Table of Contents

Game Creator

Getting Started

Documentation

Inventory

Dialogue

Advanced

Introduction

FROM ZERO TO HERO

What is Game Creator?

Game Creator is an ecosystem that gives Unity developers a world class technology platform from which they can build games that work seamlessly across multiple platforms quickly and efficiently.

In this pages you'll learn everything you need to start using **Game Creator** like a PRO.

Why should I use Game Creator?

Whether you are an *indie developer aficionado* or a *Unity guru*, **Game Creator** works for you.

For non-developers

- Game Creator is packed with a full featured **Visual Scripting Tool** that allows to create complex interactions using user-friendly and non-technical elements such as **Hotspots**, **Actions**, **Events** and **Triggers**
- Save & Load the state of your game using the **variables**
- Use the **Character** component to bring your characters to live. Define simple patrol patterns, make them chase the player or create entire daytime routines!
- Give your games a professional look using the **Camera Motors**, which allow to change the camera behavior in real time! For example, you can use an *adventure camera* to follow the player and switch to an *animated camera* to show that awesome sword inside the chest the player just picked!

For developers

- Extendable **open API** to create custom **Actions** with built-in tools to make the process even faster
- Integrate it with your favorite tools (like **PlayMaker**, **UFPS**, ...)

> Missing a feature? **Open an issue** in our help desk and we will help you.

How do I get started?

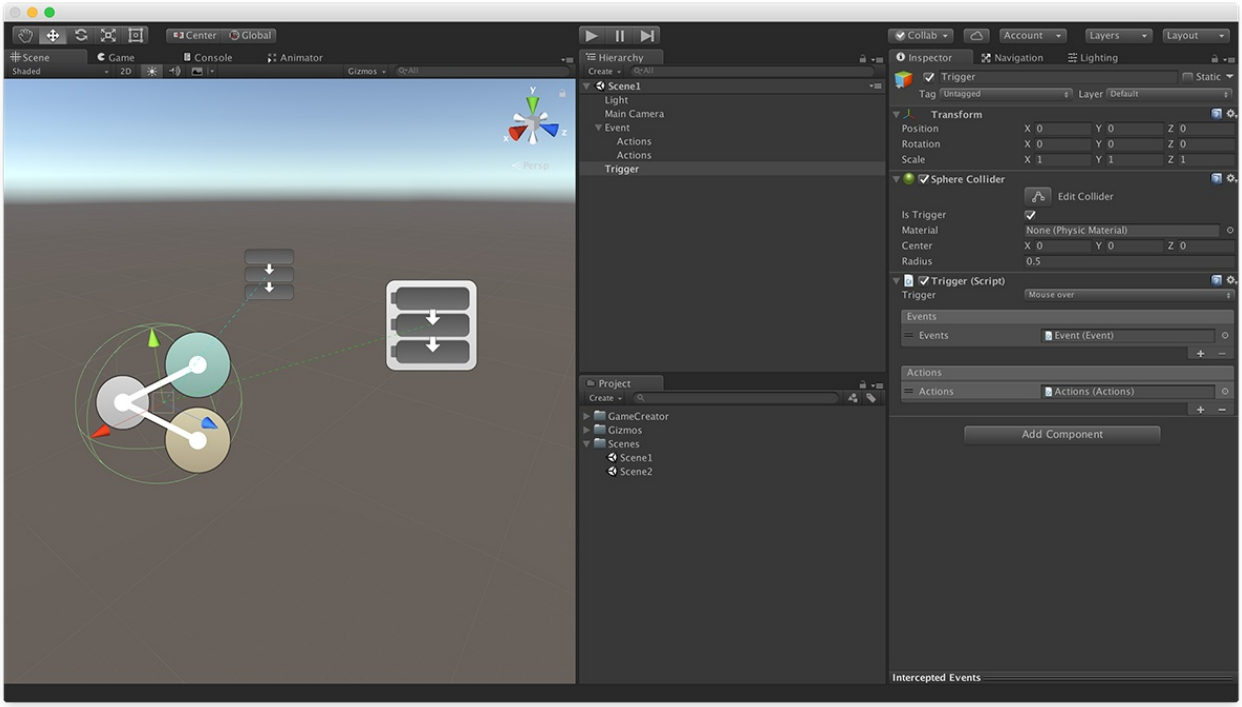
In this pages you'll learn everything you need to start using **Game Creator** like a PRO.

There are a couple of ways you can get up to speed on making games with **Game Creator**:

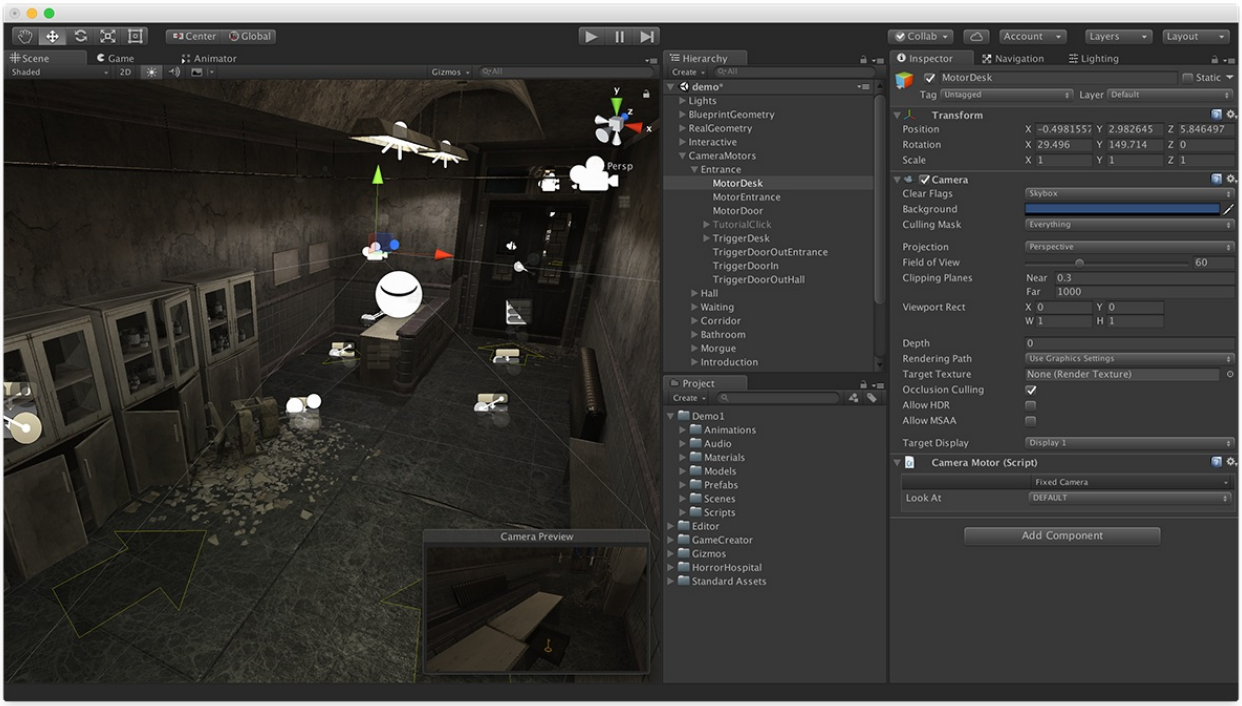
- Watch the video tutorials (~14 min)
- Read this documentation

> Feel free to send us a message at hello@catsoft-studios.com with your feedback.

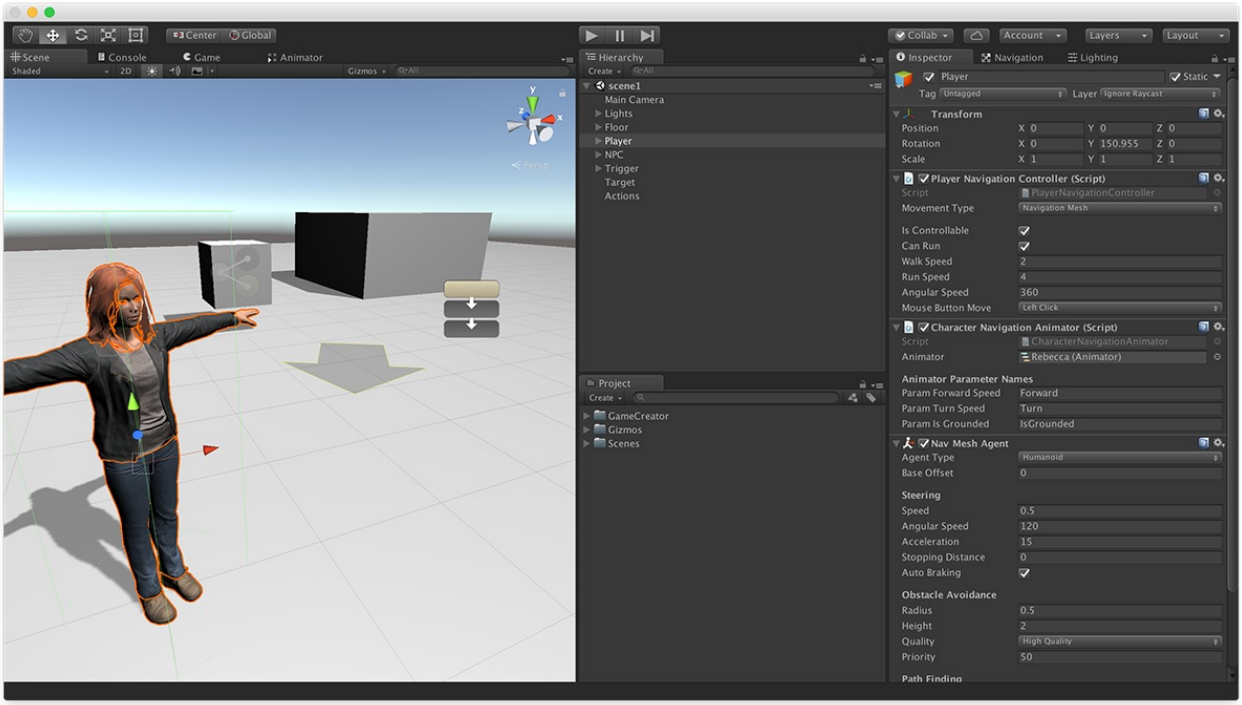
Game Creator Screenshots



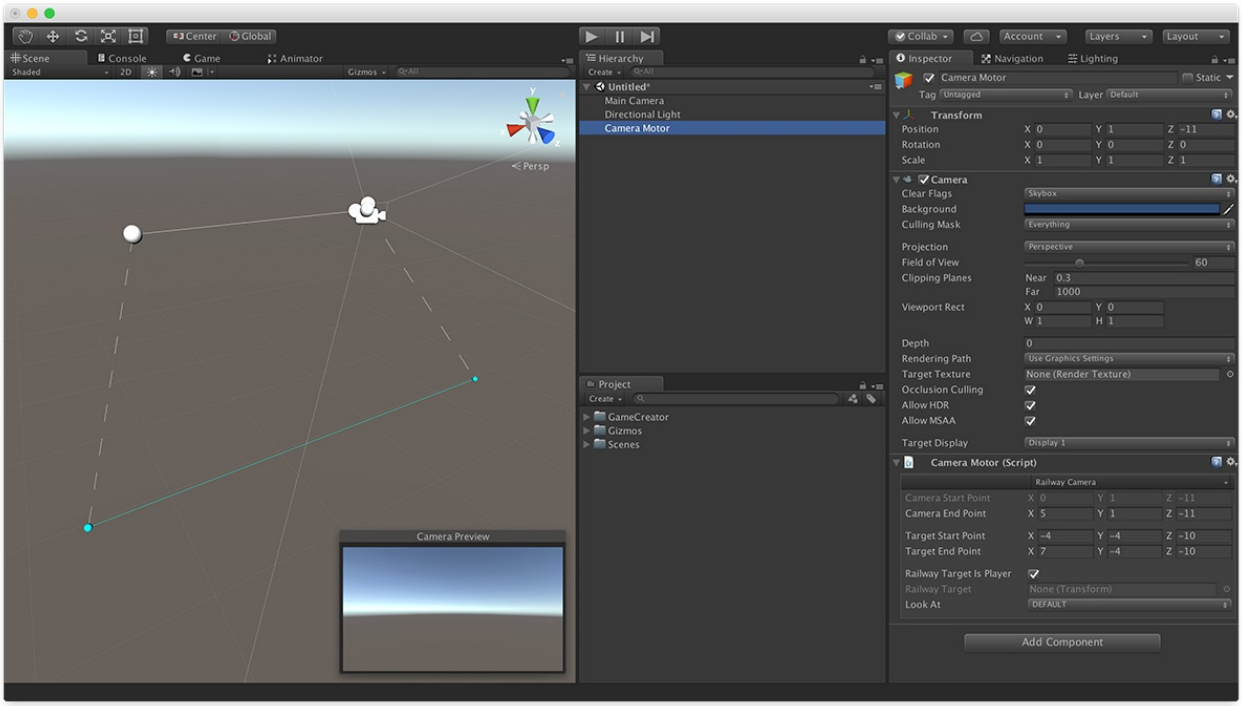
SC.1: A **Trigger** linked to an **Event** and an **Action**



SC.2: The entrance of the demo in the *Editor*



SC.3: Playground scene for the **Character Navigation** module



SC.4: Handles of a *Railway* camera type of the **Camera**

Setup

Setting up **Game Creator** is really easy.

1. Download the latest `gamecreator.unitypackage`
2. Open your Unity project or create a new one
3. Import the package in **Unity**

> If you got the **Game Creator** package from the **Asset Store** you can skip the first two steps

Once you have **Game Creator** imported in your project you can kickstart your game, though if it's the first time you're using it, we recommend you play with the **sample scenes** and follow the *Getting Started* section.

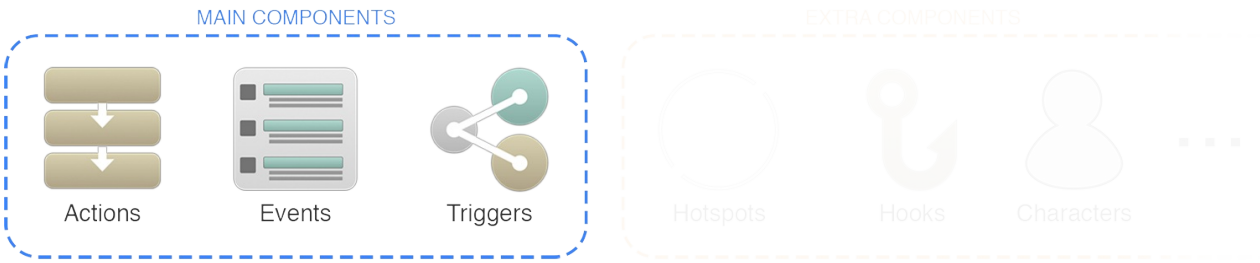
> **IMPORTANT!** The *Game Creator* folder and its content must always be at the root of the project and should never be moved.

Overview

The heart of **Game Creator** is composed of 3 components. On top of these, there are others which add an extra layer of flexibility and help you build your games even faster.

Main Components

Game Creator has 6 main components: **Action**, **Events**, **Triggers**, **Hotspots**. **Cameras**, **Characters** and the **Player**.



Actions

A set of *instructions* that are sequentially executed. For example:

- Move *player* next to *chest*
- Play `Open` animation on *chest*
- Give the player *10 coins*

Events

Conditions the execution of **Actions**. For example, you can check if a *chest* has been opened. If so, don't increase the player's *coins*, but show the message: `This chest has already been looted`.

Triggers

Triggers react to inputs and can execute multiple **Actions** and **Events**. For example, you can detect when the player enters a *lava zone* and execute an **Action** that makes the player take damage.

High Level Components

The following components help you build your game even faster.

Hotspots

Hotspots are like a *Swiss Army Knife* for interactive elements. It contains multiple functionalities than can be turned *on* and *off* depending on the scenario. These functionalities include **moving the player's head** towards the **Hotspot** when he's within a radius or change the **cursor icon** when the mouse hover it.

Hooks

If you are a programer, **Hooks** allow you to easily access important components from the class name following the Singleton pattern. For example, to get the current player you can use `HookPlayer.Instance`.

> **Hooks** are only useful if you are a programmer and want to extend core functionalities.

Cameras

Cameras control how the world is viewed by the user. You can create cameras that behave like surveillance cameras, others that follow the player and even animated cameras for cutscenes.

Character

The **Character** component is a very flexible one that allows to easily create game characters that can interact with **Actions**. Define patrol patterns, complex daytime routines and even AI behaviors such as following the player when his health is low, make them walk when entering houses and run otherwise, etc...

