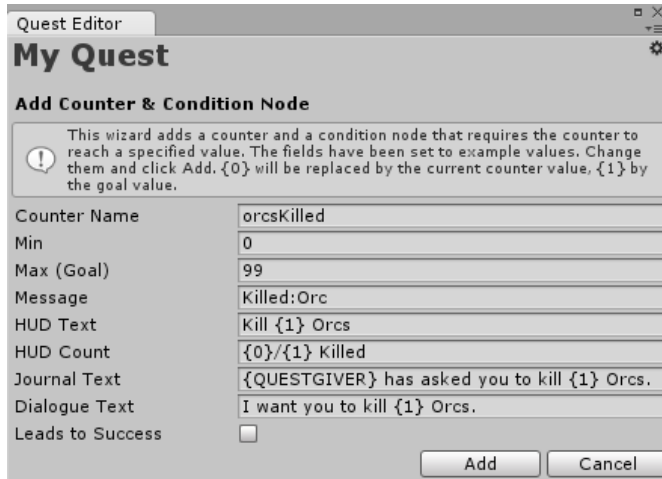


Message Requirement Wizard



The Message Requirement Wizard adds a Condition node that listens for a message from Quest Machine's message system.

Counter Requirement Wizard



Quest Editor

My Quest

Add Counter & Condition Node

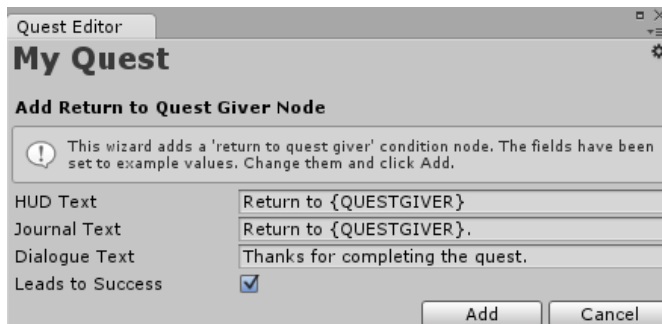
This wizard adds a counter and a condition node that requires the counter to reach a specified value. The fields have been set to example values. Change them and click Add. {0} will be replaced by the current counter value, {1} by the goal value.

| | |
|------------------|--|
| Counter Name | orcsKilled |
| Min | 0 |
| Max (Goal) | 99 |
| Message | Killed:Orc |
| HUD Text | Kill {1} Orcs |
| HUD Count | {0}/{1} Killed |
| Journal Text | {QUESTGIVER} has asked you to kill {1} Orcs. |
| Dialogue Text | I want you to kill {1} Orcs. |
| Leads to Success | <input type="checkbox"/> |

Add Cancel

The Counter Requirement Wizard adds a counter that increments when it receives a message, and a Condition node that waits for the counter to reach a goal value.

Return To Quest Giver Wizard



Quest Editor

My Quest

Add Return to Quest Giver Node

This wizard adds a 'return to quest giver' condition node. The fields have been set to example values. Change them and click Add.

| | |
|------------------|-------------------------------------|
| HUD Text | Return to {QUESTGIVER} |
| Journal Text | Return to {QUESTGIVER}. |
| Dialogue Text | Thanks for completing the quest. |
| Leads to Success | <input checked="" type="checkbox"/> |

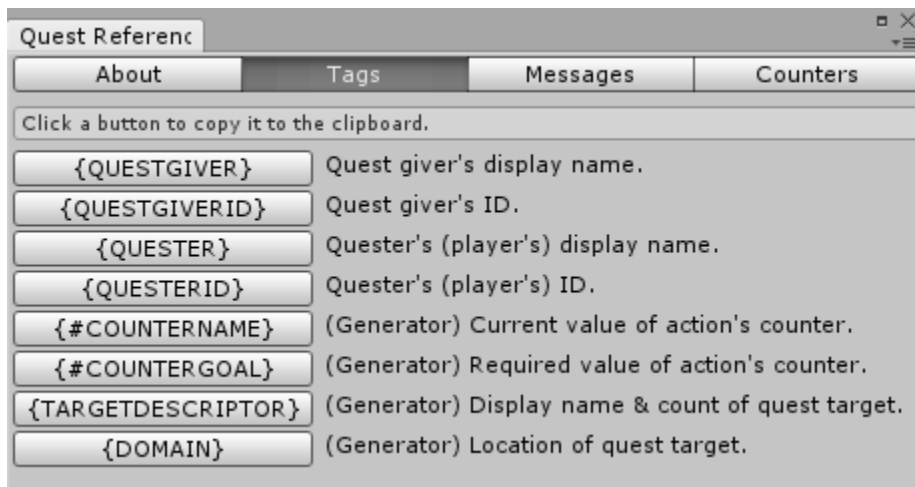
Add Cancel

The Return To Quest Giver Wizard adds a condition node that waits until the player returns to speak with the quest giver. This wizard will ask if you want to listen for a “Discuss Quest” message, which sets the condition node true just *before* showing the dialogue UI, or a “Discussed Quest” message, which sets the condition node true after showing the dialogue UI.

Reference Window

The Reference Window is a utility that lets you copy tags and messages to the clipboard to make it easier to add them to your quest content.

To open the Reference Window, select menu item **Tools** → **Pixel Crushers** → **Quest Machine** → **Quest Reference**.



Click on a button to copy it to the clipboard. Then position the cursor in the desired field in the quest inspector or Text Table editor and paste in the value.

Text Tables

You can assign text tables to quest givers to allow them to look up their text from a table instead of entering it directly into a quest.

To create a text table, right-click in the Project view and select **Create** → **Pixel Crushers** → **Common** → **Text** → **Text Table**.

Text tables serve two purposes: Localization and Dialects.

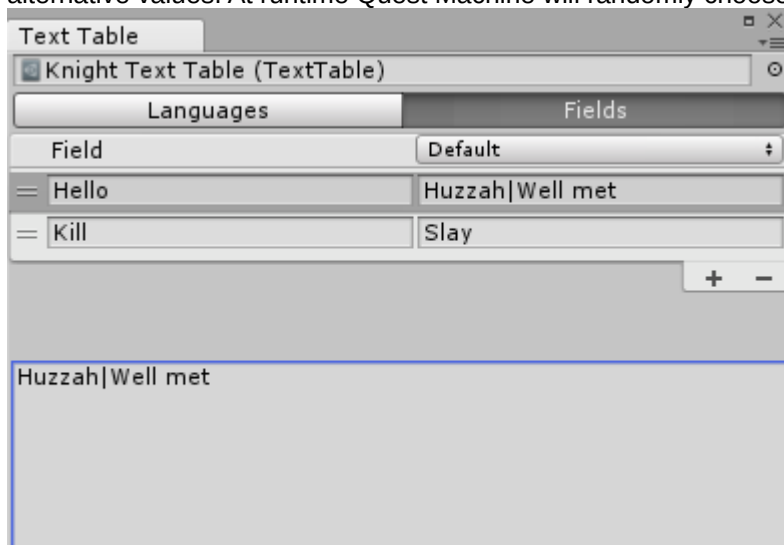
Localization

To add a language, use the Languages tab. For more information on localization, see [Appendix 1: Text Tables and Localization](#).

Dialects

Dialects make quest text more interesting by using different synonyms for different NPCs. For example, a pirate may greet the player with “Ahoy!” while a knight might greet her with “Huzzah!”

To define a field, use the Fields tab. Select the language from the dropdown, and then use the Field list to edit field names and their values. You can use the pipe character (|) in your field value to specify alternative values. At runtime Quest Machine will randomly choose one alternative.



Dialects are especially powerful with procedurally-generated quests because the same quest (e.g., kill 5 orcs) generated by different NPCs could end up with very different-sounding text, such as:

“Ahoy, matey! I want ye to keelhaul 5 scurvy Orcs.”

or:

“Greetings, fine hero! I beseech thee to slay 5 vile Orcs.”

Chapter 4: Quest Scene Setup & Management

This chapter describes how to set up your scene to work with Quest Machine and how to manage quests at runtime.

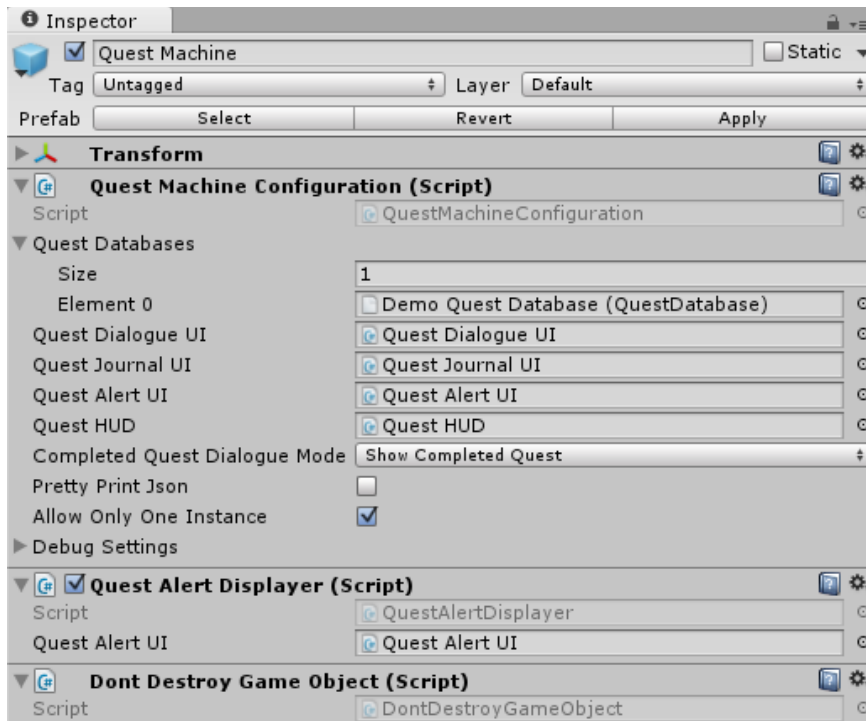
How to Set Up the Scene

To set up a new scene:

1. Add the **Quest Machine** prefab, which is located in **Plugins ► Pixel Crushers ► Quest Machine ► Prefabs**. This contains a configuration component, quest alert displayer component, and Quest Machine's default UI, which you can customize. Assign a quest database. The Quest Machine prefab is covered in more detail in the next section, *Quest Machine Configuration*.
2. (Optional) Add the **Input Device Manager** prefab, located in the same folder. This gracefully handles input changes between keyboard/mouse, joystick, and touch.
3. (Optional) If you want to use Quest Machine's save system, add a Save System component, typically to an empty GameObject named **Save System**, but it's fine to add it to the Quest Machine GameObject instead.
4. Add a **Quest Journal** component to the player. Optionally add a **Position Saver** to save the player's position.
5. Add a **Quest Giver** component to NPCs. Assign quests.
6. (Optional) To procedurally generate quests, add a **Quest Generator Entity** and optional Reward Systems to the NPC. Add **Quest Entity** components to other GameObjects. (See [Chapter 6: Quest Generation](#) for more details.)