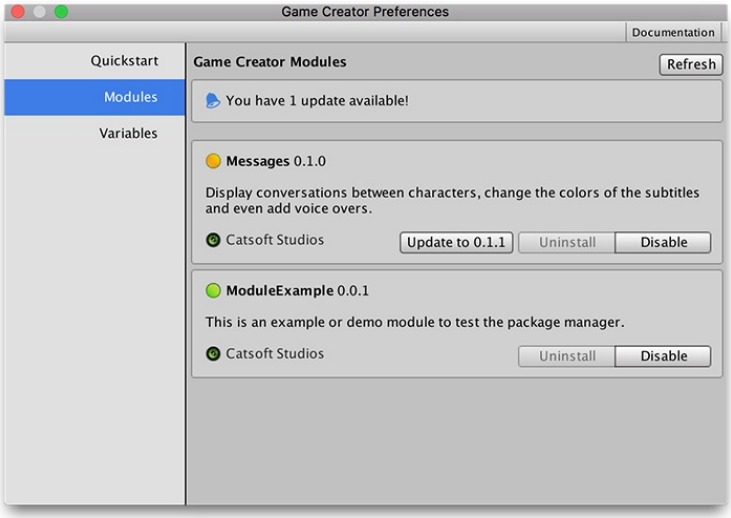
Player Character

The **Player Character** inherits all the functionality from the **Character** component and adds a set of options to control its movement with the *mouse*, *keyboard* or *joystick*.

Module Manager

Game Creator has been built with modularity in mind: There is a *Core* and then there are the rest of the *modules*. These modules can range from a simple third-party tool integration (such as PlayMaker, Behavior Designer or any other Unity package) to a custom functionality like an Inventory system or a Dialogue framework with multiple choices.

In order to keep a clean project, **Game Creator** comes with a *Module Manager*, which is a system that allows to enable/disable and uninstall different modules right from within the Editor.

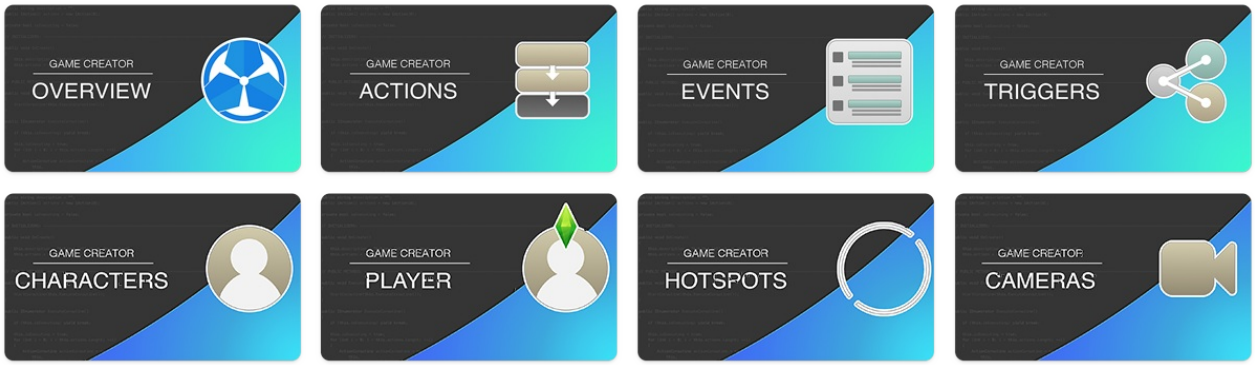


For example, here's a module called **Messages**, which allows to display conversations between *Characters* and have a voice over. If you feel you don't need this module anymore, you can either **Disable** it (in case you want to enable it later) or completely remove it from your project by clicking **Uninstall**.

> You can open the **Module Manager** clicking on the *Toolbar*/Game Creator/Preferences or pressing the **Cmd** + **M** (or **Ctrl** + **M** if you're using a **Windows** PC).

Quick Start

Welcome to our *Quickstart guide* on how to create games with **Game Creator**. To familiarize with the **Game Creator** tools we highly encourage you to read the Overview section and watch the 15 minute Getting Started video tutorial series.



Example Scenes

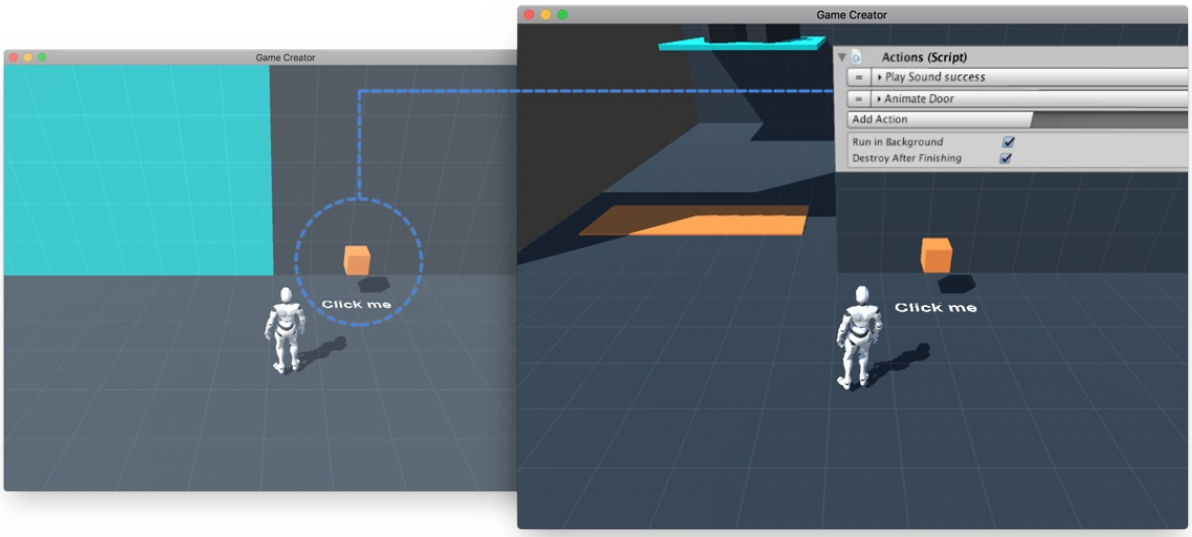
The **Game Creator** package comes with playable scenes that provide examples on how to create common mechanics or gameplay situations. They are divided into different sub-scenes (named *Example-1*, *Example-2*, ...).

> Checkout the WebGL version of the demo

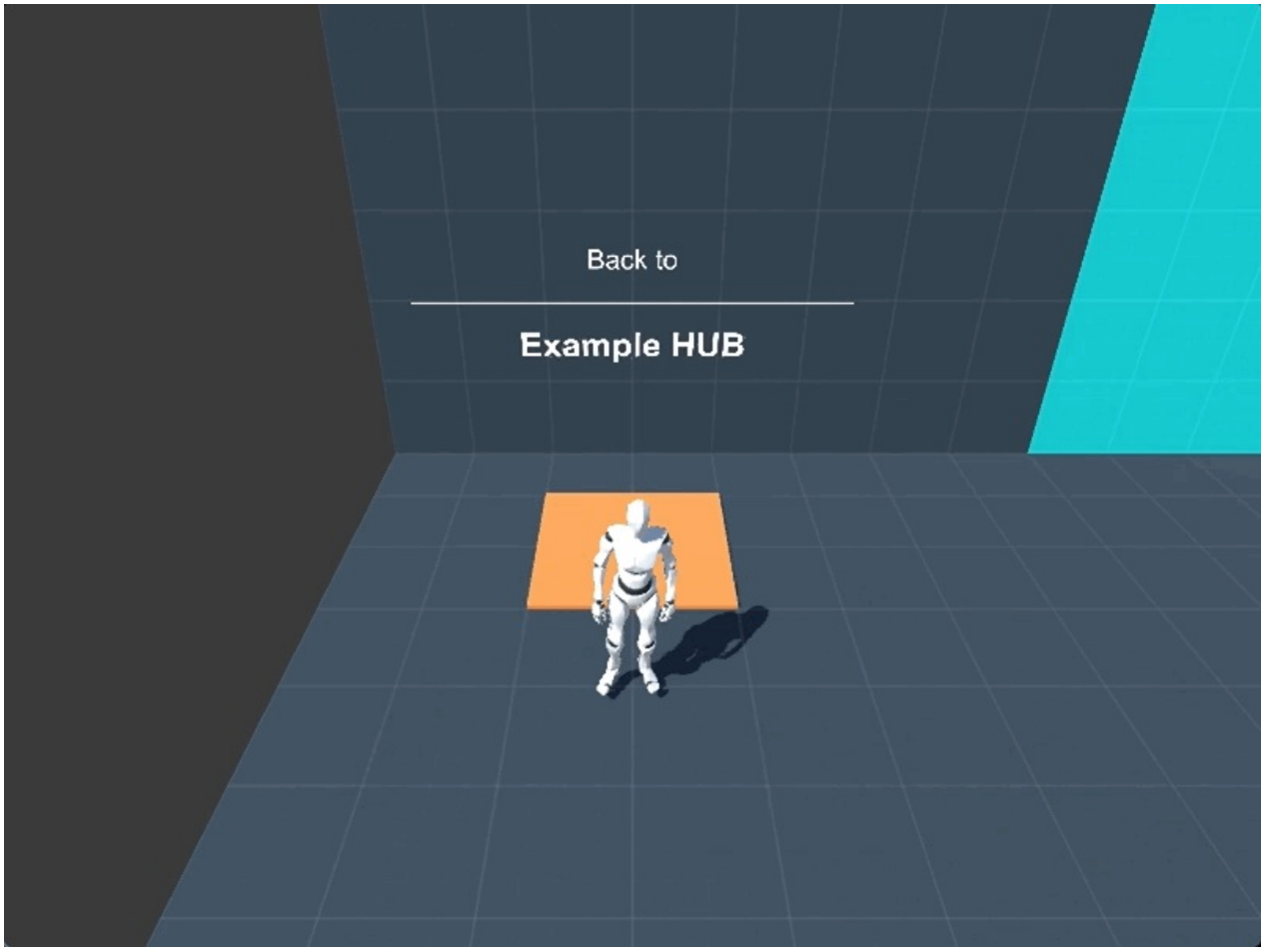
To see these examples, first you need to **Enable** the module that contains them. To do so, press `Cmd + Alt + M` or click the `Game Creator -> Module Manager` on the toolbar. There you can select the **Examples** module and click the *Enable* button.

> **Game Creator** has a *Module Manager* which allows to add and remove different parts. For example, if your game needs a complex Dialogue system, you can install the **Dialogue module**. If you later decide you won't need it, you can safely disable and remove it.

The first part of *Example-1* shows how to create a **Trigger** that when clicked executes an **Action** that opens the door and reveals the path to the next section.



> Most of the examples also show you a *snippet* of the component so you easily understand how to create such behavior.



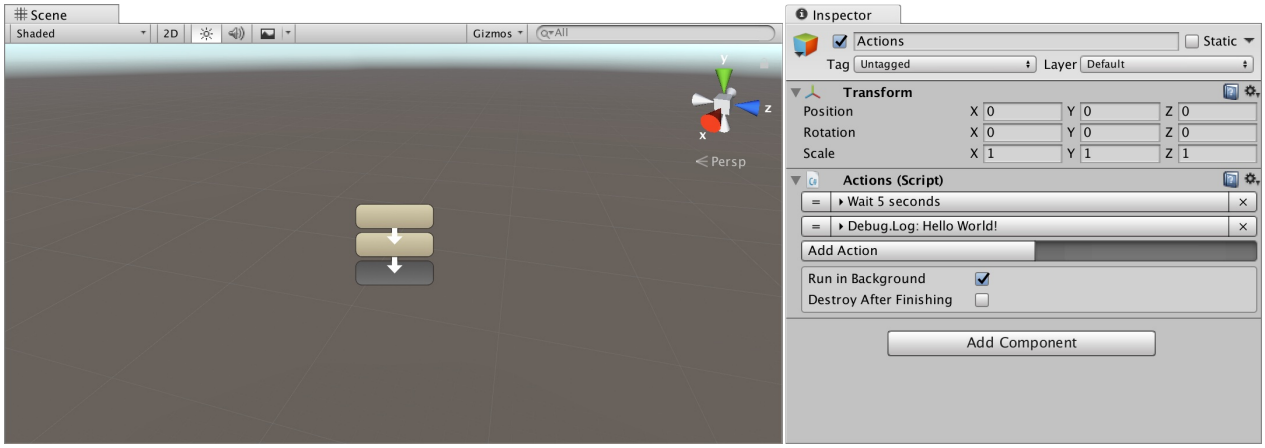
What's next

Once you played with the example scenes and overviewed how they are made you should be able to start creating games.

> If you have any questions or feel this documentation lacks something important, contact us at hello@catsoft-studios.com

Actions

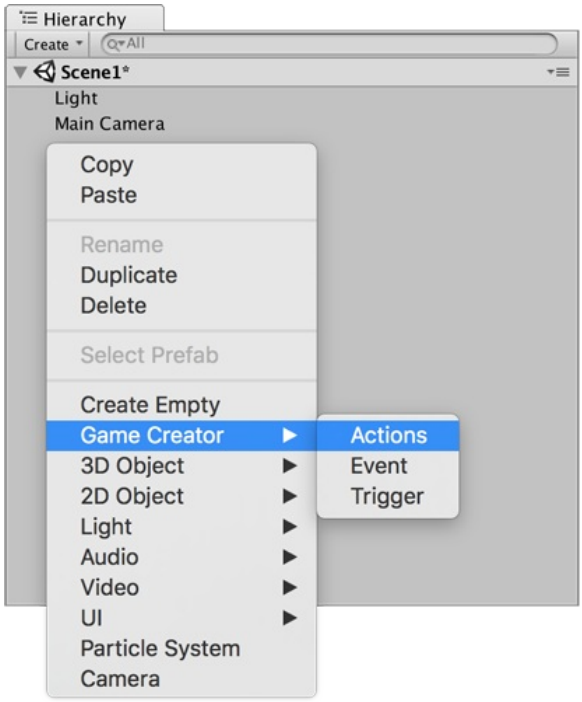
Actions are the main feature behind **Game Creator**. They allow to animate characters, activate interruptors, move objects, etcetera.



> We're constantly incrementing the number of **Actions** available. Feel free to open a ticket if you are missing one.

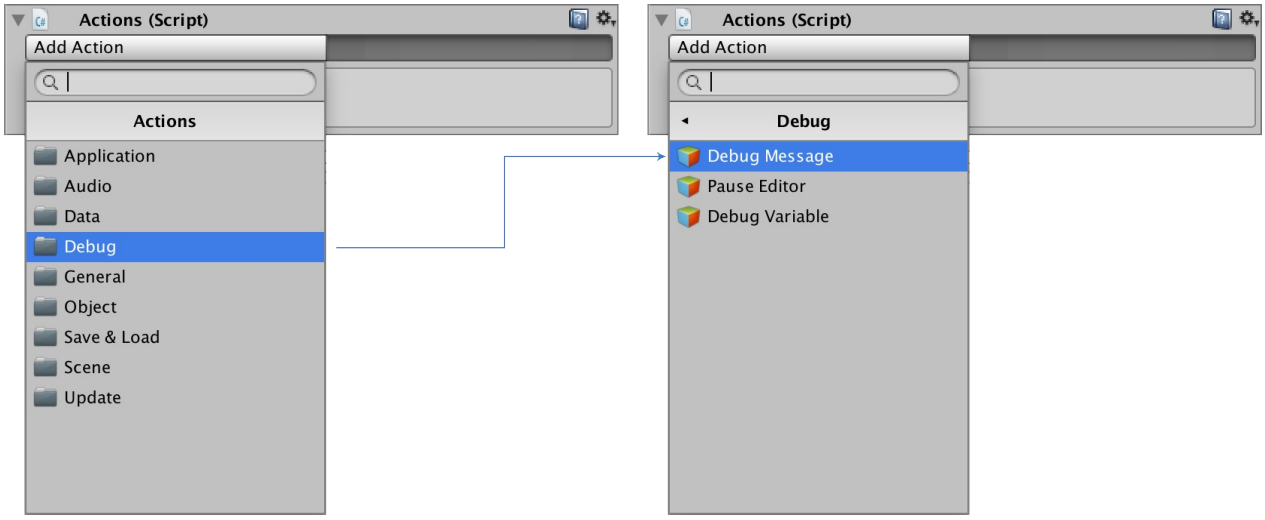
Creating Actions

To create an **Action container** (or **Actions** object) right-click in the *Hierarchy Panel* and navigate to `GameCreator → Actions`.

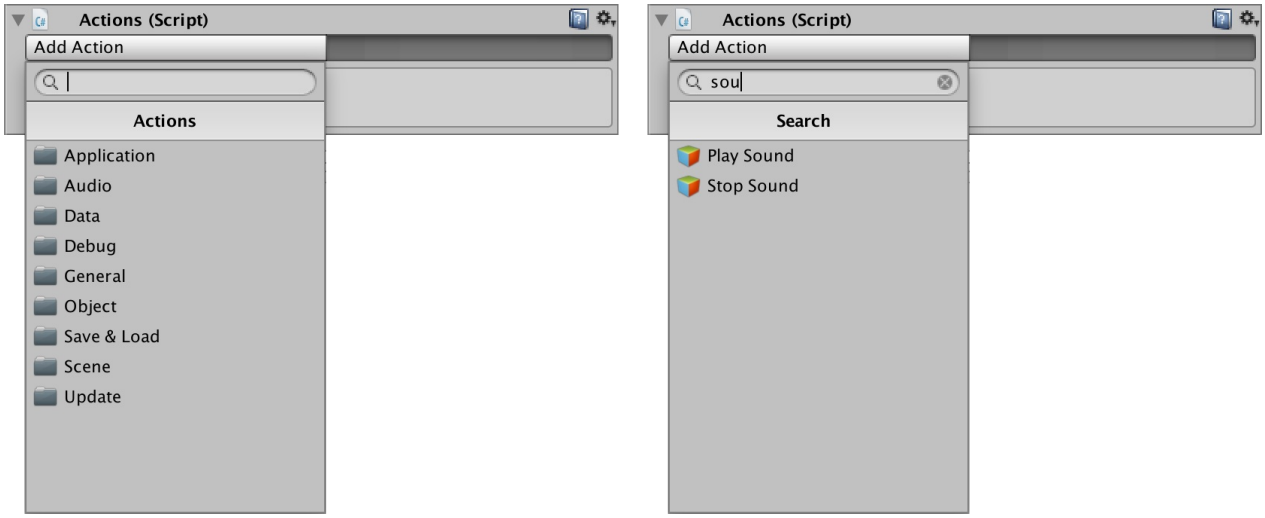


> Alternatively you can create an *Empty GameObject* and add an **Actions** component.

To add **Actions** (instructions) to a container, click on the `Add Action` button and a dropdown menu will appear. All the **Actions** are organized in different categories.



You can also use the *search box* to easily find the **Action** you want.

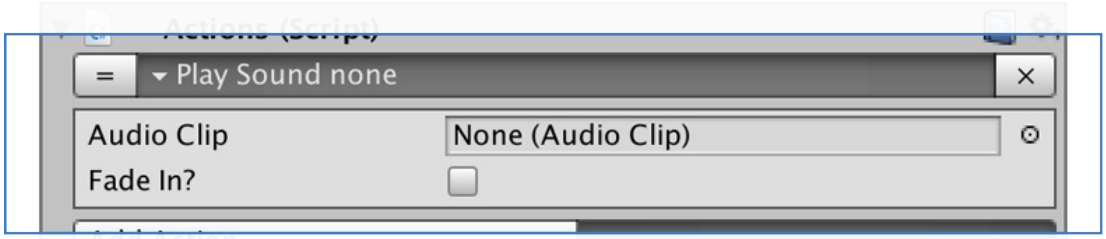


> You can also navigate the **Actions Panel** with the *keyboard*

Action Anatomy

An **Action** is a single instruction that when executed *does something*. For example, the action `Wait` puts the execution on hold for a number of seconds.

Here's an example of an action called `Play Sound`.



- Collapse/Expand it clicking on the Action's header to reveal its options
- Reording can be done by dragging the `=` symbol
- To remove an Action click on the `x` button

> Try to familiarize with the available **Actions**. Once you know their name you can create complex interactions in blink.

Extra Options

You can configure the behaviour of an **Actions** object at the end of the component.

Run in background

When the `Run in background` checkbox is marked the Action will always be executed when called. If the checkbox is left in blank, if another **Actions** object is being executed and is set to run in the foreground, the first one won't be executed.

> Two Actions can't be executed at the same time if they both have the `Run in background` checkbox unmarked.

Destroy After Finishing

Game Creator is all about fast development. If you want to destroy an **Actions** object after all of its instructions have been executed, mark this checkbox.

> You can accomplish the same behavior using the `Destroy` action at the end of the container of **Actions**.

Available Actions

Here's the full list of all available **Actions** that come with **Game Creator**.

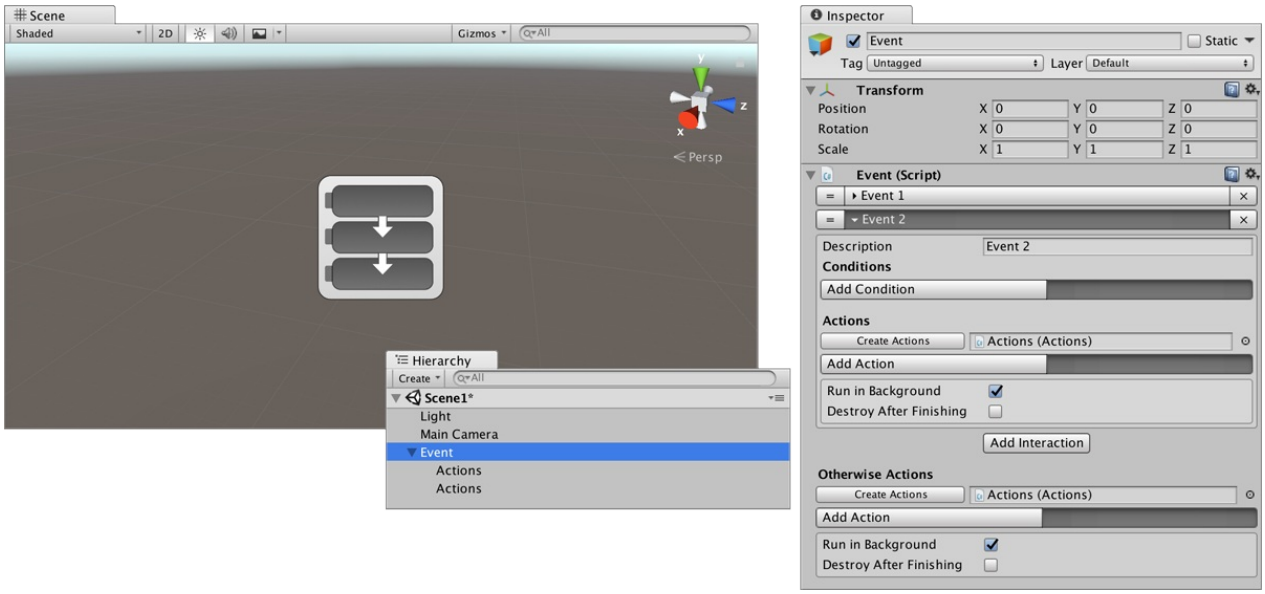
NAME	DESCRIPTION
Camera/**Change Camera**	Change between camera motors
Camera/**Camera Damping**	Add a spring-like effect to to the camera
Camera/**Adventure Camera settings**	Change different properties from a specific Adventure Camera Motor
Character/**Character Change State**	Change the character's stance state (drunk, hurt, normal, ...)
Character/**Character Gesture**	Make a Character do a gesture from a dropdown list
Character/**Character Jump**	Makes a Character Jump
Character/**Move Character**	Instruct a character to move to a position/transform/*Marker*
Character/**Change Property**	Change a character's property, such as toggling walk/run, interactivity, ...
Character/**Head Track**	Make a Character look at a certain position
Character/**Player Movement Input**	Change how the player is controlled
Character/**Character Direction**	Allows to change how the character moves
Application/**Cursor**	Change the cursor's appearance
Application/**Open URL**	Open a website in the web browser
Application/**Quit Game**	Quits the game
Audio/**Pause Audio**	Pauses/Resumes current audio sources
Audio/**Play Music**	Plays an *AudioClip* as a music track
Audio/**Play Sound**	Plays an *AudioClip* as a sound effect

Audio/**Stop Music**	Stops the music track
Audio/**Stop Sound**	Stops a sound effect
Audio/**Change Volume**	Changes the volume of all audio sources
Data/**Reset Variables**	Resets all variables to its default values
Data/**Variable**	Modifies the value of a variable
Debug/**Message**	Prints in the Editor's Console a message
Debug/**Pause Editor**	Pauses the Editor
Debug/**Debug Variable**	Prints the value of a variable
General/**Execute Actions**	Executes another Actions object
General/**Call Methods**	Executes a set of previously defined methods
General/**Call Event**	Calls an Event and waits (or not) for its execution to end
General/**Time Scale**	Modifies the Time Scale. Useful for pausing or adding *Bullet Time* effects
General/**Wait**	Waits an amount of seconds before executing the next Action
General/**Random Wait**	Waits a random amount of seconds between min and max values
General/**Comment**	Does nothing, but appears with a different color. Useful for development
Object/**Animate**	Call any Trigger/Float/Bool/Integer/String parameter of any Animator
Object/**Animator Layer**	Change the weight of any Animator's Layer weight
Object/**Destroy**	Destroys any given Game Object
Object/**Timeline**	Play, Pause or Stop any Timeline/Playable Director object
Object/**Instantiate**	Instantiates any given Game Object
Object/**Look At**	Makes the target face a certain object/direction
Object/**Physics**	Manipulate a Rigidbody's property and add Force
Object/**Set Active**	Toggle the state of the given Game Object
Object/**Enable Component**	Toggle the state of the given Game Object Component
Object/**Transform**	Change the Transform's properties of the target
Save & Load/**Current Profile**	Change the current save game's profile (0: default)
Save & Load/**Load Game**	Loads the current profile's game state

Save & Load/**Save Game**	Saves the current game's state
Scene/**Load Scene**	Loads a scene identified by its name
Messages/**Simple Message**	Displays a message with a defined color with an optional voice over

Events

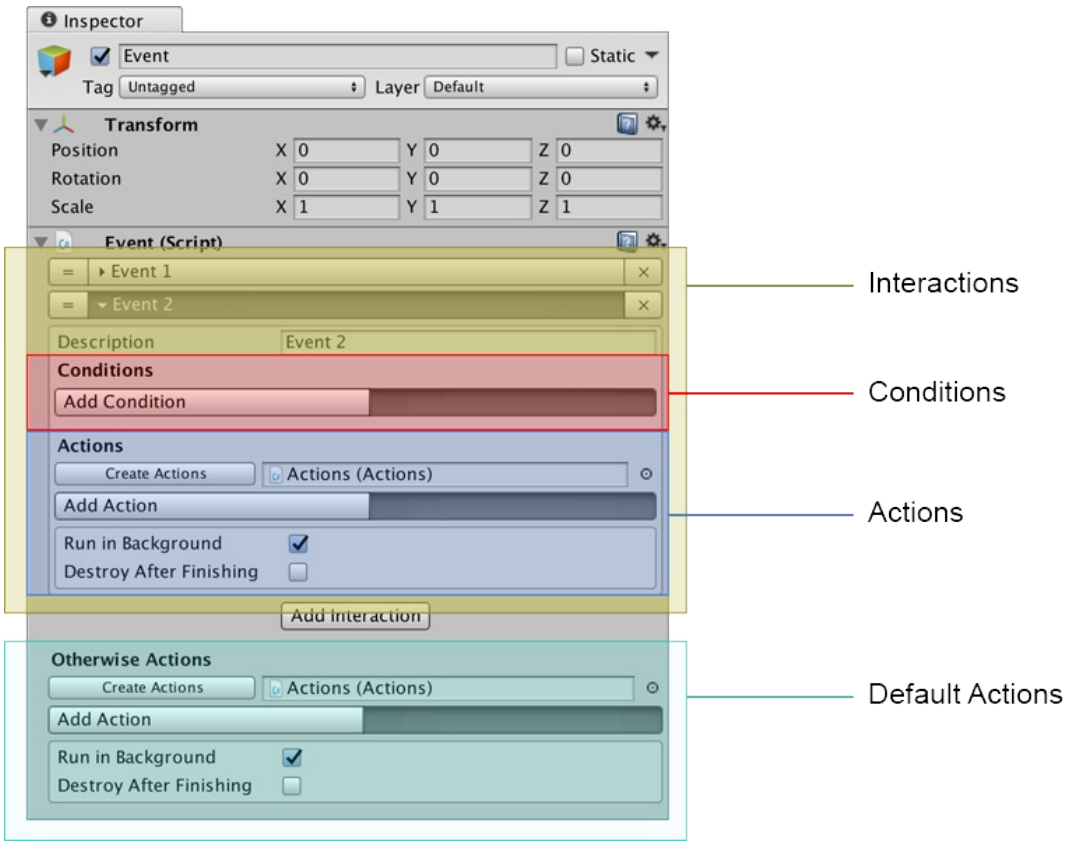
Events are wrappers around **Actions** which allow to condition the execution of these.



They are made of multiple **Interactions**, which contain a collection of **Conditions** and **Actions**. An **Event** has also a *Default Actions* collection that is executed if none of the **Interactions** are satisfied.

Event Anatomy

Events work much like Actions, but instead of immediately being executed when called, they perform a set of *checks* before deciding which action to call (if any at all).



An **Event** follows the following process:

- Check the first **Interaction**
  - If all **Conditions** are *true*
  - Then executes its **Actions**
- If not all **Conditions** where *true*, jump to the second **Interaction**
  - If all **Conditions** are *true*
  - Then executes its **Actions**
- [...]
- If any of the **Interaction**'s **Conditions** was *true*, then execute a special **Actions**

Only one set of **Actions** will be called in an **Event**. If the first **Interaction**'s **Conditions** are all true, only its **Actions** will be executed and the rest of the **Interactions** won't even be checked.

> **Events** are hard to get at the beginning. For a more comprehensive explanation, watch this video tutorial (less than 3 minutes).

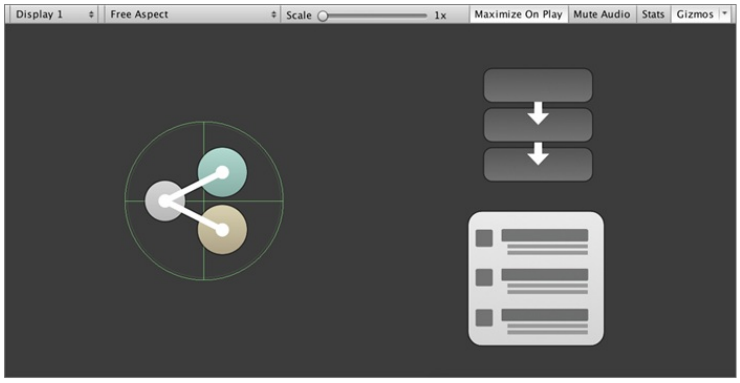
Available Conditions

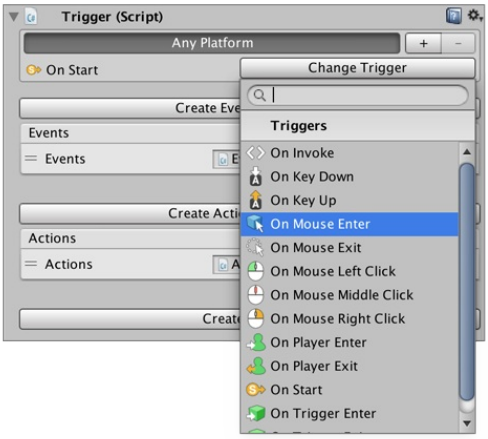
Here's a full list of all **Game Creator**'s available **Conditions**.

NAME	DESCRIPTION
Data/**Variable**	Checks if it is less, equal or greater than a value or another variable
General/**Simple Condition**	For debug purposes. Returns true of checked. False otherwise
Input/**Keyboard**	Checks the state of a certain key of the keyboard
Input/**Mouse**	Checks the state of a certain button of the mouse cursor
Object/**GameObject Active**	Checks if a target Game Object is active or not
Object/**Exists GameObject**	Checks if a target Game Object has been destroyed or not
Character/**Character Property**	Checks if a property of a Character is true or false

Triggers

Triggers are used to fire **Actions** and **Events** when *something* happens. This "*something*" can range from the player pressing a certain key to an NPC's entering a certain area.