The Quest Machine, Input Device Manager, and Save System prefabs act as persistent singletons, meaning they will survive scene changes, and they will automatically destroy duplicates in new scenes. This allows you to add an instances of these prefabs to every scene to make it easy to playtest individual scenes in the Unity editor.

Quest Machine Configuration

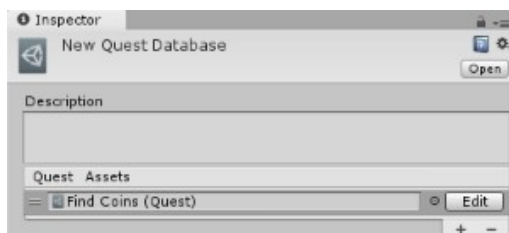


The Quest Machine prefab has a **Quest Machine Configuration** component that points to child GameObjects containing the UIs. It also has a **Quest Alert Displayer** that listens for alert messages and sends them to the alert UI, and a **Dont Destroy Game Object** component that allows it to survive scene changes. The Quest Machine Configuration also has a list of Quest Databases, which are described in the next section.

The **Debug Settings** turn on logging, including Message System logging. To log Message System activity only for a specific GameObject, add a **Message System Logger** component to that GameObject.

Quest Database

Quest databases give Quest Machine access to your quests even if the quest giver is no longer present in the scene. To create a quest database, right-click in the Project view and select **Create** → **Pixel Crushers** → **Quest Machine** → **Quest Database**. Then add quest assets to it and assign it to the Quest Machine Configuration.



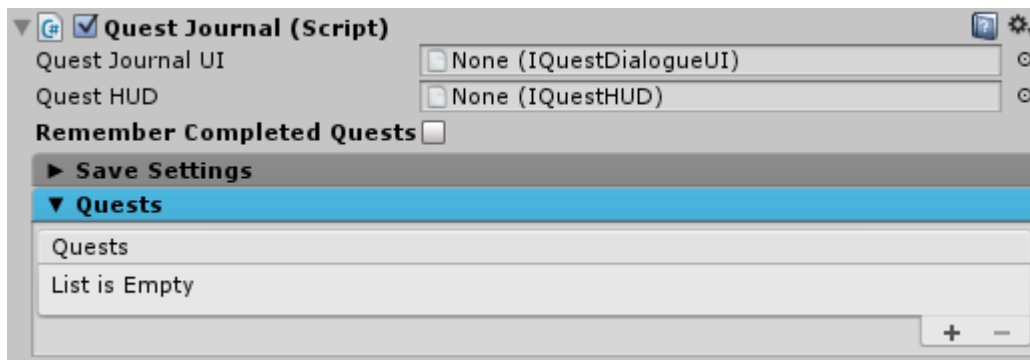
Input Device Manager

If you're using the built-in UI system, you can add the **Input Device Manager** prefab to your scene. The Input Device Manager component on this prefab gracefully handles transitions between mouse, joystick, and keyboard control, and helps the UIs know when to auto-focus UI buttons (in joystick and keyboard mode) and when to leave buttons unfocused (in mouse mode).

You may also need to add a standard Unity EventSystem if your scene doesn't already have one.

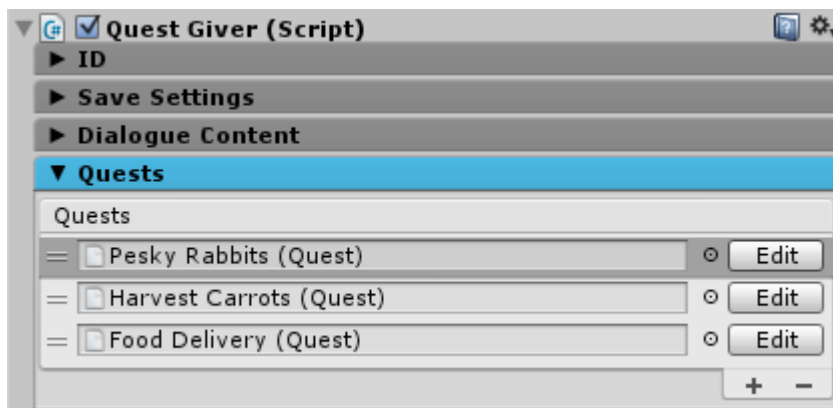
Quest Journal

Add a **Quest Journal** component to your player. This component keeps track of the player's quests.



Quest Giver

To allow an NPC to give quests, add a **Quest Giver** component as described in the Quick Start section.

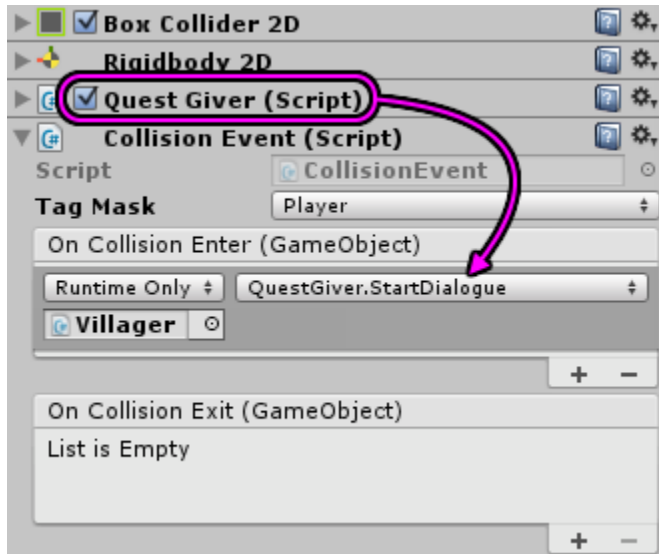


To make a quest giver give a quest, you can start a dialogue (described below) or use the methods `QuestGiver.GiveQuestToQuester()` or `QuestGiver.GiveAllQuestsToGiver()`.

How to Start Dialogue

To start dialogue with an NPC, first make sure the NPC has a **QuestGiver** component. Then call **QuestGiver.StartDialogue**, passing the player GameObject, or **QuestGiver.StartDialogueWithPlayer** to use the first GameObject in the scene that's tagged as "Player".

You can do this in a script, but it's also possible to configure it without any scripting. For example, to configure the NPC to start dialogue as soon as the player bumps into it, add a **CollisionEvent** component (which is described in greater detail later in this chapter). In the On Collision Enter (GameObject) event, click "+". Then assign the QuestGiver component and select StartDialogue.



If the NPC has a trigger collider and you want to start dialogue when the player enters the trigger collider, add a **TriggerEvent** component instead of a CollisionEvent.

In either case, make sure your player's GameObject is tagged as Player, or change the Tag Mask on the CollisionEvent / TriggerEvent component.

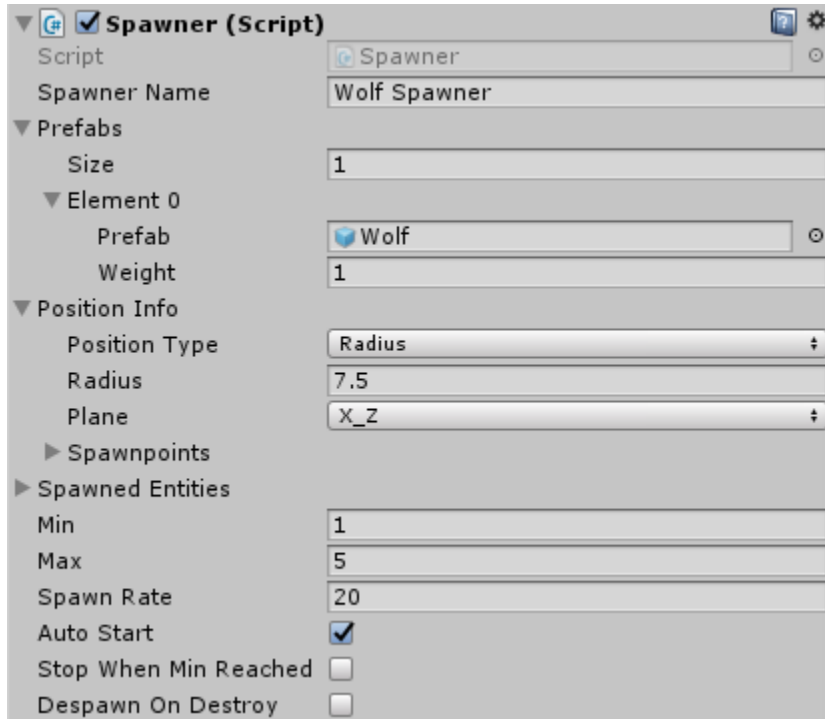
Quest Indicators



If you want your NPC to show a quest indicator, add a Quest Indicator Manager component as described in the *Quest Indicators* section in [Chapter 5: Quest UIs](#).