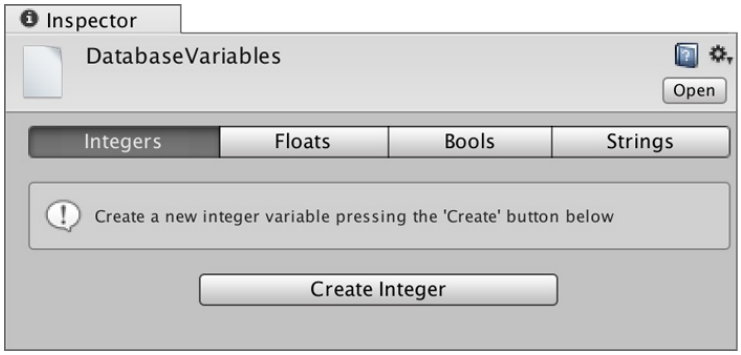
> A **Trigger** can fire multiple **Actions** and/or **Events** at the same time. The execution order is always from top to bottom.

Here's a list of the *conditions* in which a **Trigger** starts calling **Events** and **Actions**:

NAME	DESCRIPTION
On Hover Hold Key	True if the user releases a key after X seconds when hovering the Trigger
On Hover Press Key	True if the user presses the key when hovering with the pointer the Trigger
On Invoke	The Trigger can only be called from the method `Execute()`
On Key Down	Fires when the user starts pressing the key
On Key Hold	Fires when the user releases a key that has been pressed for X seconds
On Key Up	Fires when the user releases a key
On Mouse Enter	Checks if the mouse is hovering the Trigger
On Mouse Exit	Checks if the mouse is exiting the Trigger
On Mouse Left Hold	Fires if the left mouse is released after a specific number of seconds
On Mouse Right Hold	Fires if the right mouse is released after a specific number of seconds
On Mouse Middle Hold	Fires if the middle mouse is released after a specific number of seconds
On Mouse Left Click	Checks if the left mouse has been clicked
On Mouse Right Click	Checks if the right mouse has been clicked
On Mouse Middle Click	Checks if the middle mouse has been clicked
On Player Enter	Fires when the Player enters the Trigger
On Player Enter Key	Fires when the Player Enters the Trigger and presses a Key
On Player Exit	Fires when the Player exists the Trigger
On Player Stay Timeout	Fires when the Player enters and stays a specific number of seconds in the area
On Start	Fires as soon as it can
On Trigger Enter	Fires when a Collider enters the Trigger
On Trigger Exit	Fires when a Collider exists the Trigger
On Trigger Stay Timeout	Fires when a Collider enters and stays a specific number of seconds in the area

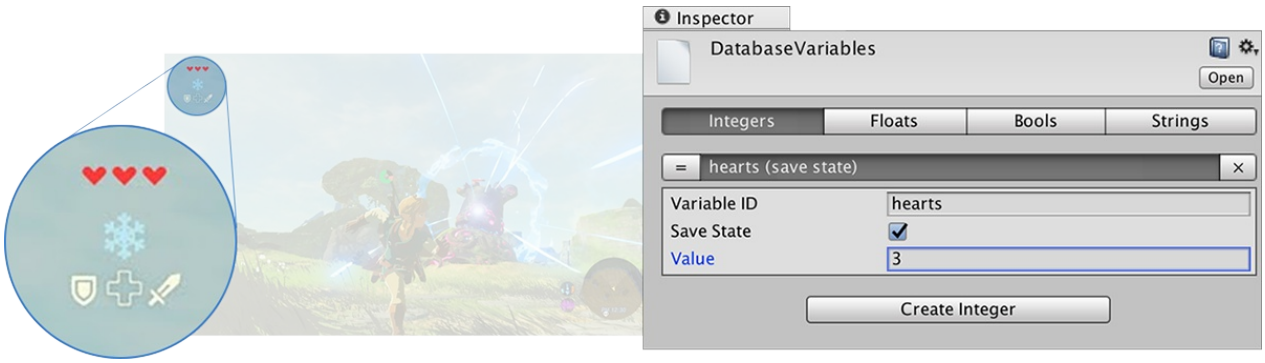
Variables

Game Creator comes with a variables system that allows you to keep track of the game progress and save the state between play sessions.



A **variable** is a value container with information that can be changed during gameplay.

> For example, in a **Zelda** game, the hearts at the top of the screen would be a variable identified by the name "*hearts*" or "*life*"



A **variable** is has a *name* that identifies it (for example: “shotgun ammo“) and a *value*. This value can be an **integer**, a **floating point value**, a **string** or a **boolean** (*true*/*false*).

Variable Anatomy

Variables are composed of three fields:

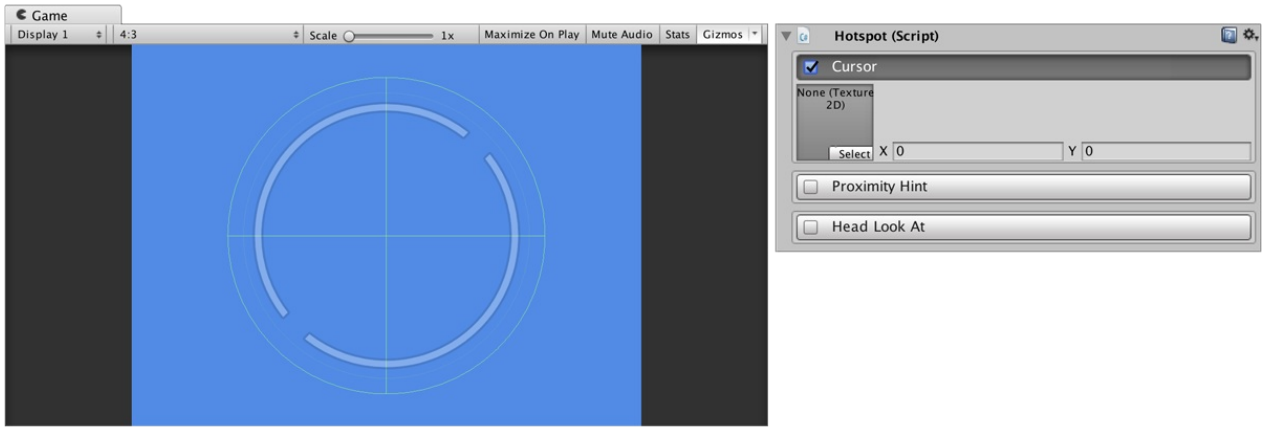
- **variableID**: a unique name that identifies this variable. Try to be descriptive.
- **saveState**: if checked, when the action `Save Game` is executed, its value will be stored.
- **value**: its default value. The type changes depending on the variable type.

To restore the values of a previous play session call the action `Load Game`

> If two variables have the same same **variableID** an error message will prompt during the game in the *Editor*

Hotspots

Hotspots are interactive elements that provide a simple interface for common gameplay mechanics that are not part of the core of the game. For example, a typical hotspot is located complementing a **Trigger** with the option to change the cursor texture when the mouse hovers the object.



> A **Hotspot** has different options that can be enabled/disabled individually.

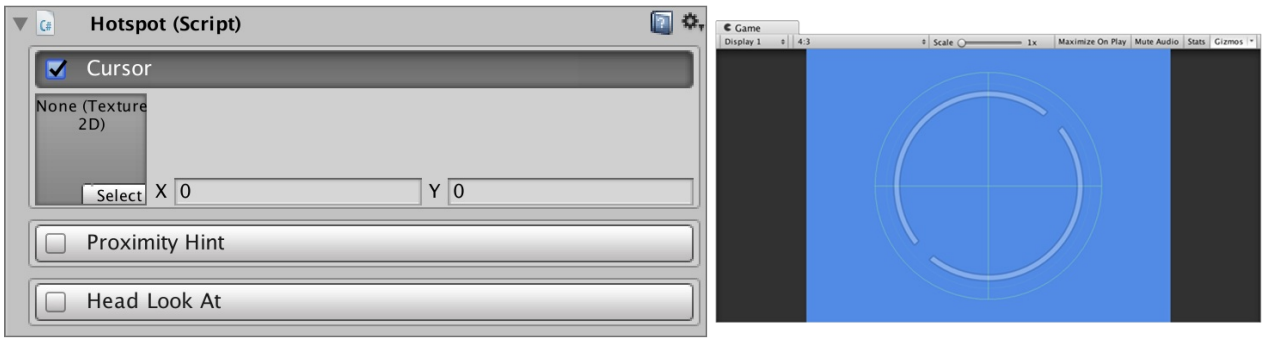
Creating a Hotspot

To create a **Hotspot** element simply *right click* on the *Hierarchy Panel* and select `Game Creator → Other → Hotspot`. You can also create a **Hotspot** element adding the component to any game object.

Hotspot options

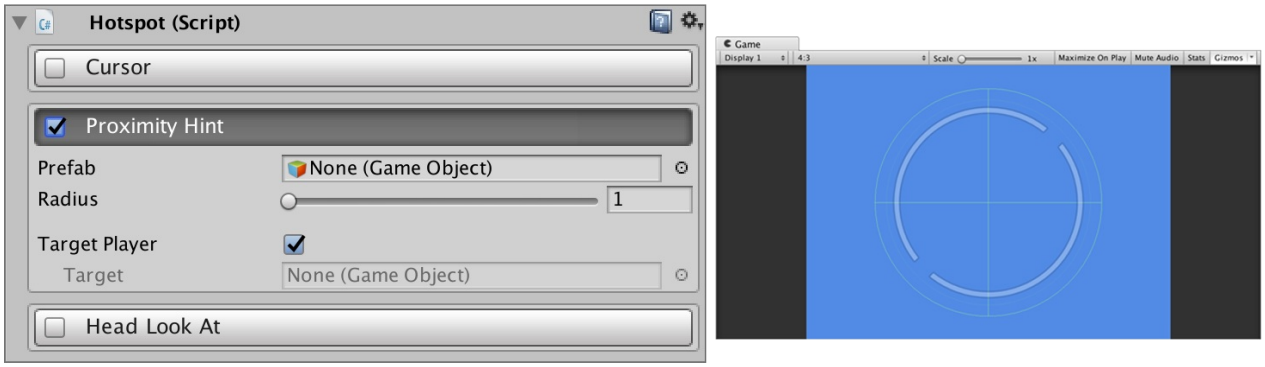
A **Hotspot** component has different options that can be enabled.

Cursor option



Whenever the mouse cursor hovers the **Hotspot**'s collider, the cursor texture will switch to the specified. You can also tell Unity the position of the click point inside the texture (in pixel units).

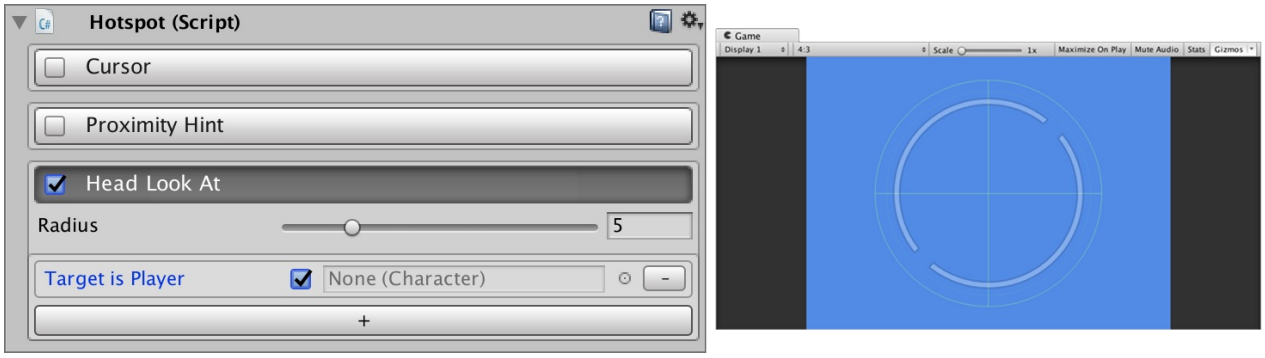
Proximity Hint option



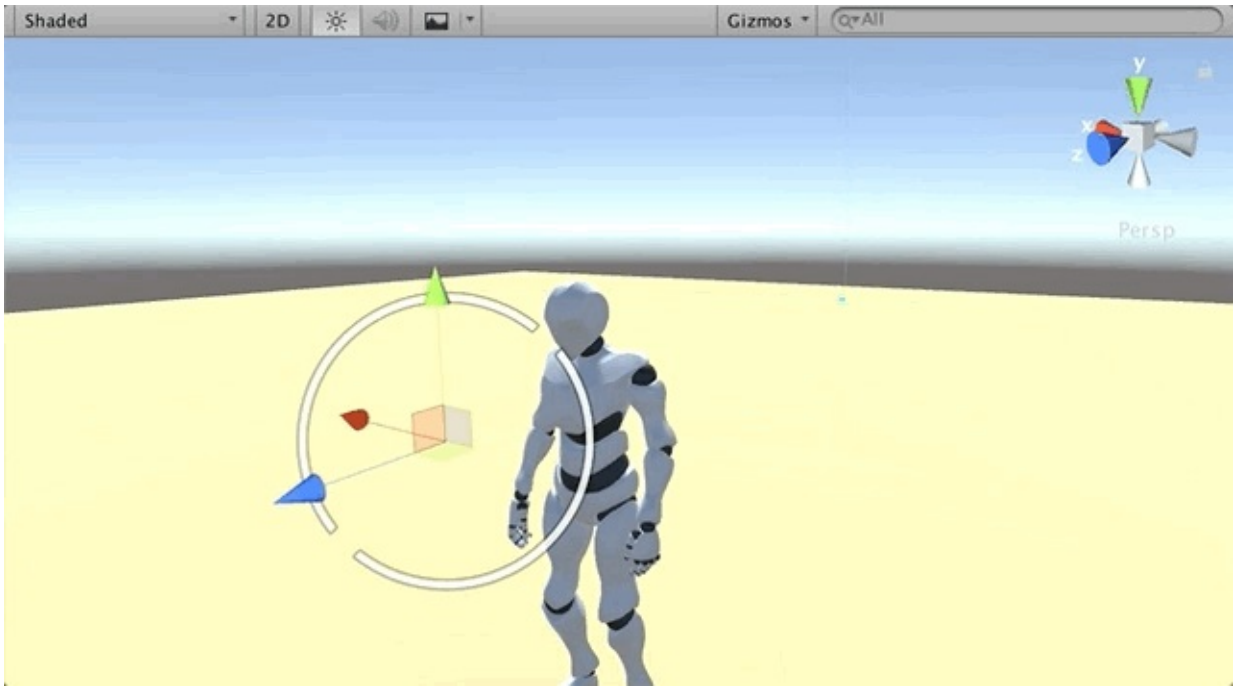
Sometimes you want to hint the user (with a glow for example) there's an important object when the player is nearby.

An instance of the **Prefab** game object will be created once and **enabled**/**disabled** whenever the player enters or exits the area specified by the radius.

Head Look At option



If you have a *humanoid* character as a player you can make him **look at** the **Hotspot**. This is a subtle effect but a great way to indicate important objects without the need of using UI elements.

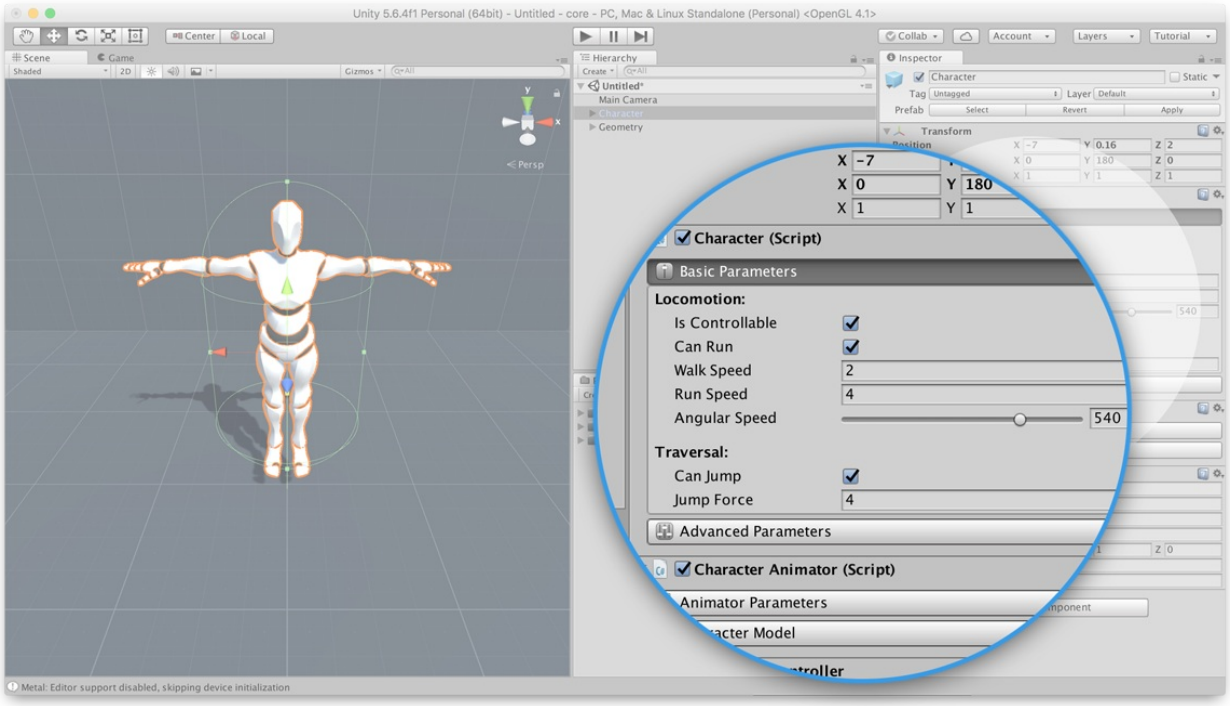


Characters

The **Characters** component allows to define an object as a character. A **Character** can be moved around using the appropriate **Actions**. You can also tweak different properties such as if the character can *walk*, *run*, its *speed* and more

Since version *0.2.5* a character can also perform one of multiple **gestures**, such as waving its hand, picking an object, drinking a potion, ... And be in one of multiple states, such as crouched, injured and so on.

Since version *0.2.6* a character can also **Jump** obstacles.



> This component is very useful if you want to kickstart your game in a few seconds.

Character Properties

The **Character**'s properties is divided into two big sub-groups: The **Basic** properties, and the **Advanced** properties.

Basic Properties

As its name implies, the **Basic** character properties are easily understandable and directly affect the **Character**'s behavior.

- **Is Controllable:** An on/off property that allows to set a character can be controlled with **Actions**.
- **Can Run:** A toggle property that restricts the speed of the character. If set to true, the maximum speed the character can move is `runSpeed`. Otherwise the `walkSpeed` will be used.
- **Walk Speed** and **Run Speed:** The lineal speed the character moves when walking/running.
- **Angular Speed:** The speed at which the character rotates towards its desired direction expressed in **degrees per second**. If this property is set to 180, a character will need one second to turn to its opposite direction.
- **Can Jump:** Allows to give the character the ability to jump. If set to false, even if the Character receive an order from an **Action** to jump, it won't do it.
- **Jump Force:** Default jump force used by the *Player*'s jump input and the **Jump Action**

Advanced Properties

The **Advanced** character properties are meant to be modified by more advanced users who need finer-grain control over the *Player* and other *NPCs*.

- **Slope Speed Up** and **Slope Speed Down:** In real life, a human doesn't move at the same speed when running on a flat surface as it does on a steep one. This parameter allows to define the amount (percent) of *speed* that is subtracted from going up or down.
- **Face Camera Direction:** If checked, the character will always face where the camera's direction. Very useful when making a game from a First Person perspective.
- **Can Use Navigation Mesh:** If checked, the character will automatically use the *Unity's Navigation Mesh* system when moving from one point to another and change back to using the *Character Controller* if not moving. It is recommended to use this, as it allows to avoid obstacles when moving characters around the scene.

> **Can Use Navigation Mesh** is disabled by default because the user has to **Bake** the Navigation Mesh first. To know more about using the **Navigation Mesh** follow this link.

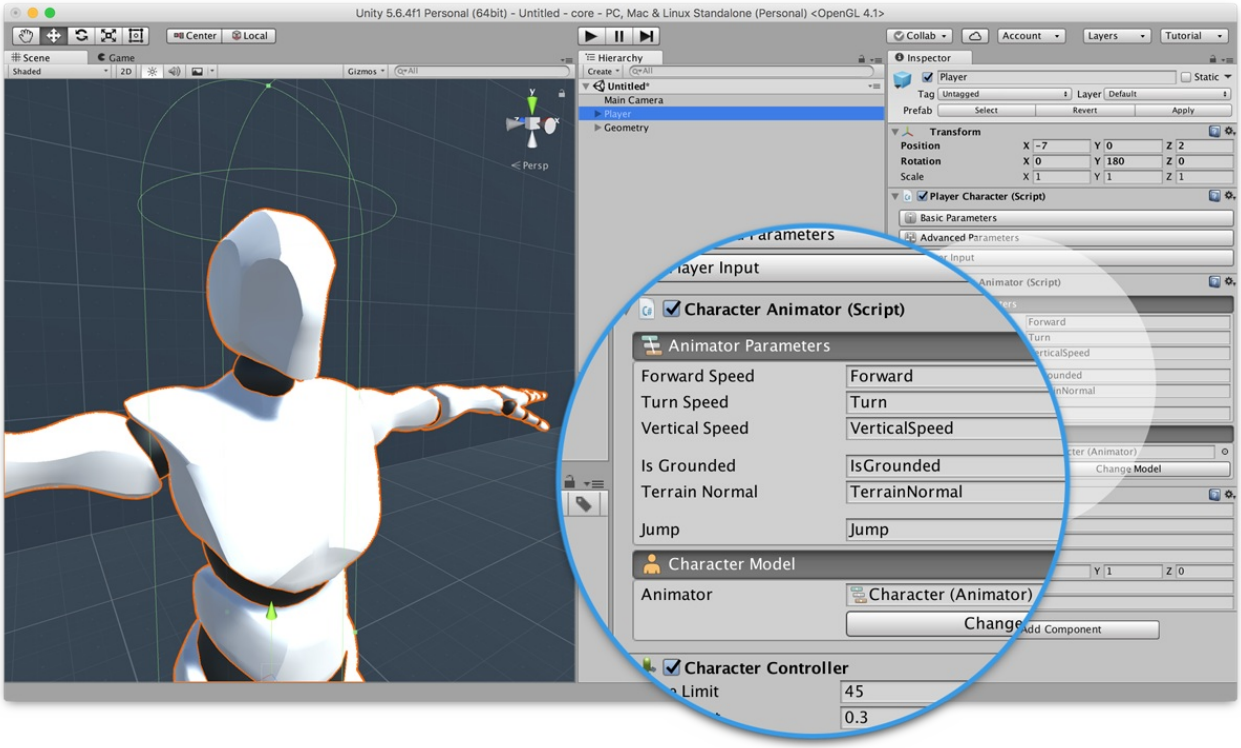
Inverse Kinematics

Since version 0.4.1 **Game Creator** allows the use of **Inverse Kinematics** (aka **IK**). This advanced technique allows to correctly place the feet of a character taking into account the steepness of the terrain, instead of relying on the animation of the character.

Game Creator goes one step further and has a custom feature called **Weight Compensation**, which allows the character to slightly elevate or crouch depending on the inclination of the floor, so not only the feet are properly aligned, but also the knees gracefully bend.

Animating the Character

In order to make the system more flexible, the **Character** and its animation system have been split in two different components.



Likewise, the **Character Animator** has two different sections: The **Animator Parameters** and the **Character Model**.

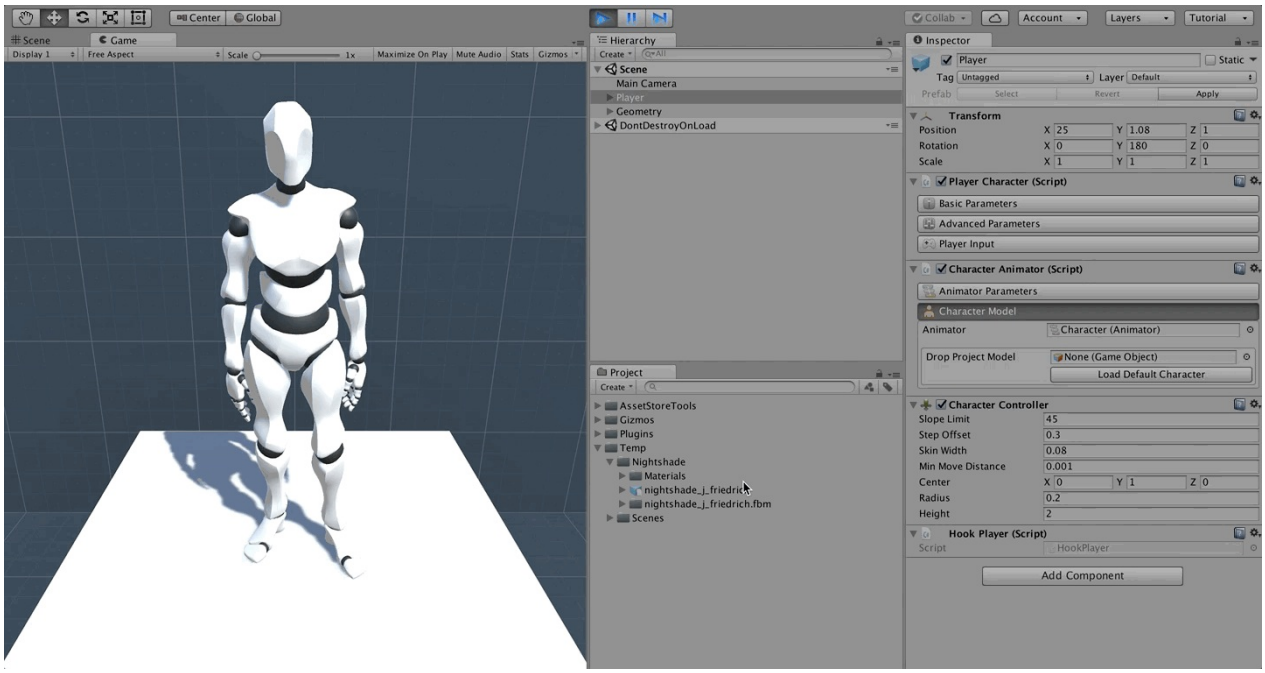
Animator Parameters

These string values behave as a proxy between the **Character** component and the **Animator**, gathering locomotion information from the logic controller and plugging it into the **Animator**. The string values defined inside the **Animator Parameters** allow to have a custom **Animator Controller**.

> We don't recommend tweaking these parameters unless you are certain of what you want to do. In case you want to use your own Animation solution, you can access the current **Character** state by calling a public method named `GetCharacterState`. It returns a `Character.State` class with information about if the character is grounded or not, its direction, etc.

Character Model

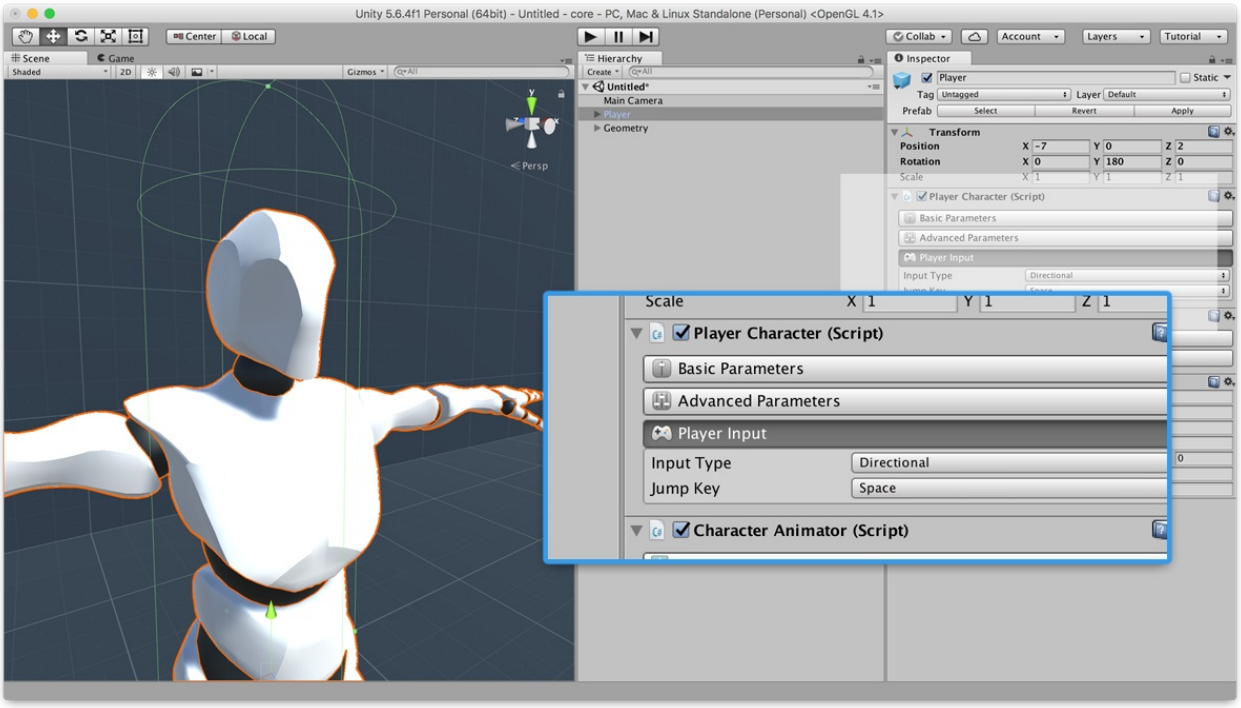
You'll probably want to make a game with your custom character models. Luckely, changing between characters in **Game Creator** is as easy as clicking the **Change Model** button and dragging in the 3D model you want to use from your *Project Panel*. **Game Creator** will take it from here and automagically update the character with the new one.



> You can also change characters in **Playmode** in case you want to see how they look like. Switching back to **Editmode** will undo any changes made.

Player Character

The **Character Player** inherits all the functionalities from the **Character** component. This section assumes you are familiar with the **Character** component. If you aren't, head to read it and carefully read it before proceeding.



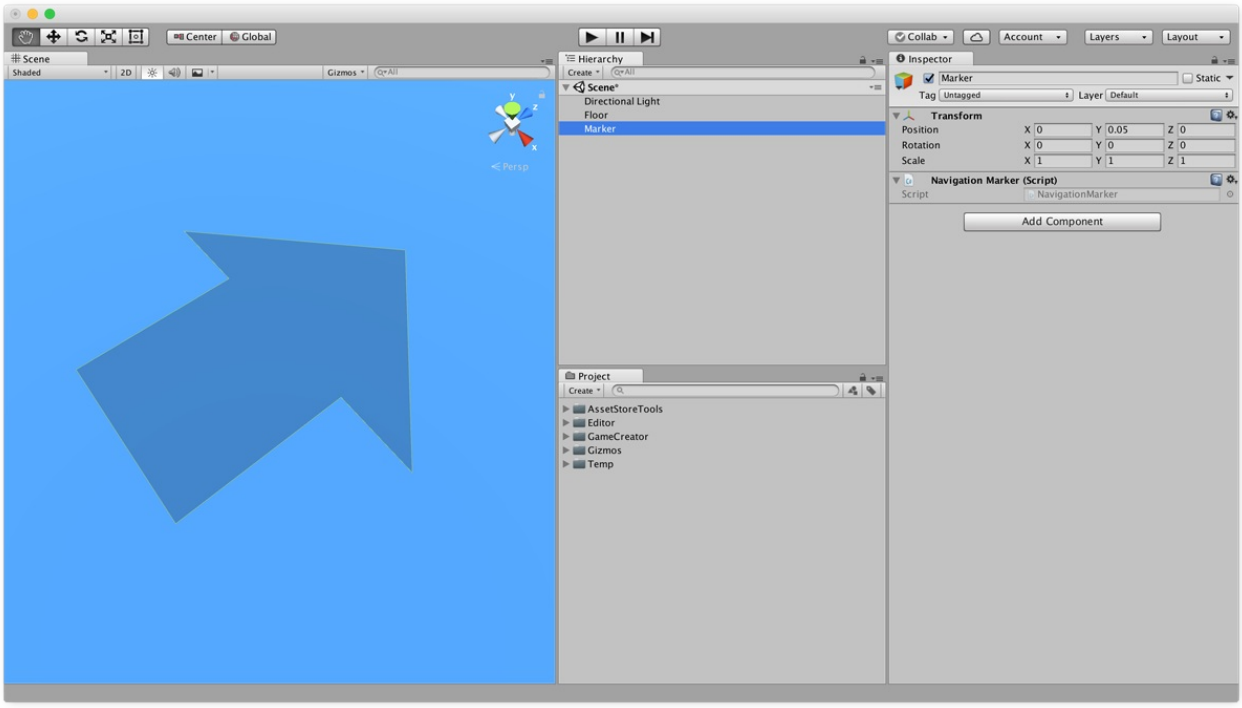
The player is just a **Character** with a custom **section** called **Input** that allows to define which user's keys/button/joystick inputs do what. You can choose to either move the player using the *mouse* (selecting **Point & Click**) or using the keyboard/joystick (selecting **Directional**).

> If the **IsControllable** property is set to false, the player won't respond to the user's input.

You can also define which key will make the **Player** jump, though bear in mind that if the Player has the **Can Jump** checkbox set to false, it won't jump.