Spawner

Spawners add new instances of one or more prefabs to the scene. In the demo scene, spawners replenish the scene with new carrots and creatures as the player removes them.



In the **Prefabs** section, specify the prefabs that the spawner can spawn and the relative weight (probability) that it will spawn.

In the **Position Info** section, specify whether the spawner will spawn instances in a radius around itself or only at specific spawnpoints. If you select *Spawnpoints*, create empty child GameObjects as spawnpoints, position them, and assign them to the **Spawnpoints** list. If you want to assign starting entities instead of empty GameObjects, add a **Spawned Entity** component to them to allow the spawner to keep track of them.

You can control spawners manually or by using the **Control Spawner** action in quests.

Quest Events

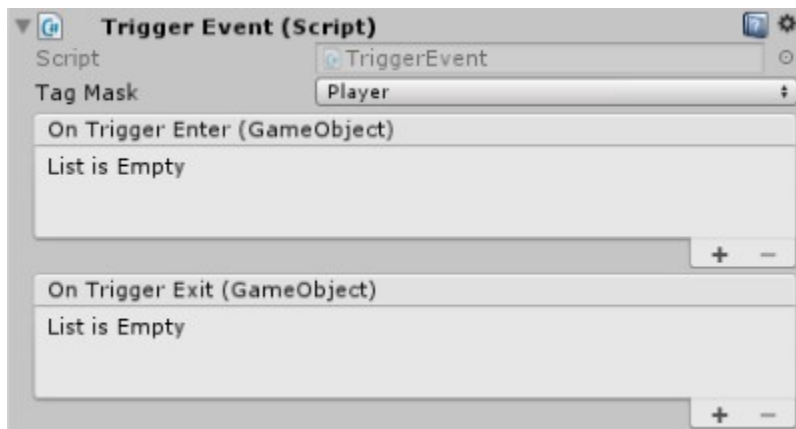
The following components will assist you in controlling quest activity.

UnityEvent Components

UnityEvents are a convenient way to hook up quest behavior in the inspector. Quest Machine includes the following general purpose event-detecting components that can invoke UnityEvents. You can use them in conjunction with a **Quest Control** component to control quest activity.

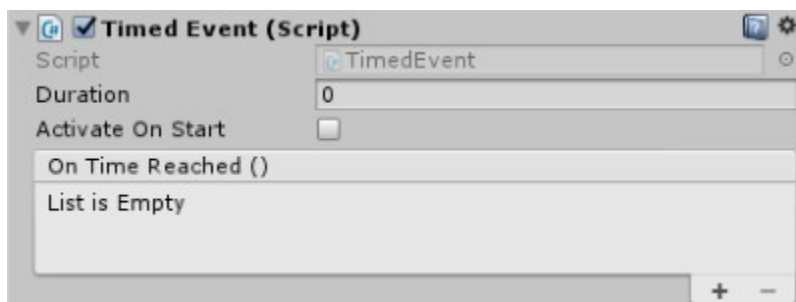
Collision Event and Trigger Event

Invokes UnityEvents when a collider with the specified tag(s) enters or exits. A Trigger Event component is a common way to make something happen with a quest when the player enters a trigger area.



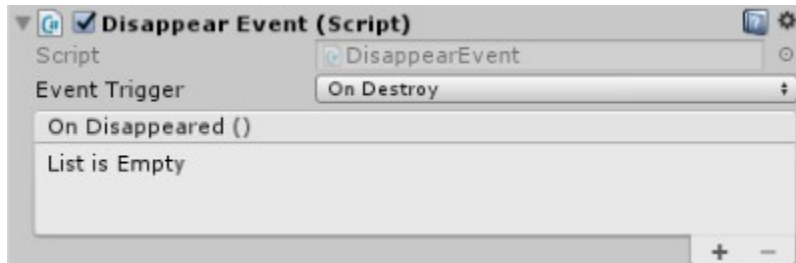
Timed Event

The **Timed Event** component invokes a UnityEvent when a specified duration has passed. You can also configure it to invoke the UnityEvent when the component starts.



Disappear Event

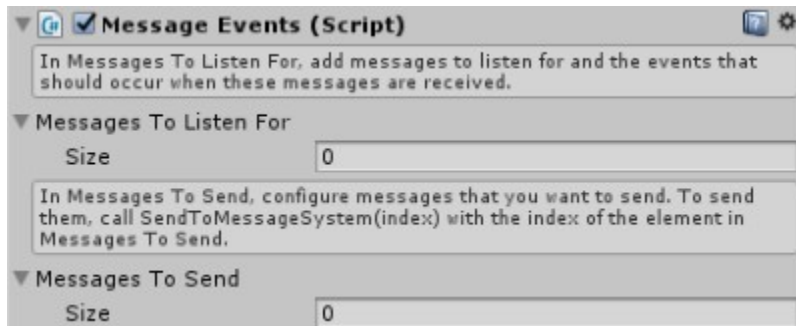
The **Disappear Event** component can be configured to invoke a UnityEvent when disabled or destroyed. This is a convenient way to trigger activity when the player destroys a target or picks up an item from the scene.



If you don't want to invoke this UnityEvent when the scene is unloaded – for example when changing scenes – call `SceneNotifier.NotifyWillUnloadScene()`. The [Save System](#) does this automatically; if you're using the Save System, you don't need to worry about it.

Message Events

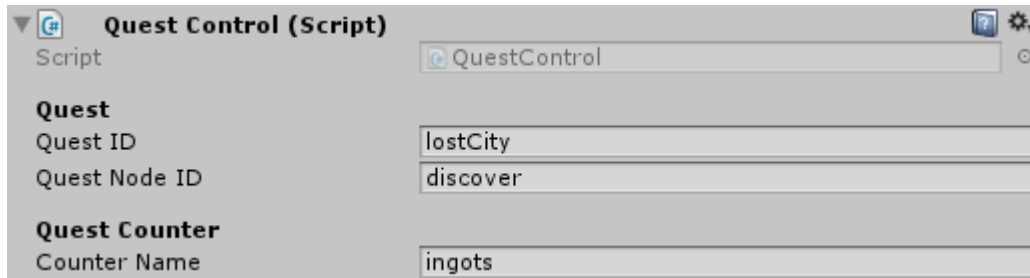
The **Message Events** component works with Quest Machine's Message System. You can use it to invoke UnityEvents when receiving specific messages, or to send messages.



Programmers may find it more convenient to implement the [IMessageHandler](#) C# interface in their own scripts instead.

Quest Control

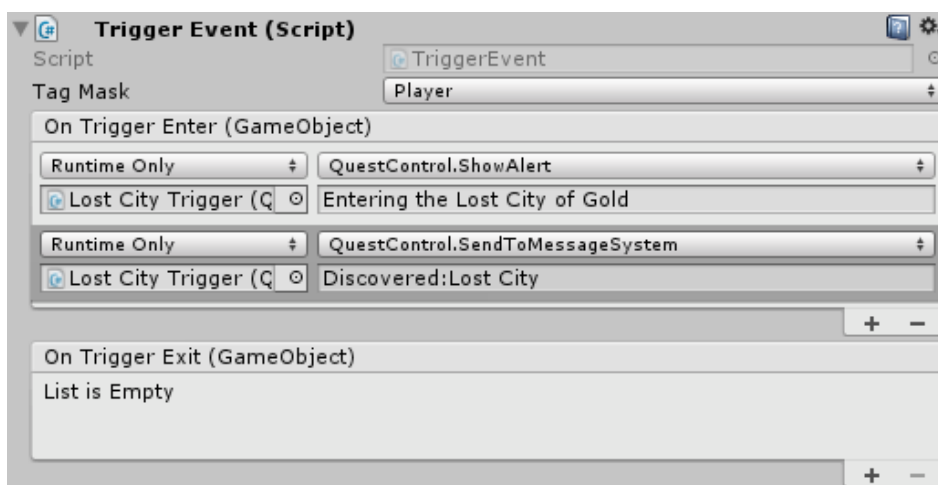
The **Quest Control** component gives you methods that you can hook up UnityEvents such as a UI button's `OnClick()` event, or a Trigger Event (when a `GameObject` enters a trigger collider), or a Disappear Event (when a `GameObject` disappears). Some methods require a quest ID, quest node ID, or counter name. Set the fields on **Quest Control** to specify this information.



The available methods are:

| Method | Description |
|---|--|
| <code>SetQuestState</code> | Sets the quest (Quest ID) to a state. |
| <code>SetQuestNodeState</code> | Sets the node of (Quest ID)+(Quest Node ID) to a state. |
| <code>SetQuestCounter</code> | Sets the value of a quest counter (Quest ID)+(Counter Name). |
| <code>IncrementQuestCounter</code> | Increments the value of a quest counter (Quest ID)+(Counter Name). |
| <code>SendMessageToMessageSystem</code> | Sends a message to the Message System. To include a parameter, add it after a colon (:), such as "Get:Coin". To include a value, add it after another colon: "Get:Coin:5". |
| <code>ShowAlert</code> | Shows a string using the alert UI. |

For example, say you have an "explore" quest to discover the Lost City of Gold. The quest has a condition node that listens for the message "Discovered" with parameter "Lost City". You can add a trigger collider to the Lost City's entrance and hook up its `OnTriggerEnter()` event to `QuestControl.ShowAlert` and `QuestControl.SendMessageToSystem` with these parameters:



Chapter 5: Quest UIs

Quest Machine is GUI system independent. This means you can use any GUI system. Quest Machine ships with a robust implementation for Unity UI that also automatically detects and supports TextMesh Pro. It also comes with a preconfigured UI that you can use as-is or customize to look the way you want. (Note: The evaluation version does not ship with TextMesh Pro support.)

Quest Machine uses these UIs:

- **Dialogue:** Used when talking with NPCs to accept and complete quests.
- **Journal:** Shows the player's active and completed quests.
- **HUD:** Shows tracking information for active quests.
- **Alert:** Shows pop-up messages.
- **Indicators:** Typically shown over NPCs' heads to indicate quest-related activity.

The **Quest Machine** prefab includes a set of default UIs that you can use as-is, customize, or replace. These UIs use Quest Machine's Unity UI.

How the Unity UIs Work

The default UIs work with template UI elements. To display content at runtime, they instantiate copies of the templates and fill them in with the relevant details. You can customize the UI by changing the appearance of the UI elements or by assigning new UI elements to the UI component.

More details about each type of UI is below, followed by instructions on how to customize them.

Dialogue UI

These are the template elements in the default dialogue UI:



Journal UI

These are the template elements in the default journal UI:



The **Group** element is a button that acts as a foldout for quests that are categorized under a common group name.

Active quests that are trackable will show the toggle element to allow the player to toggle tracking on and off.

An inactive panel holds the template elements for the Abandon Quest dialog.

HUD

These are the template elements in the default HUD:



Alert UI

These are the template elements in the default alert UI:



Indicators

Typically shown over a quest giver's head, indicators tell the player that an NPC is available for quest-related dialogue. Indicators can be in different states such as Offer or Talk.



To show indicators, the NPC must have a **Quest Indicator Manager** component, which may be on a child GameObject. This component should also reference a **Quest Indicator UI**, which is a GameObject or prefab that contains a GameObject (typically a sprite or 3D model) for each indicator state.

The NPC's Quest Indicator Manager keeps track of the indicator states of all quests that the NPC is involved in. It tells the Quest Indicator UI to show the indicator that corresponds to the highest priority indicator state. For example, say Johann is ready to reward the player for completing the Harvest Carrots quest, and he can also offer the Pesky Rabbits quest. The higher priority indicator state is the Harvest Carrots quest's Talk state, so the Quest Indicator Manager will show the indicator for this indicator state.

How to Customize the Appearance of UIs

The easiest way to customize the Dialogue UI, Journal UI, HUD, or Alert UI is to start with the Quest Machine prefab. Break the prefab instance, replace the textures on the UI elements, and reposition the UI elements as you like.

To customize quest indicators, make a copy of the **Quest Indicator UI** prefab and replace the sprites with different sprites or GameObjects.



Custom UI Implementations

To implement custom UI code instead of using the built-in Unity UI implementations, simply implement the relevant C# interface: `IQuestDialogueUI`, `IQuestJournalUI`, `IQuestHUD`, or `IQuestAlertUI`. The interface scripts are fully documented.

If you create custom content types, you will need to write a custom UI implementation or at least make a subclass of the built-in Unity UI implementation to handle the new content type.

Chapter 6: Quest Generation

Hand-written quests have their limits. They can only express what the author has anticipated at design time. Hand writing quests is also very time-consuming and labor-intensive.

Procedural quest generation adds a significant extra level of depth to your game by creating new quests at runtime based on changes to the game world that occur during play.

This requires some initial setup at design time to define the elements of the game world that can be involved in procedurally generated quests. But, once configured, they can be used to automatically generate an unlimited number of quests both at design time and runtime.

This chapter describes how Quest Machine generates quests and how to configure your project for dynamic quest generation.

Overview of How Procedural Quest Generation Works

This is how Quest Machine generates a quest:

1. From the areas of the game world that the quest giver is aware of, select an entity that the quest giver deems urgent, such as a threatening enemy.
2. Devise a plan to relieve the urgency, such as defeating the enemy. (Internally, Quest Machine uses STRIPS-style goal oriented action planning, or GOAP.)
3. Select rewards appropriate for the difficulty of the plan.

This process relies on the following types of elements:

- **Domain:** An area of the game world.
- **Entity:** A creature or object in the game world. Entities have:
 - **Actions:** Things that can be done to or with the entity.
 - **Drives:** The entity's personality traits (e.g., violent or peaceful?)
- **Urgency Function:** Rules that determine the urgency of an entity's presence.
- **Action:** Things that the player can do to or with entities. Actions have:
 - **Motives:** Personality-based rationale for choosing this action. Includes a text explanation presented to the player and a set of drives (values) that the generator uses to choose actions.
 - **Requirements:** States that the game world must be in before doing the action.
 - **Effects:** How the game world will change if the player does the action.
- **Reward System:** Methods of rewarding the player for completing quests.

Since the quest giver must project its world model into the future to make plans, it works with abstract elements such as *entity types* and *domain types*, rather than entities and domains, which are the actual instances in the game world. These abstract elements are stored as assets in your project.

To manage these assets, you'll use the Quest Generator window, accessed through menu item **Tools** → **Quest Machine** → **Quest Generator**.

The following sections describe quest generation elements and editor windows. A step-by-step setup tutorial is at the end of the chapter.

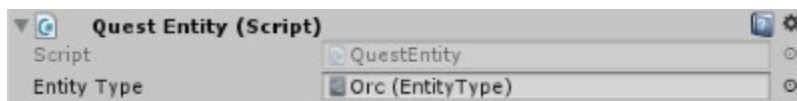
Quest Generator Elements

The Quest Generator uses three types of objects in the game world:

- **Quest Entity:** A GameObject that a quest generator can include in a quest.
- **Quest Generator Entity:** A GameObject that can generate quests and offer them to the player.
- **Quest Domain:** An area of the game world that a quest generator watches for the existence of quest entities,

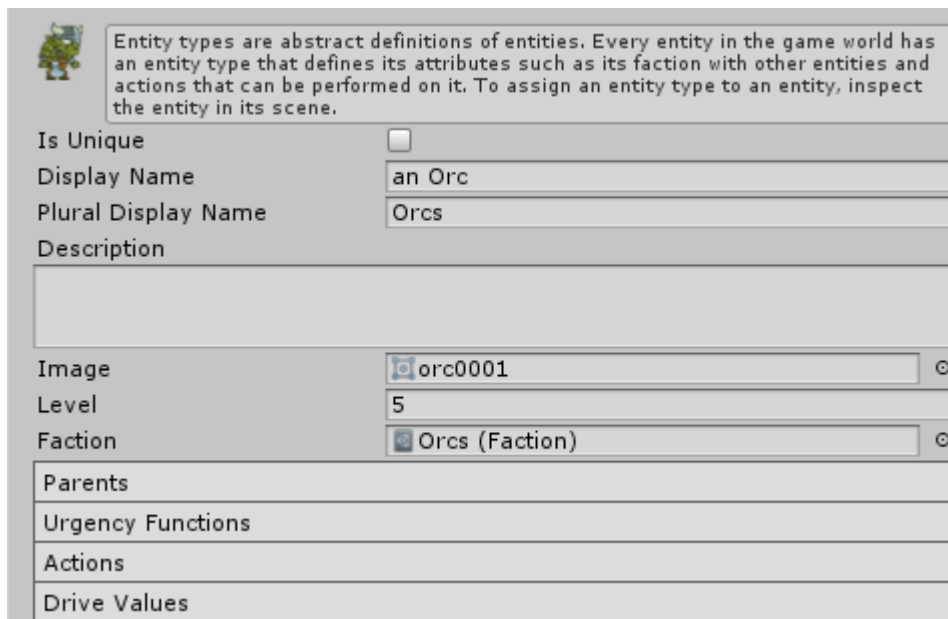
The following sections describe these elements in more detail.

Quest Entity



A quest entity is a GameObject in a quest domain that a quest giver can involve in a quest task. For example, if a knight generates a quest to rid the forest of Orcs, those Orcs are quest entities.

Entity Type

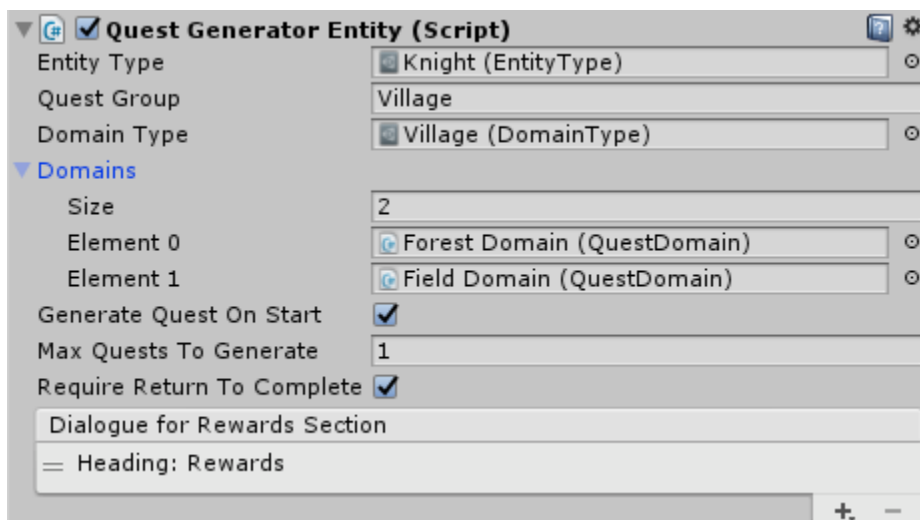


Quest entities, as GameObjects in the scene, represent the actual state of the game world. The quest generator, however, must also anticipate the effects of quest tasks on the game world. To do this, it maintains internal models that represent what it expects the world state to be as the player completes quest tasks. For example, if a quest task requires the player to play a pipe to summon rats, the anticipated world state will contain rats, even though the actual current world state may not contain rats. This means the quest generator can't operate solely on quest entities in the scene. Instead, it works on a more abstract level with *entity types*.