Markers

Markers are special game objects used to set as destinations for moving characters. These **Markers** that are represented with a big yellow arrow.



To create a **Marker** go to the *Hierarchy Panel* and select `GameCreator → Navigation → Marker`. Drag it to the place you want and rotate it.

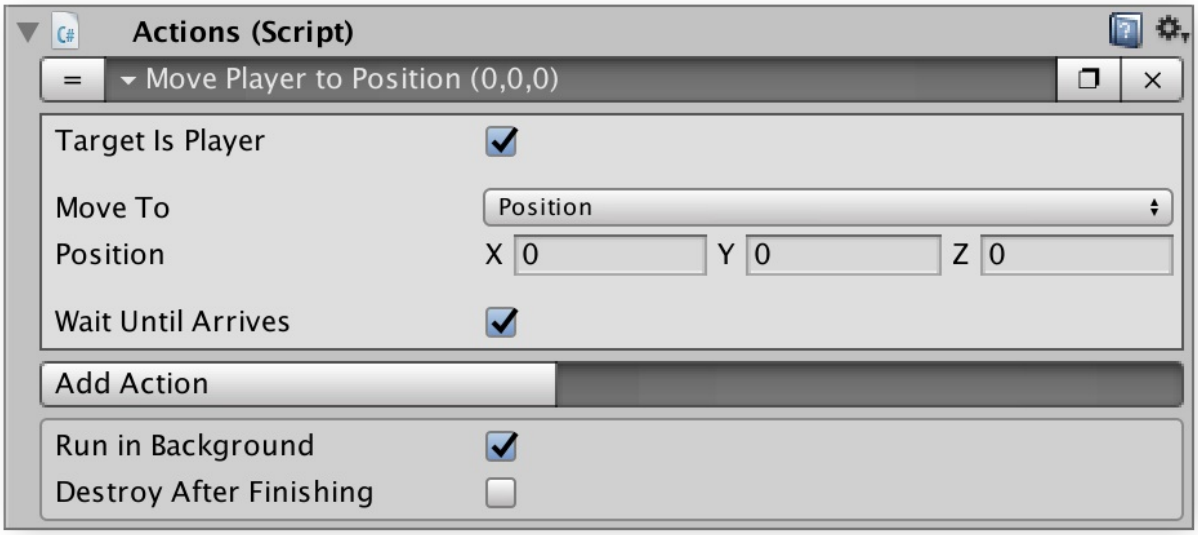
> Setting a **Marker** as a destination will cause the character not only to move towards it but also face the direction the **Marker** is looking at.
>
> Since version 0.4.1 you can also change the color of a specific **Trigger** arrow. By doing so, you can create a color-code to visually identify which one does what.

Character Actions

The **Character** component provides a way to control how the player and the rest of the characters behave.

Move Character Action

The main **Action** is called `Move Character` and as it name implies, it allows to command a character to move to a certain position.



There are quite a lot of properties to play with this **Action** so let's break them down.

Target

The target is the character that is going to move. You can either choose to automatically select the player (in case there is a `Player Navigation Controller` in the scene) or manually add a game object with a `Character Navigation Controller`.

Move to

- **Position**: Specify a 3D position in *world space*
- **Transform**: The destination is another object's position
- **Marker**: The destination is a special object called *Marker*

> **Tip!** Use a *Marker* instead of an empty object if you want the character to go to a position and **face the same direction as the Marker**

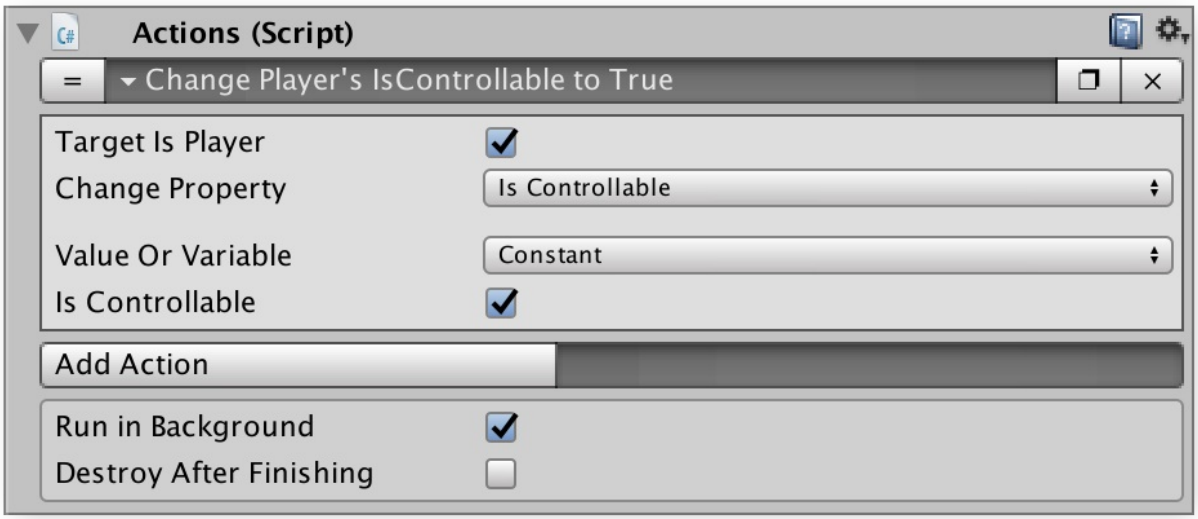
Wait Until Arrives

Check this checkbox if you want to halt the execution of the **Actions** and wait for the character to reach its destination.

> **Alert!** Use the `wait Until Arrives` checkbox with caution. If a character can't reach its destination the list of **Actions** won't resume.

Change Properties Action

Apart from instructing a character to move to a certain position you can also modify how it moves.



The properties to modify are:

Is Controllable

Change whether the character can be controlled by other actions.

> Setting the player to **not controllable** at the beginning of an **Actions** sequence can be useful when starting a cinematic sequence

Can Run

Set whether the character uses the **run** or the **walk** speed when moving

Set Walk Speed

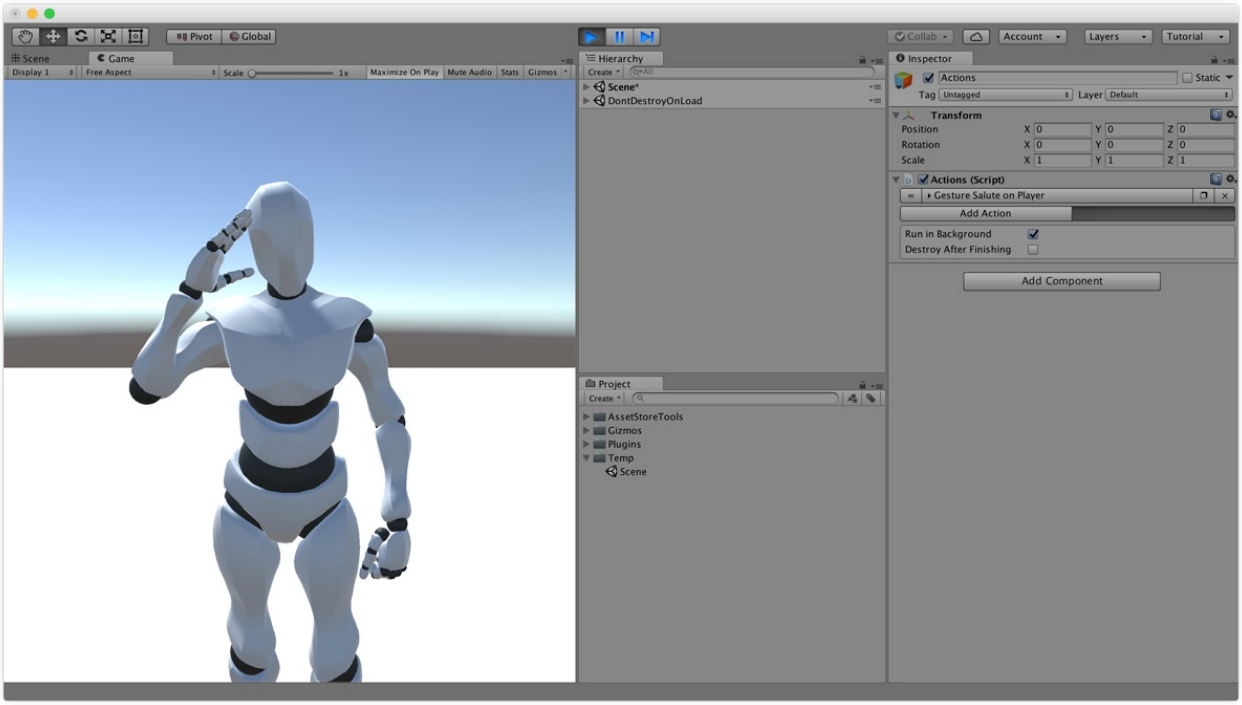
Sets the walking speed. Can be set with a **variable**'s value.

Set Run Speed

Sets the running speed. Can be set with a **variable**'s value.

Character Gestures

The **Character** with the **Character Animator** can perform multiple gestures. A **Gesture** is a small movement the character does that doesn't imply changing its state. For example, when two people interact, their hands move according to their verbal language to emphasize their communication.



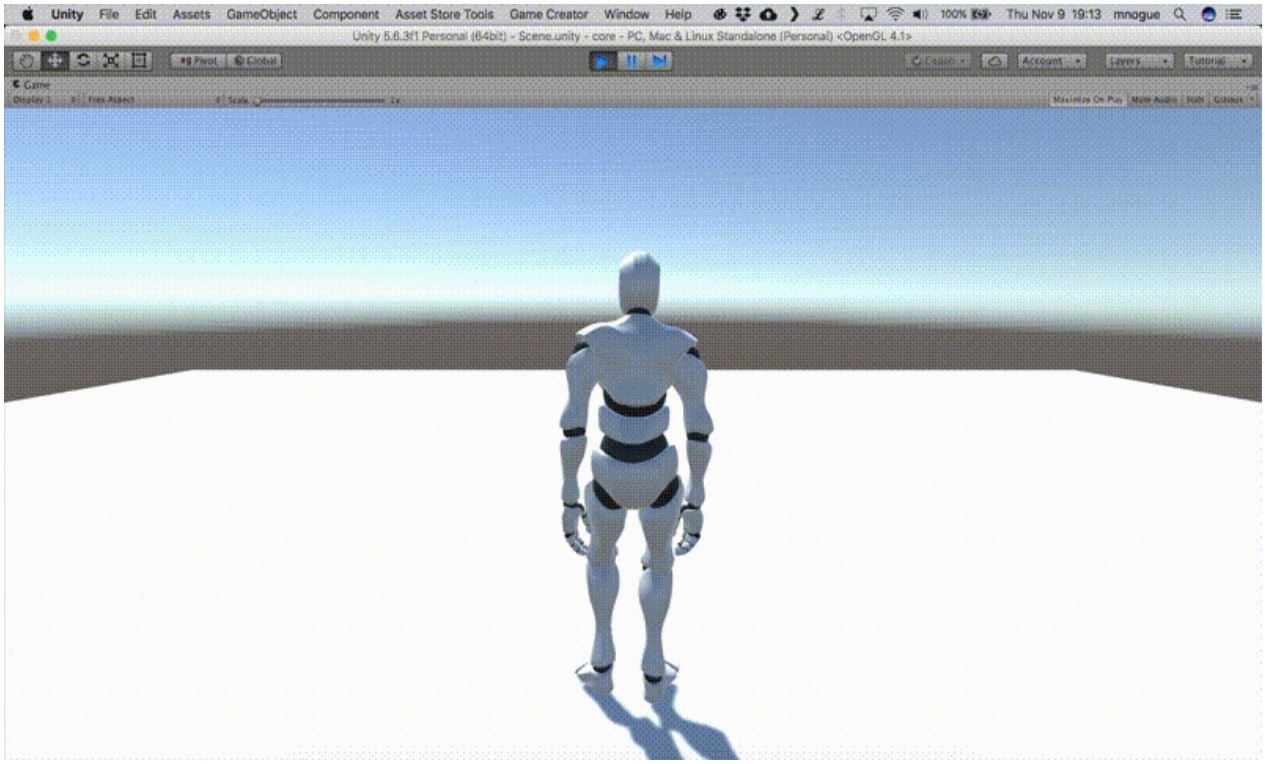
> Since version 0.4.1 you can add custom **Gestures** using the new animation system.

To add a custom animation, simply use the **Character Gesture Action** and drag your animation to the `gesture` field.

Character States

Apart from performing different **Gestures**, a **Character** can also be in different **States**. The difference between a **State** and a **Gesture** is that a **State** is part of a range of Locomotion animations.

For example, you can make the **Player** enter the **Crouch** state when detecting if guards are nearby so sneak past behind them, or move around looking **Drunk**.



> States are not **boolean** (on/off) values. You can blend between the root locomotion and a state setting the *amount* value between **0.0** and **1.0**. This is very useful if you want, for instance, to set the player's normal-injured state based on its remaining health.

Default States

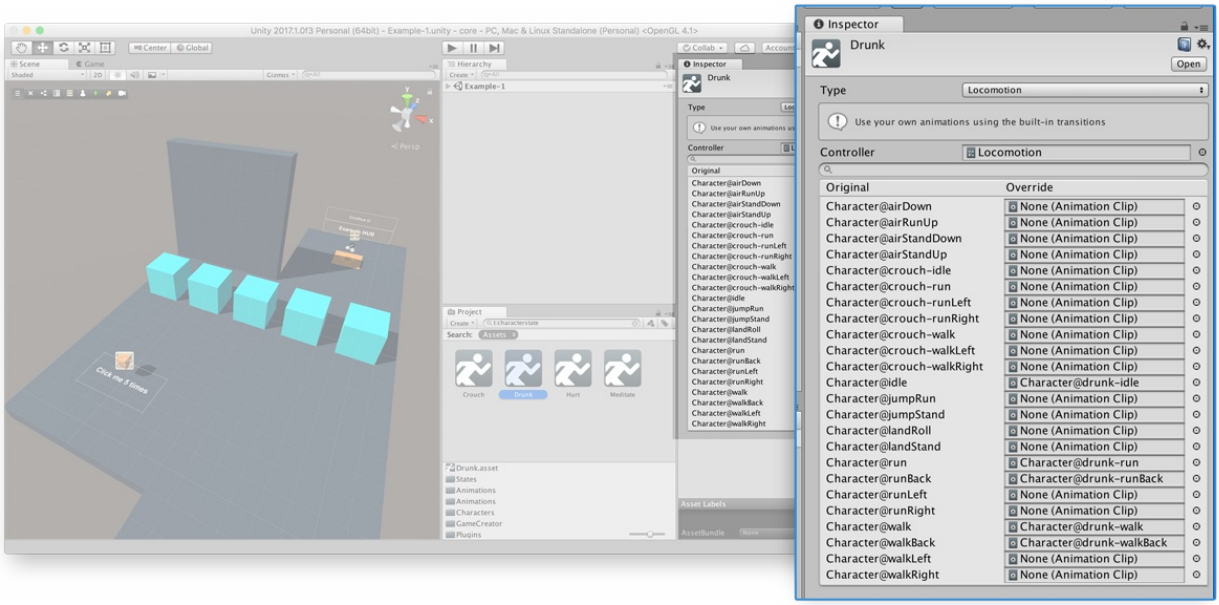
Game Creator comes with some default **States**:

- **Crouch:** The character crouches in a stealthy way
- **Drunk:** The character appears... Not sober ^^'
- **Hurt:** The character moves hoping with just one leg
- **Mediate:** The character sits crossing its legs and placing its hands onto his knees.

Custom States

Game Creator allows you to easily create custom **States**. To do so, right click on the *Project Panel* and select `Create → Game Creator → Characters` and a **State** asset will be created. You'll see there are different types of **States**.

- **Simple State:** Just like the *Meditate* state, a simple state is a character staying in a single pose. This can be sitting on a chair, leaning on a wall, ...
- **Locomotion State:** Like the **Drunk** state, the Locomotion state is a state where you can override the different movement animations of the character. You'll be presented with a list of animation fields where you can drag and drop custom ones to. If you leave any animation in blank, the default one will be used instead.
- **Advanced State:** This is for advanced users who want fine-grain solutions or have a custom **Mecanim Animator Controller**. Drag and drop the animator and call the different parameters need (if any) from a custom script to use it.