Entity types contain a number of attributes to help the quest generator plan quests:

- **Urgency Functions:** Define how important the quest entity is to a quest giver. For example, a sick orphan might have very high urgency to a compassionate priest but low urgency to a self-centered assassin.
- **Actions:** Actions that can be performed on the quest entity, such as “Attack” or “Fetch,” and the results they produce.
- **Drive Values:** The “personality” of the entity, used primarily by quest generator entities to select actions that fit its preferences.
- **Factions:** Relationships (like/dislike) to other quest entities.
- **Parents:** A list of parent entity types from which the entity type inherits attributes such as actions and urgency functions.

Quest Generator Entity



A Quest Generator Entity is any GameObject that can generate quests. Examples of quest generators:

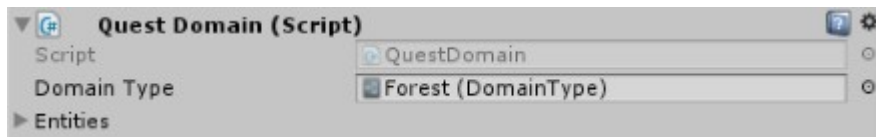
| Quest Giver/Generator | Example Quest |
|----------------------------------|--|
| A farmer | Clear his field of rabbits. |
| A baker | Fetch five apples so she can bake a pie. |
| A space station mission terminal | Transport cargo to a mining asteroid. |

To generate a new quest, the quest giver examines the quest entities in its domains, identifies a high priority entity, develops a plan to handle the entity, and selects rewards to entice the player.

To prioritize entities and develop plans, quest givers use these attributes:

- **Drives:** How much the quest giver regards certain values, such as *Security* or *Wealth*.
- **Factions:** Relationships (like/dislike) to specific quest entities.
- **Domains:** Areas of the game world that the quest giver is aware of.
- **Dialect:** Phrases used to customize the way the quest giver speaks. For example, a sailor's greeting might be “Ahoy, matey!” while a cowboy's might be “Howdy, pardner!”
- **Rewards:** Selected using Reward System components on the quest generator entity.

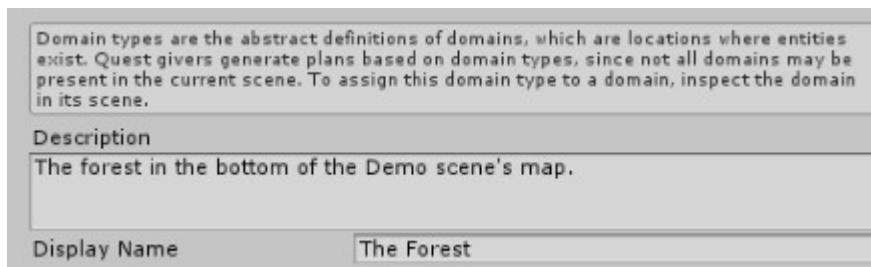
Quest Domain



A quest domain is an area of the game world that a quest giver is aware of. Quest givers can be aware of multiple domains. Domains contain quest entities. Domains don't always have to represent physical areas; they can represent anything that can contain quest entities, such as a character's inventory. Examples of quest domains are:

- A farmer's field.
- The farmer's inventory.
- Contested space near a space station.

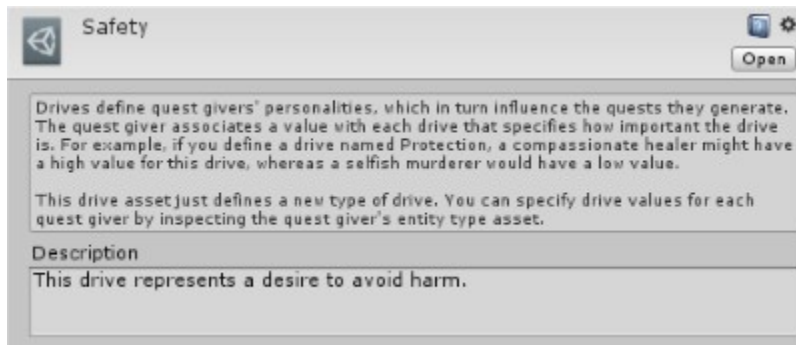
Domain Type



Domain types are the domain equivalent to quest entities' entity types. They're abstract versions that the quest generator can use to plan ahead.

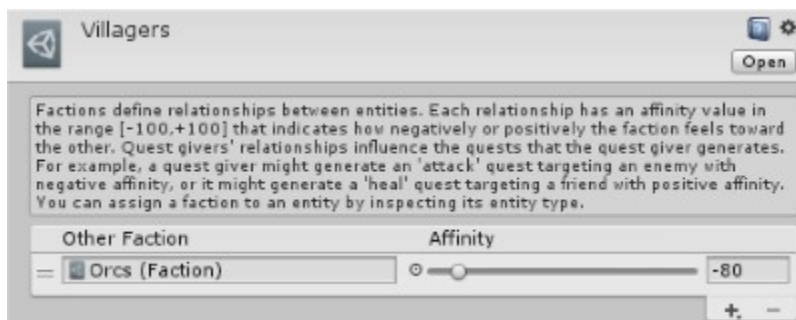
The following sections describe how to set up the quest generator elements in your project.

Drives



Drives are very simple assets with only a **Description** field that's for your internal reference. They're important because actions' motives associate values with drives. For example, the motive of killing the NPC's enemy could associate a high positive value with the Safety drive.

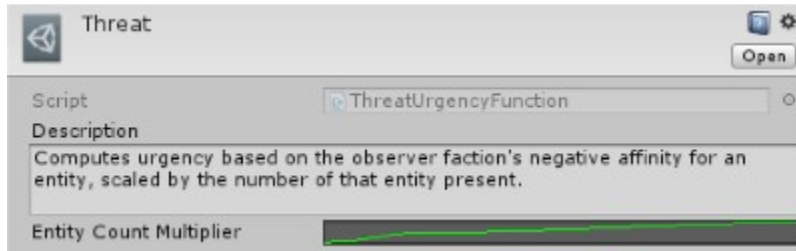
Factions



Factions define a set of relationships. In the Villagers faction shown above, villagers have a poor (-80) relationship to Orcs.

You'll almost certainly want to define more factions. Quest Machine supports third-party integrations for more advanced faction management from products such as Love/Hate.

Urgency Functions

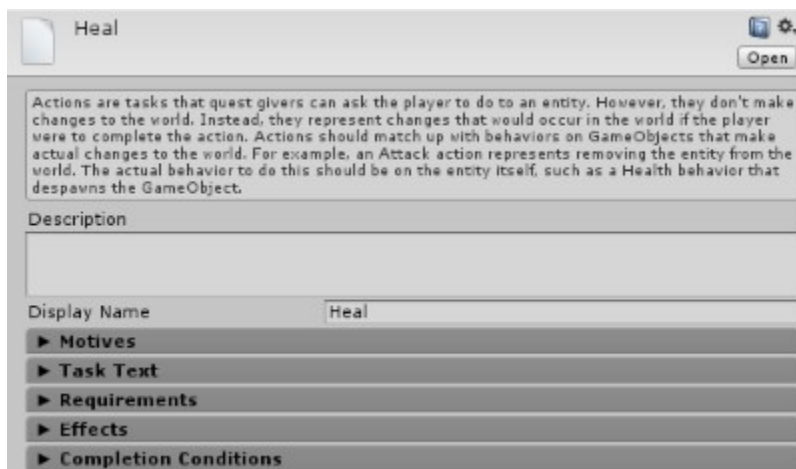


Urgency functions tell the quest generator how important an entity is to the quest giver.

Quest Machine ships with these urgency functions:

- **Threat:** Based on the quest giver's negative affinity for an entity. For example, since villagers have -80 affinity to Orcs, a villager would give an Orc an urgency score of 80.
- **Faction:** Based on the quest giver's positive affinity for an entity.
- **Literal:** The quest giver always gives the entity a specific urgency score.
- **Drive Alignment:** Based on how well the observer's and entity's drive values align.

Actions



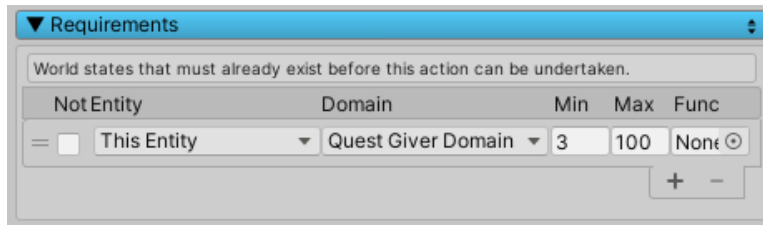
Actions are the things that questers can do to entities. For example, a dry plant can be *watered*, a gold vein can be *mined*, or a warp engine can be *powered on*. Actions are critical to quest generation because they define the tasks that the quest can ask the quester to perform. To be included in quest generation, an entity type or its parents must have at least one action that can be done to it. In a generated quest, each action corresponds to a quest node.

Actions are the most complex of the assets involved in quest generation. They're composed of:

- **Motives:** Text that explains why the quest giver wants this action done, along with drive values to help match the right motives to the quest giver's personality.
- **Task Text:** Text to show in UIs when the quest node is in various states.
- **Requirements:** World model conditions that must be true in order to start this action.
- **Effects:** Expected changes in the world model when this action is complete.
- **Completion Conditions:** How the quest knows when the action is complete.