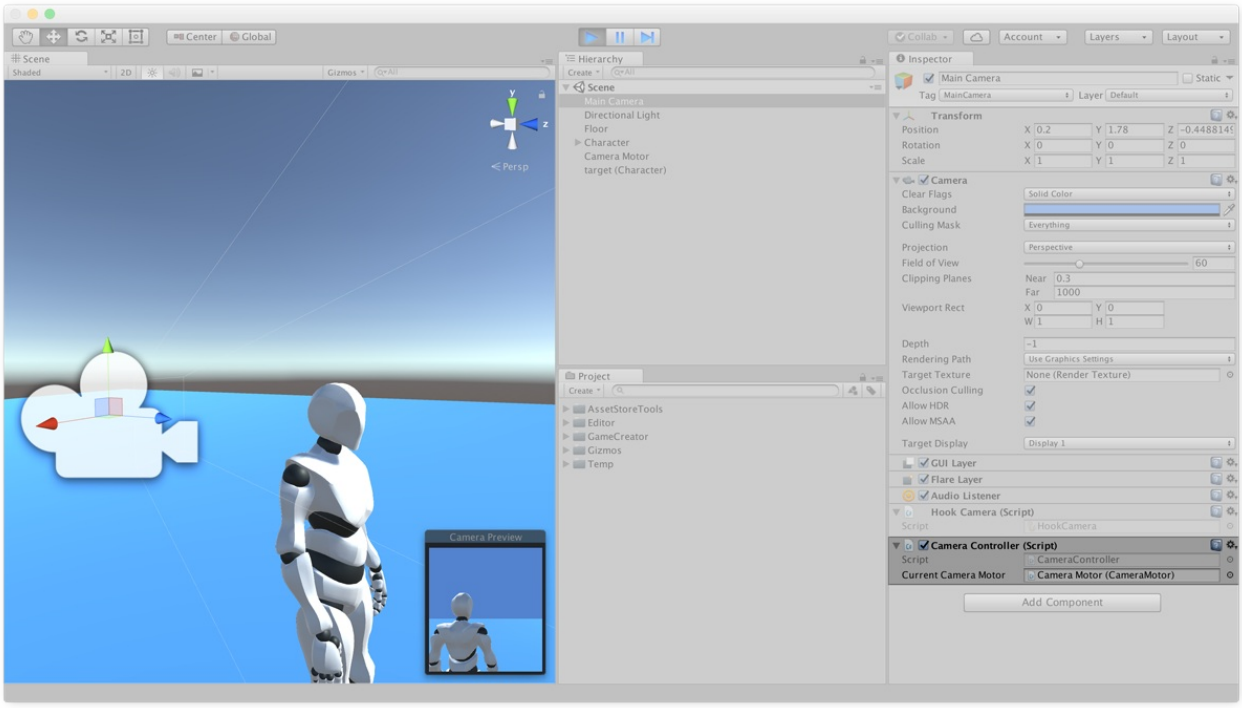


> In the image above you can see an example of a **Locomotion State** asset with a complete list of all possible animations.

When using a **Character State Action** you'll have to drag and drop this **State** asset to the corresponding field and it will automagically bend between the default animations and the new state.

Camera

In order to know which one is the *main camera* and apply different behaviors to it, **Game Creator** provides a component called **Camera Controller**.



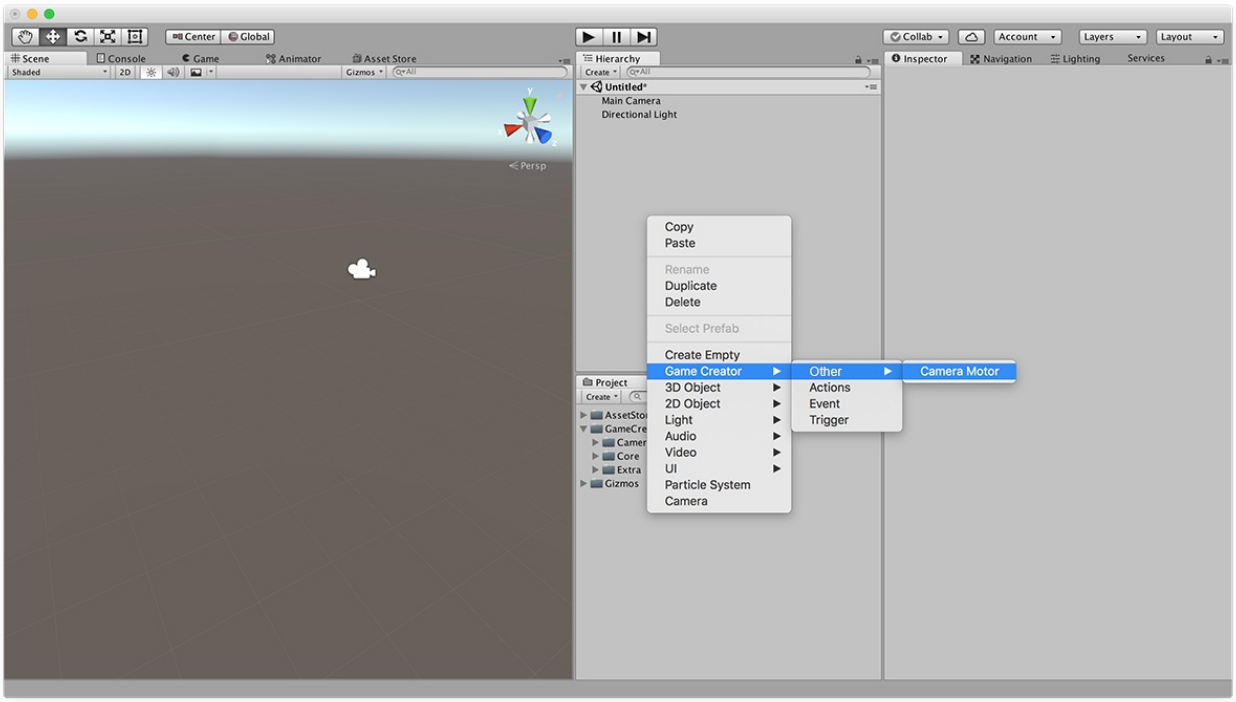
This component is extremely simple as it only has one property field to fill with a *Camera Motor* object. This **Current Camera Motor** field tells the **Camera Controller** which motor is currently affecting it.

> See the next section for more information on Camera Motors.

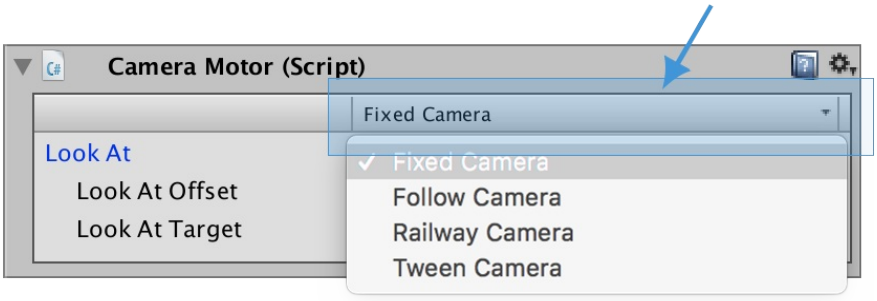
Camera Motors

Camera Motors are objects that tell the *main camera* how to behave. Only one motor can be active at a time.

To create a **Camera Motor** right-click on the *Hierachy Panel* and select `GameCreator → Other → Camera Motor`.



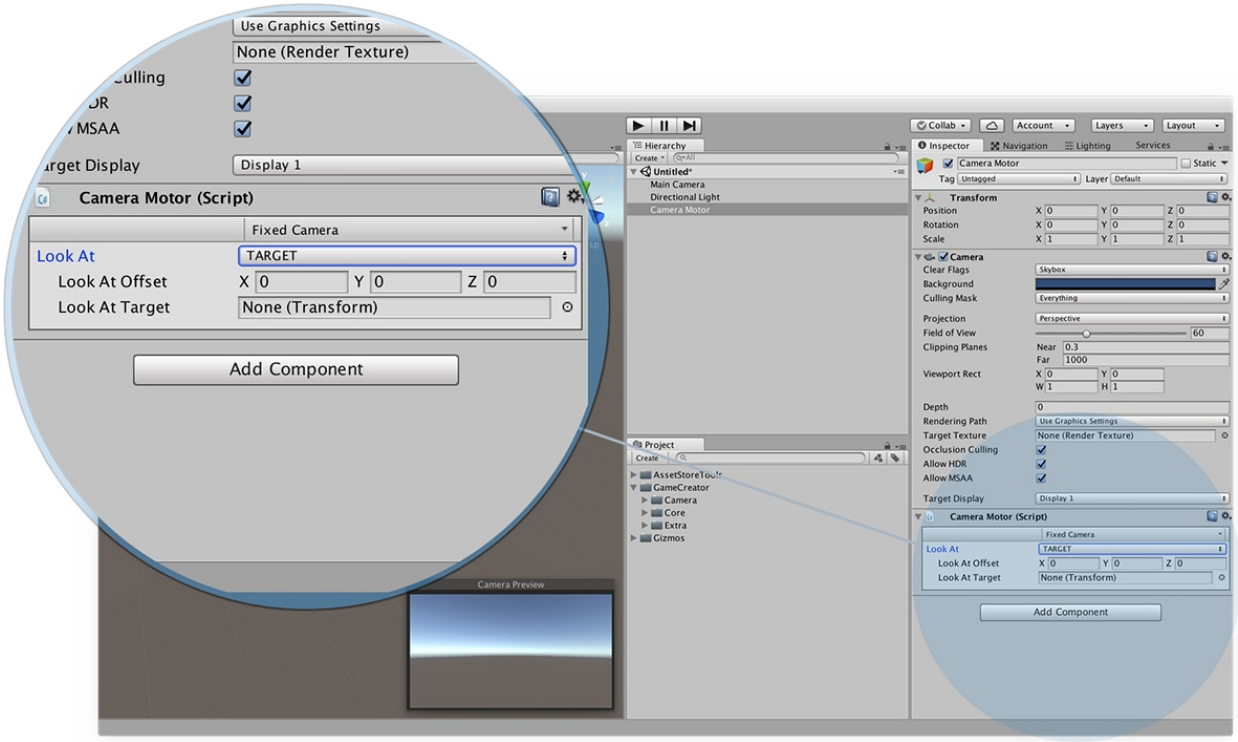
There are different types of motors and to change between them simply click on the drop-down menu of the box that surrounds the motor.



Fixed Camera Motor

This is the simplest type of **Camera Motor**. It stays at a fixed position, just like a *surveillance security camera* and it can rotate to look at different types of targets.

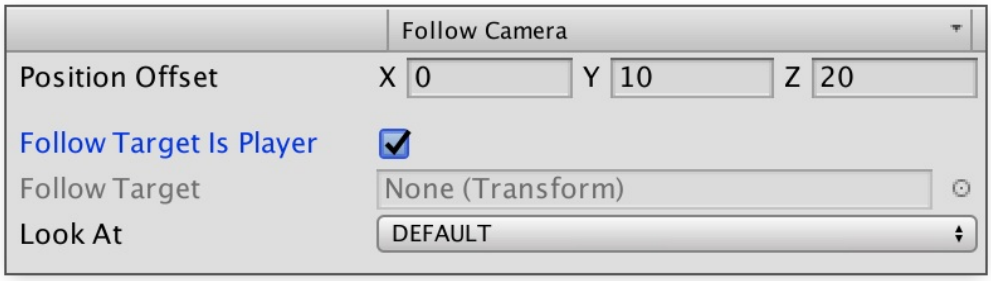
- **Default**: Doesn't rotate
- **Player**: Looks at the player
- **Target**: Looks at a specific target
- **Position**: Looks at a specific world-space position



> This type of camera is used for games like old-school *Resident Evil* or *Point & Click* adventures.

Follow Camera Motor

The **Follow Camera Motor** is similar to the **Fixed Motor** as it allows to look at a specific target/position/player but it also allows to follow it.



> This type of camera is usually used for RTS or top-down games.
>
> There is a little dampening between the position of the target and the actual movement of the camera to prevent it from jittering.

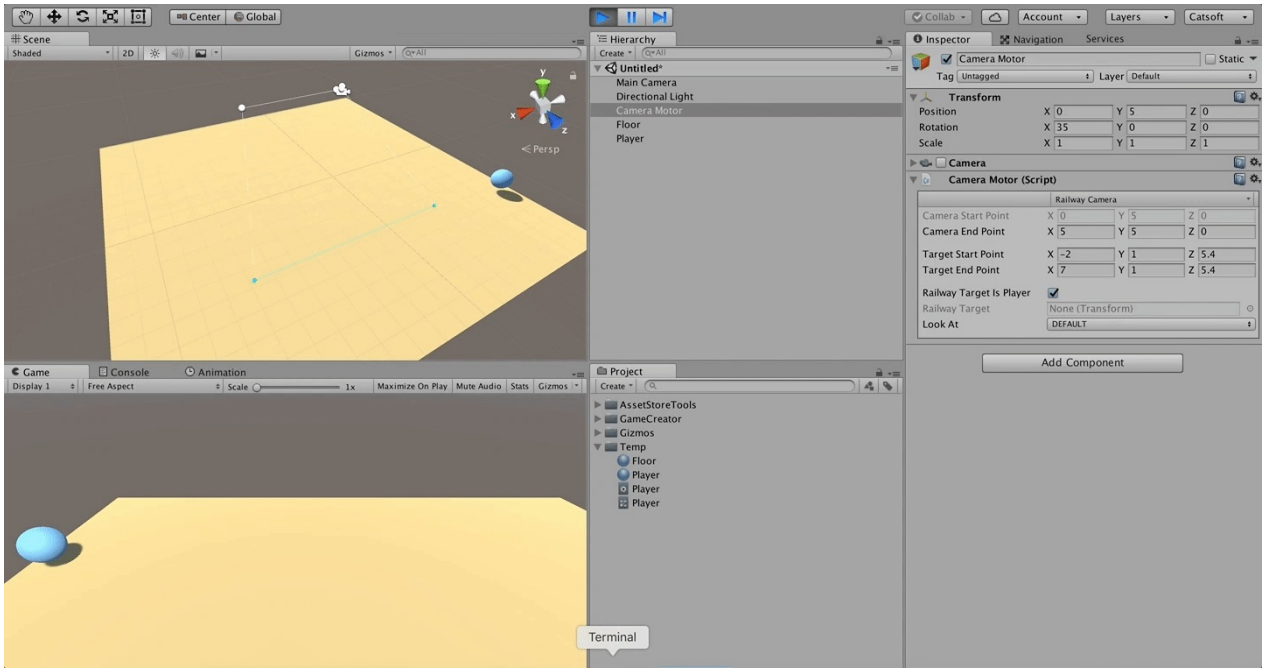
Target Camera Motor

The **Target Camera** behaves much similar like the **Follow Camera** but also accepts a **Look At** target. It's mostly used for *Shoulder* cameras where the camera is anchored to a position (Player's shoulder) but the focus of the camera is targeting another scene object.

> **Tip!** This camera can also be used to highlight an important place

Railway Camera Motor

The **Railway Camera Motor** is a little bit more sophisticated type of motor. It allows the camera to follow the target along a path (or rail) but never exceed it. Here's a small demo:



The player (*blue ball*) moves from right to left. When the player is within the *plane* of the blue line, the camera linearly interpolates the player's position with its min-max path.

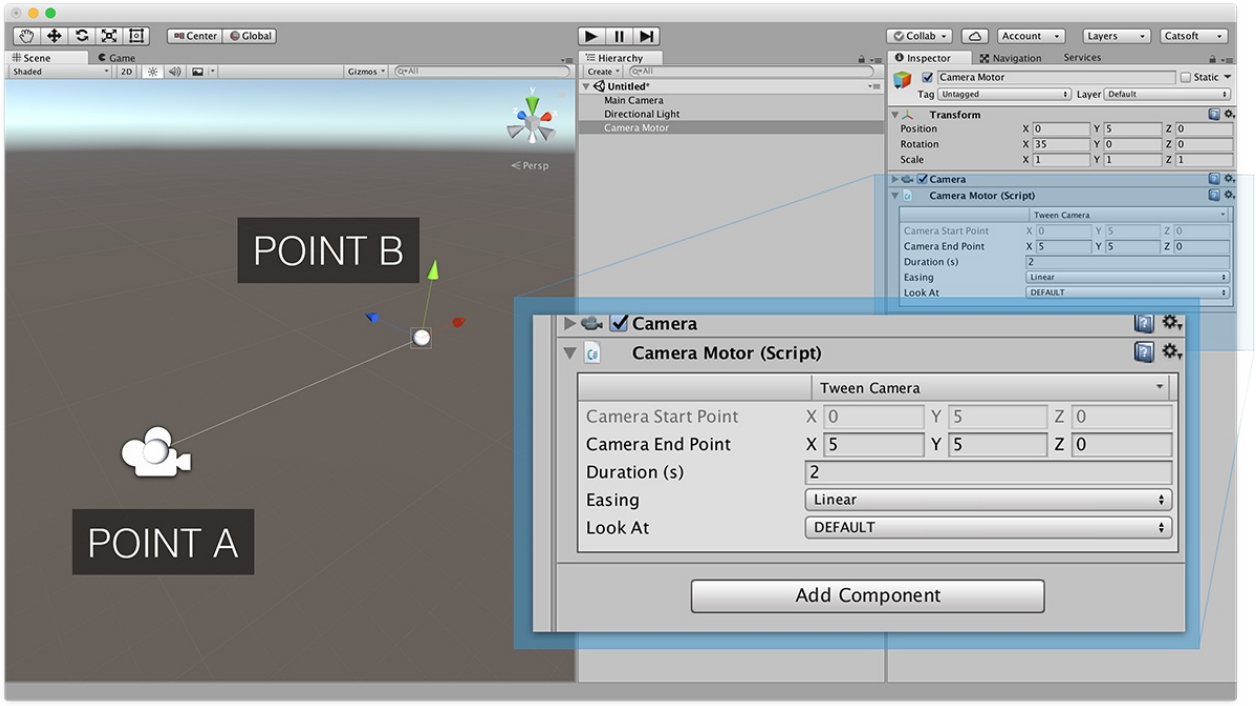
> In other words, the camera follows the player as long as the player is within the blue line. Otherwise it stays at the edge of the path.