Action Requirements

The typical way to define requirements for an action is to specify that a certain amount of an entity type must be in a domain type. In the screenshot below, the action is only available if 3 of the entity type are present in the quest giver's domain (e.g., its inventory or its house):



The screenshot shows a 'Requirements' panel with a blue header. Below the header is a text box stating 'World states that must already exist before this action can be undertaken.' Below this is a table with columns: 'Not Entity', 'Domain', 'Min', 'Max', and 'Func'. The first row contains a checkbox, 'This Entity', 'Quest Giver Domain', '3', '100', and 'None'. There are '+' and '-' buttons at the bottom right of the table.

| Not Entity | Domain | Min | Max | Func | |
|--------------------------|-------------|--------------------|-----|------|------|
| <input type="checkbox"/> | This Entity | Quest Giver Domain | 3 | 100 | None |

However, you can also tie your own code to requirements by defining a custom requirement function. A requirement function defines a condition that must be true in order for a quest generator to use an action.

Requirement functions are ScriptableObject assets. To add a requirement function to an action, create a requirement function asset, and assign it to the action's Requirements section > Func field. The Templates folder contains a starter template script.

Quest Machine includes one built-in subclass: `FactionRequirementFunction` requires that one entity type's faction toward another entity type is within a specified range. You can examine `FactionRequirementFunction` for ideas. It makes use of the `EntitySpecifier` type, which lets the designer specify entity types in flexible ways (e.g., current quester, current quest giver, specific entity type, etc.).

Several of the integration packages include additional requirement functions. For example, the Love/Hate integration includes a function that checks Love/Hate's more extensive faction system.

How to Set Up Quest Generator Assets

This section describes how to set up the abstract elements, such as actions and quest entity types, that exist as assets in your project, apart from the active scene.

You can create and manage these assets in the Quest Generator window. To open the Quest Generator window, select menu item **Tools** → **Pixel Crushers** → **Quest Machine** → **Quest Generator**.



To create a new asset, select the tab corresponding to the type of asset and click the **New** button.

Alternatively, since the abstract elements are assets, you can create them directly in the Project view by right-clicking and selecting **Create** → **Pixel Crushers** → **Quest Machine** → **Generator** → *type*. You can also move them around to different folders to organize them.

Entity Type Images

If you've assigned images to your entity types, you'll need to add those images to your quest database also. This will allow Quest Machine's save system to access the images when loading a saved game even if the entity type isn't currently in the scene. To add them to your quest database, inspect the quest database. Expand **EntityType Images**, and click **Scan EntityTypes....** Then specify the folder containing your entity types. If you haven't assigned images to your entity types, you can skip this step.



How to Set Up Quest Generator Components

Once you've done all the hard work setting up the generator assets, setting up the scene components is very easy.

Quest Entity

To set up a quest entity, add a **Quest Entity** component and assign an entity type. In the Demo scene, Orcs, Wolves, Rabbits, and Carrots are quest entities.

Quest Domain

To set up a quest domain, create a GameObject with a trigger collider. Then add a **Quest Domain** component and assign a domain type. In the Demo scene, the carrot field and the forest are domains.

Quest Generator Entity

To set up a GameObject, such as an NPC, that generates quests, add a **Quest Giver** or **Quest List Container** component, a **Quest Generator Entity** component and one or more reward systems, which are explained below.

If you tick **Require Return to Complete**, quests will have a final condition node that requires the quester to “turn in” the quest by returning to the quest giver for a last bit of dialogue. The quest generator will use a special tag: {Return to}, as in “[Return to] Captain Molly”, when generating quest text. You can assign a text table to the quest giver to use different text than “Return to” as the value of this tag; otherwise it will use the text “Return to”.

Note that a Quest Generator Entity will not generate a quest for a goal entity if it already has an offerable quest for that entity.

In the Demo scene, Sir Goodwin and Captain Molly are quest generator entities.

Reward Systems

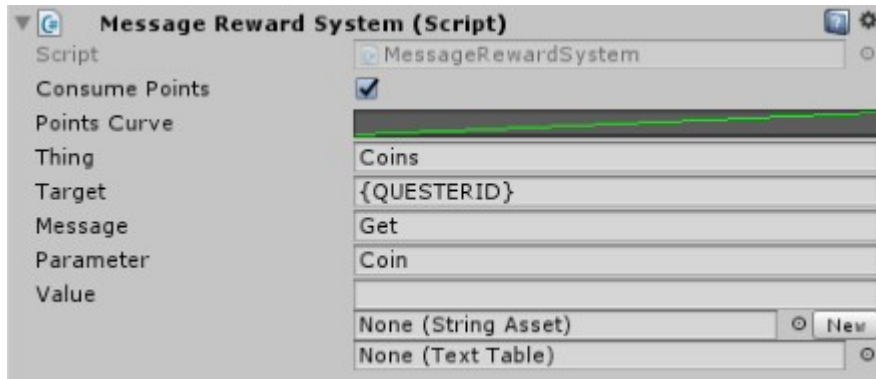
Reward systems are the mechanisms through which quest generators decide what rewards to offer and how to grant them when the player completes the quest.

Every entity type has a **Level** value that indicates its relative power or difficulty. The reward value of a quest is equal to the number of target entities times their power level. For example, the Orc entity type's Level is 5. If the quest requires the player to kill 3 Orcs, the quest's reward value is 15 points ($= 3 \times 5$). To determine rewards, the quest generator asks its reward systems to use up points until they're all used.

Quest Machine ships with two reward systems, but you can add more by creating a subclass of **RewardSystem**. The simple process is documented in the Scripting chapter.

Message Reward System

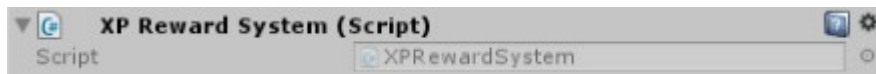
The **Message Reward System** grants a reward by sending a message with a count.



In the example above, the reward system grants coins. The number of coins is based on the quest's reward value. The reward text will be "x Coins". Upon completion, the quest will send a message "Get:Coin" with an integer value equal to the number of coins. It assumes your player listens for the message "Get:Coin". (For example, the demo scene's DemoInventory script listens for this.)

XP Reward System

The **XP Reward System** is much simpler. It grants XP without consuming reward value points.



The reward text will be "x XP". Upon completion, it will send a message "Add XP:x". It assumes that your player has a script that listens for the message "Add XP".

Quest Generator Settings

You can tweak the performance of the quest generator by inspecting the **Quest Machine** GameObject's **Quest Machine Configuration** component. The **Generator Settings** foldout has these options:

- **Max Simultaneous Planners:** Limits simultaneous quest-planning processes.
- **Max Goal Action Checks Per Frame:** When evaluating urgency of known entities, check this many actions on the goal entity each frame.
- **Max Steps Per Frame:** When planning the steps needed to accomplish a quest, evaluate this many possible steps each frame.

If your quest giver observes domains containing large numbers of entities that offer large numbers of actions, you can reduce the numbers above to eliminate stutter. On platforms with more CPU power, or if your scenes are simpler, you can typically increase these numbers.

Quest Generation Setup Example Step By Step

This section contains step-by-step instructions for setting up some procedural quest generation elements. In this example, we'll set up a giant rabbit as the nemesis of Johann the Farmer in the Demo scene. If Johann detects the giant rabbit and no other entities take precedence for him, Johann should procedurally generate a quest to deal with the giant rabbit.

1. Select menu item **Tools** → **Pixel Crushers** → **Quest Machine** → **Quest Generator** to open the Quest Generator window.
2. On the **Entities** tab:
 - Click the **New** button to create a new entity type. This will add an entity type asset named *New Entity Type*.
 - Click on the *New Entity Type* line to select it in the Project view.
 - Rename the asset to *Farmer*.
 - Click **New** again to create another entity type.
 - In the Project view, rename *New Entity Type* to *Giant Rabbit*.
 - Rename the asset to *Giant Rabbit*.
 - In the Inspector view, set the **Display Name** to *the Giant Rabbit*.
 - For the **Image**, select *rabbit0001*.
 - Set **Level** to 2.
 - Next, we'll set up factions to specify the relationship between farmers and rabbits.
3. In the Quest Generator window, click **Factions**.
 - Click **New** to create a new faction asset named *New Faction*.
 - In the Project view, rename *New Faction* to *Rabbits*.
 - Click **New** again to create another faction. Rename it to *Farmer*.
 - The Inspector view shows the *Farmer* faction.
 - Click "+" to add an empty element.
 - Assign *Rabbits* as the **Other Faction**., and set **Affinity** to -90. This means the *Farmer* faction really hates the *Rabbits* faction.
 - Next, we'll review drives and urgencies.
4. In the Quest Generator window, click **Drives**. Drives are just category names to which you can assign values. Quest Machine's Demo has already defined two drives: *Compassion* and *Safety*. Those will suffice for this tutorial.
5. In the Quest Generator window, click **Urgencies**. This section shows the urgency functions that you can assign to entity types. An urgency function tells the quest generator how urgent it is to address an entity. Urgency functions are defined in script code and return a number value, which higher values are more urgent.

Quest Machine ships with three urgency function scripts: Threat, Faction, and Literal. You can also write your own urgency function scripts. For now, the *Threat* urgency function will suffice. The *Threat* function returns the negative value of the observer's affinity to the entity type. The *Farmers* faction has -90 affinity to *Rabbits*. The negative is -(-90), or +90, which is a high urgency value. This means a farmer will think it's very urgent to deal with rabbits.

Next, we'll set up factions, drives, and urgencies on our entity types.

6. Back on the **Entities** tab, click on *Farmer*.
 - Assign the *Farmers* faction to the **Faction** field.
 - In the **Drive Values** section at the bottom, click "+" twice to add two elements.
 - Assign *Safety* to the first element, and set the value to 100.
 - Assign *Compassion* to the second element, and set the value to 90.
 - This means the farmer cares very much about his safety, and is compassionate.
7. Next, click on *Giant Rabbit*.
 - Assign the *Rabbits* faction to the **Faction** field.
 - In the **Parents** section, add an element and assign *Monster*. This means *Giant Rabbit* inherits traits from the *Monster* entity type, including the *Kill* and *Polymorph* actions. (We'll go over actions very soon.)
 - In the **Urgency Functions** section, add an element and assign *Threat*.

We'll tackle actions next.

8. In the Quest Generator window, click **Actions**. Actions are tasks that can be done to entities. Actions are the biggest part of quest generation because they define the tasks that quests can be composed of.

The Demo defines some useful actions, such as *Kill*. We'll define a less violent action called **Scare**. Instead of creating a new action from scratch, we'll save some effort by making a copy of the *Kill* action.

- Click on the *Kill* action. This will select the *Kill* asset in the Project view.
 - In the Project view, click on the *Kill* asset and press Ctrl+D (or ⌘+D) to duplicate. Rename the duplicate *Scare*. Then click on *Scare* to edit it.
 - In the Inspector view, set **Display Name** to *Scare*.
 - Motives tell the quest generator if this action is a good fit for the quest generator's drive values (personality). An action can have multiple motives. In the **Motives** section, one motive is already defined because we copied the action from *Kill*. We want to make this motive more compassionate.
 - Change this motive's Compassion value to +100.
 - Change the motive text to:
 - “Please go to the {DOMAIN} and {scare away} {TARGETDESCRIPTOR}.”
 - The words in curly braces are text tags. During play, Quest Machine will replace them with the actual values that they represent.
 - In the **Task Text** section, change the {Kill} text and its variants to {Scare}.
 - We'll leave the **Completion Conditions** alone. Under the hood, our *Scare* action will function exactly the same as the *Kill* action, but we don't need to let the player know that. This will let us re-use the kill functionality already set up in the scene.
9. Back on the **Entities** tab, click on *Giant Rabbit*. Then add the *Scare* action to the **Actions** list.
 10. In the Quest Generator window, click on **Domains**. In our scene, the farmer will only keep eyes on the *Field* domain, whose domain type is already defined, so the current domain types will suffice.
 11. Finally we get to scene setup. Since all the data is already configured, the remaining steps in the scene are going to be much simpler. Open the Demo scene, or a copy of it.

12. Add an instance of the *Rabbit* prefab to the scene, somewhere in the carrot patch.
 - Rename it *Giant Rabbit*.
 - Change its scale to (2,2,1).
 - Inspect its **Message Events** component. **Change Messages To Send > Parameter** to *Giant Rabbit*. When the giant rabbit “dies,” this component will send a message “Kill”: “Giant Rabbit”, which the *Scare* action will be listening for.
 - Remove the **Spawned Entity** component. This is only used to help the Rabbit Spawner keep track of regular-sized rabbits.
 - Assign the *Giant Rabbit* entity type asset to the **Quest Entity** component.

13. Inspect **Level > NPCs > Villager**.
 - Add a **Quest Generator Entity** component.
 - Assign the *Farmer* entity type asset to **Entity Type**.
 - Set **Quest Group** to “Farming”.
 - Set **Domain Type** to *Village*. This represents the domain where the Villager is located.
 - Assign *Level > Domains > Field Domain* to the **Domains** list. These are the domains that the Villager is aware of.
 - Tick **Generate Quest On Start**.
 - Set **Rewards UI Contents > Heading Text** to “Reward”.
 - Add a **Message Reward System** component to the Villager. Quest generators use reward system components to determine what rewards to offer based on the point value of the quest. A quest’s point value is determined by the quantity of the target and its level. By default, the Message Reward System component is configured to tell the player that she gets a number of coins based on a points curve. You can edit the curve to give more or less coins.

14. And we’re done!
 - Play the scene.
 - Inspect the Villager. Its **Quests** list should contain a new quest titled “Scare the Giant Rabbit”.
 - If there’s an issue, inspect the **Quest Machine** GameObject and tick **Debug Settings > Debug Quest Generator**. This will tell the quest generator to log its steps to the Unity editor’s Console window. Look for anything that seems amiss. For example, the Console should show this line:
 “Quest Machine: [Generator] Farmer: Most urgent fact: 1 Giant Rabbit in Field”
 It says that the Villager detected the Giant Rabbit in one of its Domains and decided that it’s the most urgent entity to make a quest about. If you don’t see this message, check that the Giant Rabbit is inside the trigger collider area of *Level > Domains > Field Domain*.

Remember that we’re here to help! If you get stuck, please email support@pixelcrushers.com any time.

Chapter 7: Scripting

Quest Machine's Scripting Reference is at: https://pixelcrushers.com/quest_machine/api/html/

How to Control Quests in Scripts

QuestMachine Class

Most Quest Machine activity can be controlled through the static QuestMachine class. This class provides methods such as:

- GetQuestState(), SetQuestState(), GetQuestNodeState, SetQuestNodeState(), AbandonQuest()
- GetQuestInstance(), GetQuestJournal()

The [QuestMachine API reference](#) contains full details.

Note: The QuestMachine class provides methods to get quest assets and quest instances. A *quest asset* refers to the quest asset file in your project. For hand-written quests, this is the original version that you have created at design time and assigned to a quest giver's Quests list. A *quest instance* is an in-memory copy of a quest asset. There may be many quest instances of a quest asset. Typically, when an NPC quest giver starts up, it creates instances of each of its quests. When the NPC gives a quest to a quester (e.g., player), the quester receives a new instance of the quest that's a copy of the NPC's quest instance.

QuestGiver Class

On GameObjects with Quest Giver components, such as quest giver NPCs, the QuestGiver class provides methods such as:

- StartDialogue[WithPlayer]()
- GiveQuestToQuester(), GiveAllQuestsToQuester()

The [QuestGiver API reference](#) contains full details.

QuestJournal Class

On GameObjects with Quest Journal components, such as the player, the QuestJournal class provides methods such as:

- ToggleJournalUI(), ShowJournalUI(), HideJournalUI()
- AddQuest(), AbandonQuest()

The [QuestJournal API reference](#) contains full details.

Message System

Sending Messages

To send a message to the Message System, call `MessageSystem.SendMessage()` or `MessageSystem.SendMessageWithTarget()` (to specify a target).

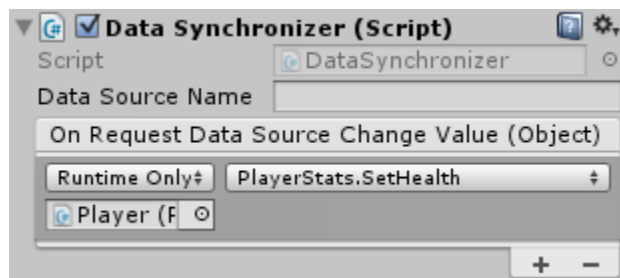
Example:

```
void OpenBlackGate()
{
    blackGateAnimator.Play("Open");
    MessageSystem.SendMessage("Opened", "Black Gate");
}
```

Receiving Messages

To allow a script to receive messages, implement the `IMessageHandler` interface. This interface has one method: `OnMessage()`. Then you can register for messages by calling `MessageSystem.AddListener()`, and unregister by calling `MessageSystem.RemoveListener()`. It's important to remove listeners before destroying the script instance; otherwise the Message System will continue trying to send messages to an invalid object.

Data Synchronizers



A *data synchronizer* uses the Message System to coordinate changes with an external data source such as one of your own scripts. It listens for the message **"Data Source Value Changed"** (also defined as `DataSynchronizer.DataSourceValueChangedMessage`) and invokes a `UnityEvent` that you can assign to your script. The parameter is an object; cast it to the type of your data. When you change the value in your external data source, call `DataSourceValueChanged(newValue)` to inform listeners.

Data Synchronizer Example:

This example synchronizes the player's gold with a quest counter. First, let's say your inventory is managed in a script named `ExampleInventory`:

ExampleInventory.cs (version 1)

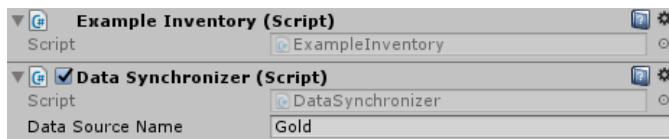
```
using UnityEngine;

public class ExampleInventory : MonoBehaviour
{
    private int m_gold;

    public int gold
    {
        get { return m_gold; }
        set { m_gold = value; }
    }
}
```

The ExampleInventory's **gold** property is the data source.

Add a **DataSynchronizer** and choose a Data Source Name. In this example, I've chosen "Gold":



When the amount of gold changes, your inventory script needs to tell the DataSynchronizer so it can notify all listeners (e.g., quest counters). To do this, modify the code to call the DataSynchronizer's DataSourceValueChanged method:

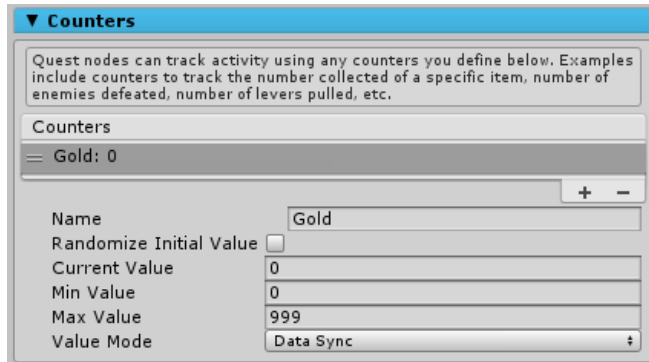
ExampleInventory.cs (version 2)

```
using UnityEngine;

public class ExampleInventory : MonoBehaviour
{
    private int m_gold;

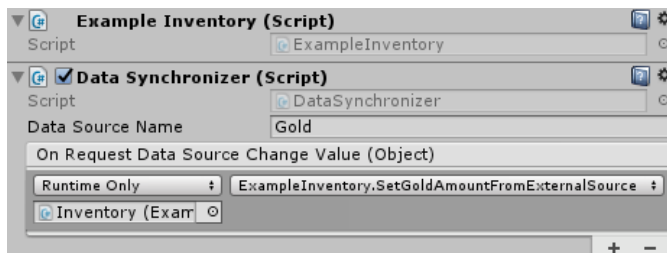
    public int gold
    {
        get { return m_gold; }
        set
        {
            m_gold = value;
            GetComponent<PixelCrushers.DataSynchronizer>().DataSourceValueChanged(value);
        }
    }
}
```

Whenever you change ExampleInventory's gold property, the DataSynchronizer will inform any listeners that are watching the data source named "Gold". For example, we could set up a counter named "Gold":



If you only need to synchronize from ExampleInventory to listeners such as quest counters, you can stop here.

It's possible to also synchronize the other way, although it's rarely used. To do this, hook up the On Request Data Source Change Value (Object) event to a method that sets gold to a value received from some external source. For example, I named the method SetGoldAmountFromExternalSource() and hooked it up like this:



Here's the script:

ExampleInventory.cs (version 3)

```
using UnityEngine;
```

```
public class ExampleInventory : MonoBehaviour
{
    private int m_gold;

    public int gold
    {
        get { return m_gold; }
        set
        {
            m_gold = value;
            GetComponent<PixelCrushers.DataSynchronizer>().DataSourceValueChanged(value);
        }
    }

    public void SetGoldAmountFromExternalSource(object amount)
    {
        gold = (int)amount;
    }
}
```



To change the value of gold to 42 from an external source, send a message like this:

```
MessageSystem.SendMessage(this, DataSynchronizer.RequestDataSourceChangeValueMessage, "Gold", 42);
```

How to Create Quests in Scripts

QuestBuilder

To make a quest completely from scratch, use a QuestBuilder object. This works similarly to the .NET StringBuilder class. It provides methods for adding different types of nodes and content. While you're using QuestBuilder, a copy of your quest is always available for modification in the QuestBuilder.quest property. When you're done, call QuestBuilder.ToQuest(). An example script is in Quest Machine ► Templates ► QuestBuilderExample.cs.

If you're writing an editor script and want to save the quest instance as an asset in your project, call QuestEditorAssetUtility.SaveQuestAsAsset().

Generate quests

To tell a quest generator entity to generate a new quest, call QuestGeneratorEntity.GenerateQuest(). You can also create a QuestGenerator object manually to generate a quest without a QuestGeneratorEntity.

Quest generator entities rely on domains to generate a mental model of the world. QuestGeneratorEntity has a delegate method named **updateWorldModel**. You can assign a method to this delegate to further update the world model before it's passed to the generator. Using this method, you can remove facts from the world model, or add facts that aren't detected in domains.

How to Create New Types

You can define your own types of conditions, actions, and UI content. To do so, subclass these classes:

- QuestCondition
- QuestAction
- QuestContent

The Templates folder contains fully-commented starter templates.

Reward Systems

The Templates folder also contains a fully-commented starter template for reward systems, which are used by the quest generator.

Urgency Function

You can also define your own urgency functions by subclassing UrgencyFunction.



Quest Events

See the [API](#) for various quest hooks, such as `Quest.stateChanged` and `QuestListContainer.questAdded`.

The **Quest List Events** component adds `UnityEvents` to Quest List Containers, including Quest Journals (e.g., players).

Save System

Quest Machine's save system has its own manual, located in the Documentation subfolder.

Chapter 8: Third Party Integration

Quest Machine ships with third party support for the products below. Each third party support package contains its own documentation.

To install third party support:

1. Check **Plugins ► Pixel Crushers ► Common ► Third Party Support** for a unitypackage, such as **Dialogue System Support.unitypackage**. Import this package first if it exists. Not all integrations have unitypackages in Common.
2. Import the integration package in **Plugins ► Pixel Crushers ► Quest Machine ► Third Party Support**, which will typically be named the same (e.g., **Dialogue System Support.unitypackage**). Remember to check the Common folder first, and import any corresponding packages there *before* importing from Quest Machine's Third Party Support folder.
3. Read the integration's PDF manual, which will be added when you import the unitypackage.

Current third party integrations include:

- **Adventure Creator** (© Icebox Studios)
- **articy:draft** (© articy Software GmbH & Co.)
- **Compass Navigator Pro** (© Kronnect)
- **Corgi Platformer Engine** (© More Mountains)
- **Dialogue System for Unity** (© Pixel Crushers)
- **DMMAP Minimap System** (© Dylan Meville)
- **Emerald AI** (© Black Horizon Studios)
- **Investor character controllers** (© Invector)
- **Inventory Engine** (© More Mountains)
- **Inventory Pro** (© Devdog)
- **Love/Hate** (© Pixel Crushers)
- **Opsive Ultimate Character Controllers** (© Opsive)
- **Opsive Ultimate Inventory System** (© Opsive)
- **ORK Framework** (© Gaming Is Love)
- **PlayMaker** (© Hutong Games)
- **Rewired** (© Guavaman Enterprises)
- **TopDown Engine** (© More Mountains)
- **uMMORPG** (© vis2k)
- **uSurvival** (© vis2k)

Chapter 9: Multiplayer & NPC Questers

Quest Machine supports multiplayer games and NPC questers. You may have noticed already that Quest Machine typically references “quester” instead of “player”. The player (or players in multiplayer) is just another quester, which is any agent that can accept and complete quests. Each quest instance records the quester's ID, which should be unique for each player or NPC quester. (For a quick way to get a unique ID, you can assign `System.Guid.NewGuid().ToString()`.)

To use the quester's display name in UI text, use the {QUESTER} tag:

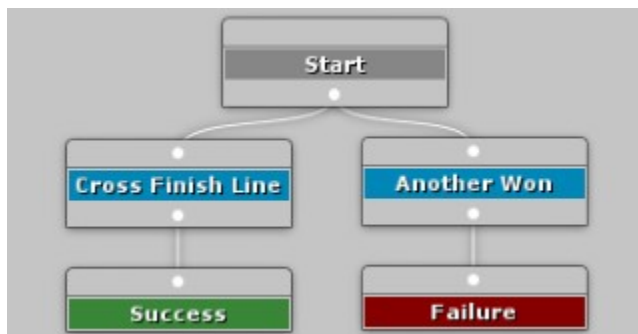
“Scare away those rabbits, {QUESTER}, before they eat my carrots!”

The {QUESTERID} tag refers to the quester's ID.

One catch with multi-quester games is that you need to keep Message System senders and targets in mind. Let's say you write a quest to Discover Atlantis. Your quest listens for the message “Discover” with the parameter “Atlantis”. If you set the Message Quest Condition's **Required Sender** dropdown to *Any*, then anyone who sends this message to the Message System will advance the quest. To ensure that the quest only advances when the proper quester sends the message, make sure to set the **Required Sender** dropdown to *Quester*. In a single-player game you can probably be less careful about this, since typically only the single player will be sending these messages.

NPCs can also be questers. So an NPC can pick up a quest to retrieve the Sword of Orc-Killing from the chest in the tower, dip it in the Enchanted Pool, and kill 5 Orcs with it, and Quest Machine will recognize when each task is done, with this caveat: Quest Machine doesn't actually animate the NPC or provide a combat system; it only generates the quest and manages it until it's done. In addition, the quest journal window is designed to be interactive, so it's really only useful for players.

It's possible to allow multiple questers to undertake the same quest but only one can complete it. As soon as one completes it, it automatically becomes “failed” for the others. To configure this, you'll typically have two paths, as in the example below, “The Race”:



In **Cross Finish Line**'s True actions, use the Set Quest Node State action to set **Another Won**'s state to Disabled, and use the Message action to send a message such as “Someone Won”.

Configure **Another Won**'s condition to listen for the message “Someone Won”.

Appendix 1: Localization and Text Tables

For localization and dialects, Quest Machine uses the Pixel Crushers common library's Text Table assets. For information on how to create and edit Text Tables, please see the separate document **Text_Table_Manual.pdf** in the Common / Documentation folder.

Appendix 2: Save System

Quest Machine can use the Pixel Crushers common library's Save System. For instructions on how to set up the Save System, please see the separate document **Save_System_Manual.pdf** located in the Common / Documentation folder.

The demo scene uses the Save System. In the demo scene, the main Save System GameObject is named **Save System**. The demo menu calls the static script method `SaveSystem.SaveToSlot()` to save the game and `SaveSystem.LoadFromSlot()` to load the game.

Quest Machine Savers

The Save System uses special components called Savers that know how to save specific data in the scene. Quest Machine components that maintain quest lists (that is, Quest Givers and Quest Journals) are also Savers.

When saving a quest that was instantiated from a quest asset, Quest Machine will save a minimal amount of data, frequently under 100 bytes. This very compact format comes with one restriction: you should not add new nodes to the quest after saving the game. However, you can certainly add new quests themselves to your game after saving the game.

When saving a procedurally-generated quest, Quest Machine saves all of the quest's content.

When loading a quest, Quest Machine does *not* re-run actions (to prevent it from re-giving rewards, etc.), so any changes to the game world that the quest makes should be captured in other Saver components.