This type of cameras are very useful for corridors or narrow rooms.

> **Tip!** You can modify the camera path and the player min-max path by clicking and dragging the white balls in the scene view.

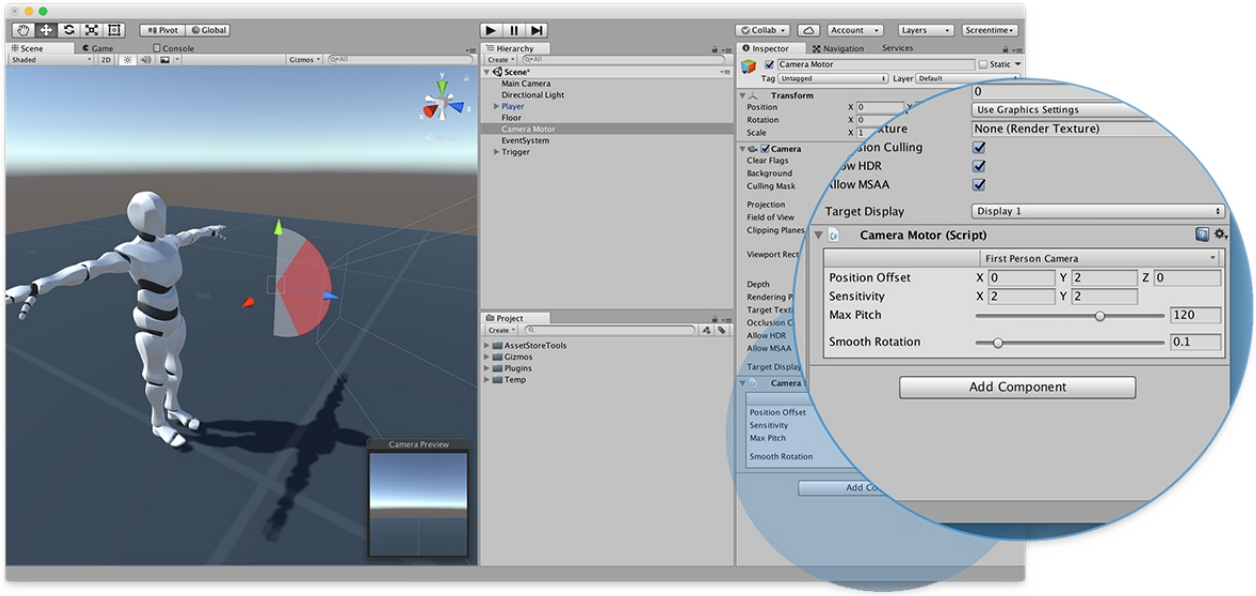
Tween Camera Motor

The **Tween Camera Motor** allows to move the camera from a position *A* to position *B* within a time window. It is mostly used for cutscenes or small animations.



First Person Camera Motor

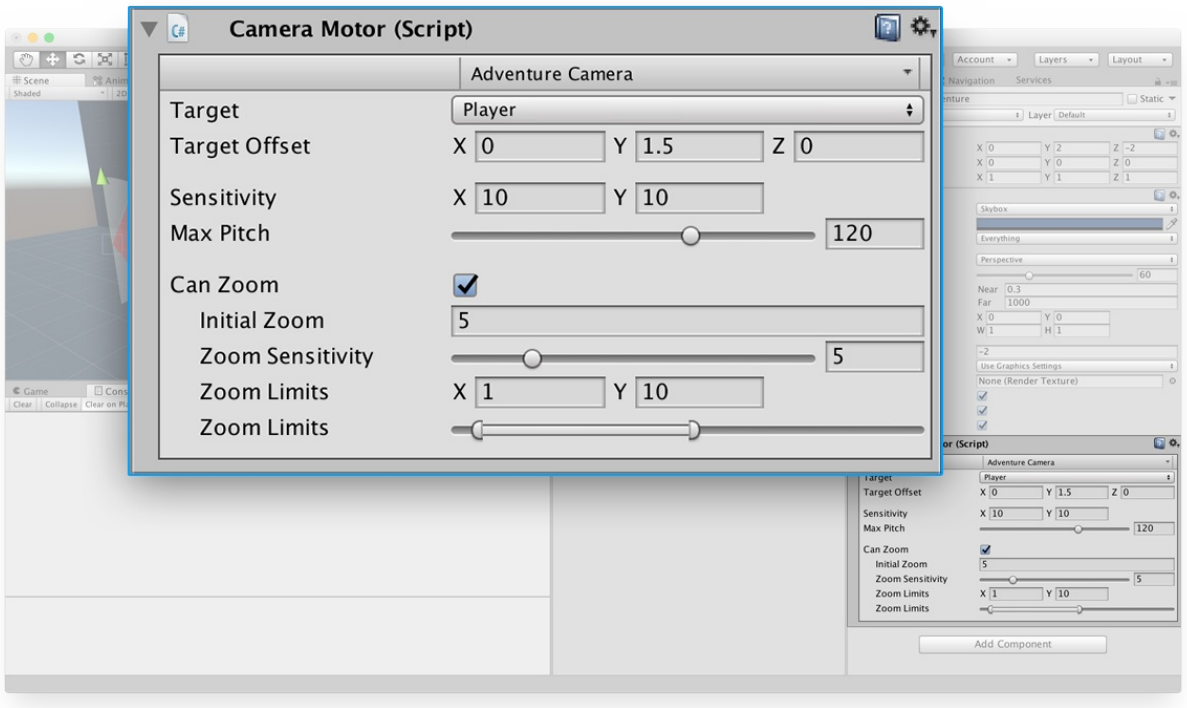
The **First Person Motor** allows to create FPS action games like *Call of Duty* or horror games like *Call of Cthulhu*.



This camera motor allows to define the mouse sensitivity, the amount of dampening/spring the camera moves relative to the mouse and the maximum pitch rotation (in case you want to limit the visual spectrum of the player).

Adventure Camera Motor

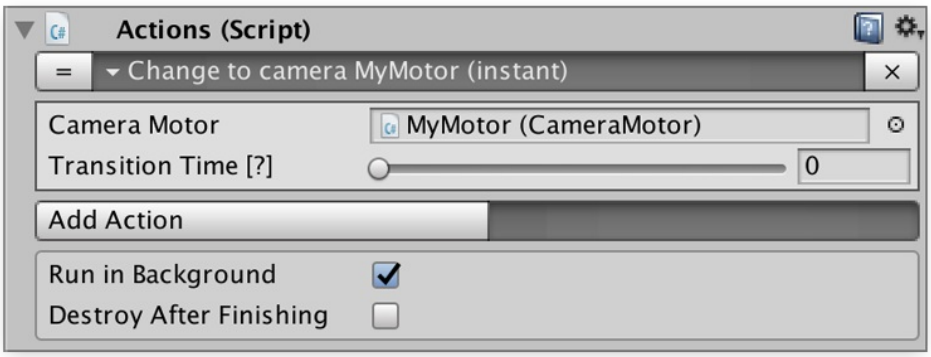
The **Adventure Camera** is an advanced camera system that behaves like the cameras from *Tomb Raider* or *Uncharted* series.



The camera orbits around a target using the mouse movement (on Desktop) or using the touch-screen if the target device is mobile. You can also toggle whether you want to avoid wall collision.

Camera Actions

The **Camera** module provides an **Action** called `Change Camera`. This **Action** allows to switch between the current camera and another one.



There is a `transition time` property that accepts float values. If set to 0 the transition between the current camera and the selected one will be instantanelously (like most of the time you'll want). But you can also smoothly move from the current camera motor to the targeted one.

Changing Camera properties

You can also use the **Camera Damping Action** to change how smoothly the *Main Camera* will follow its target.

> **Tip!** This is quite important for cameras like the **Adventure Camera**, where there shouldn't be any smoothing on the rotation.

Localization

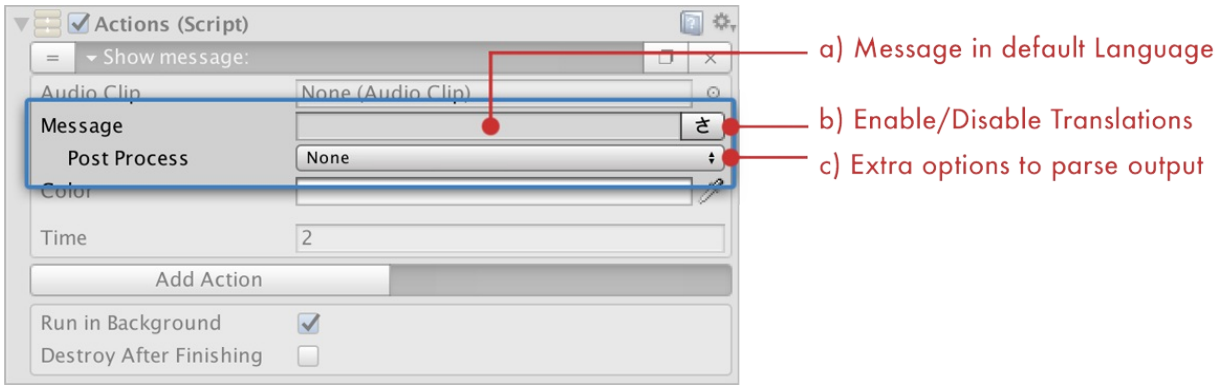
> The **Localization Tools** were first added in version **0.2.4**

Video games are international products and it would be a big mistake to think that shipping a game in only one language (*English*, for example) is enough. Luckily, **Game Creator** comes with built-in **Localization tools** that simplify the process of translating a game into multiple languages.

> We've designed the **Localization tools** to be as simple as possible and non-intrusive in your development workflow. The last thing we wanted to do was to display 5 different input fields, each for a different languages, for each localized message.

Localized Strings

Localized string fields are composed of two parts. The first is the *input field*. Here you type the text in the default language. For example, if we want to show a message saying *Hello, World!* we would simply type these characters there.

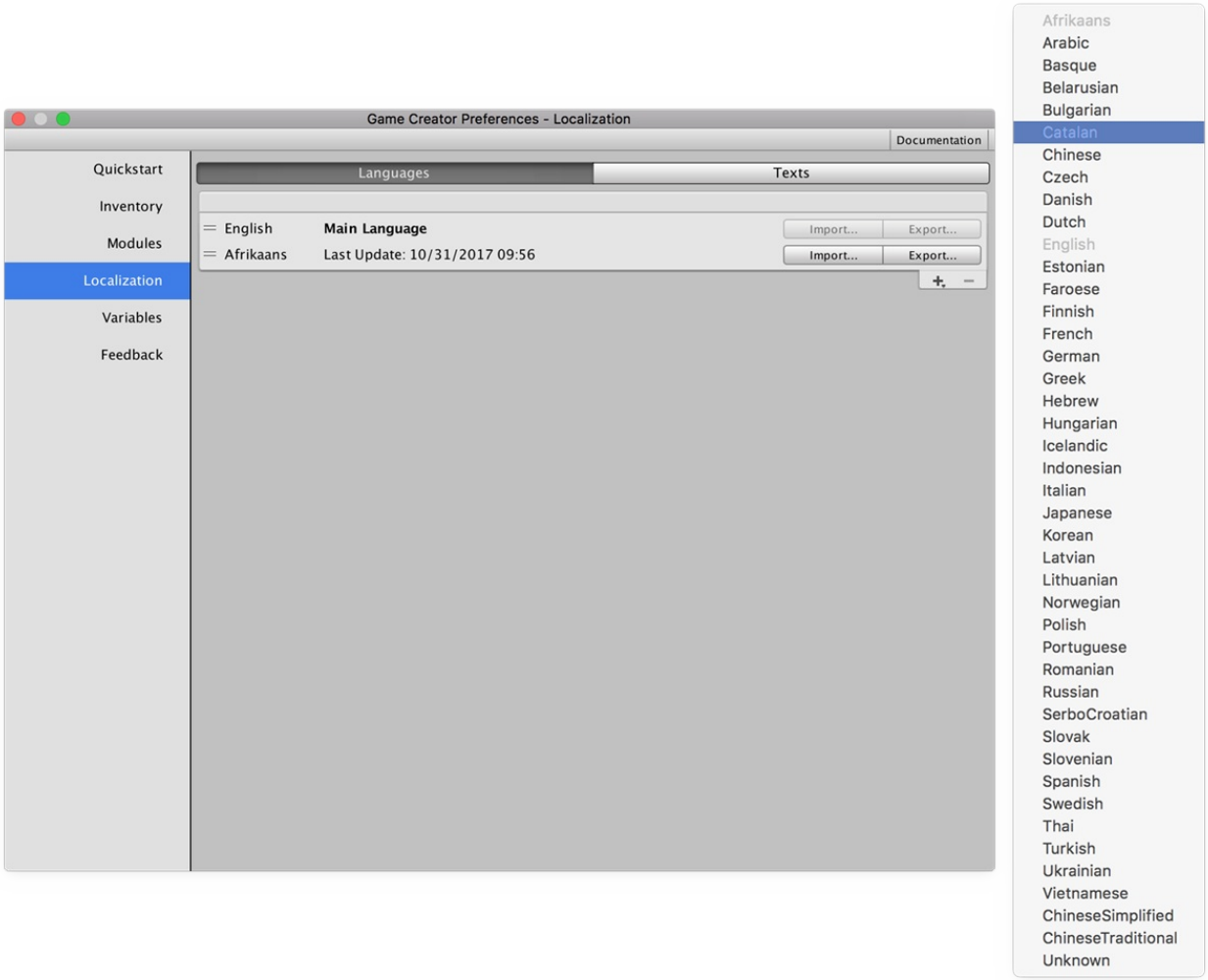


The second part is a button that enables/disables the translation. If you want the previous text to be localized, simply enable the *Kanji* character and you'll be good to go.

There's a third component named *Post Process*. This is an extra tool that allows you to post process the text displayed. The available options are:

- **None** - Default option. Does no post-process.
- **First Uppercase** - Changes the first character (not symbol) to uppercase.
- **Title Case** - Changes all the first characters in each word to uppercase.
- **All Uppercase** - Changes all characters to uppercase.
- **All Lowercase** - Changes all characters to lowercase.

But where do you actually specify the translations? That's done inside the *Preferences Menu*. Press *Cmd* + *M* or click on *Game Creator -> Preferences* to open the menu and navigate to **Localization**.



The **Localization** tab has two sections: **Languages** and **Texts**.

Languages

Here you can add new languages and export and import the translations. For example, continuing the previous example, let's say we want to add the *Spanish* language and translate the previous message.

All you need to do is to click the **+** symbol and select the *Spanish* language. Then, click on **Export...** and select where you want to save the translation document. Translation documents are in *JSON* format.

If you open the saved document you'll see that there's only one entry with the text "*Hello, World!*". This is what needs to be translated. Let's change it to "*Hola, Mundo!*", save and close the document.

To update a language click on **Import...** and select the saved file. If everything is correct, the message *Last Update* will be updated to the current time. If something happened, an error message will be prompted in the console window.

To test that everything works, create a new **Action** and use the **Change Language** to select *Spanish*.

> When changing the language in runtime, all texts are updated instantaneously in realtime. There's no need to restart the application or reload the scene.

Texts

These are all the texts that can be translated. Notice that if you remove a *Game Object* that contained a component with a localized string it might not disappear from this list, so it is a good practice to review these strings and delete those that are unused, from time to time.

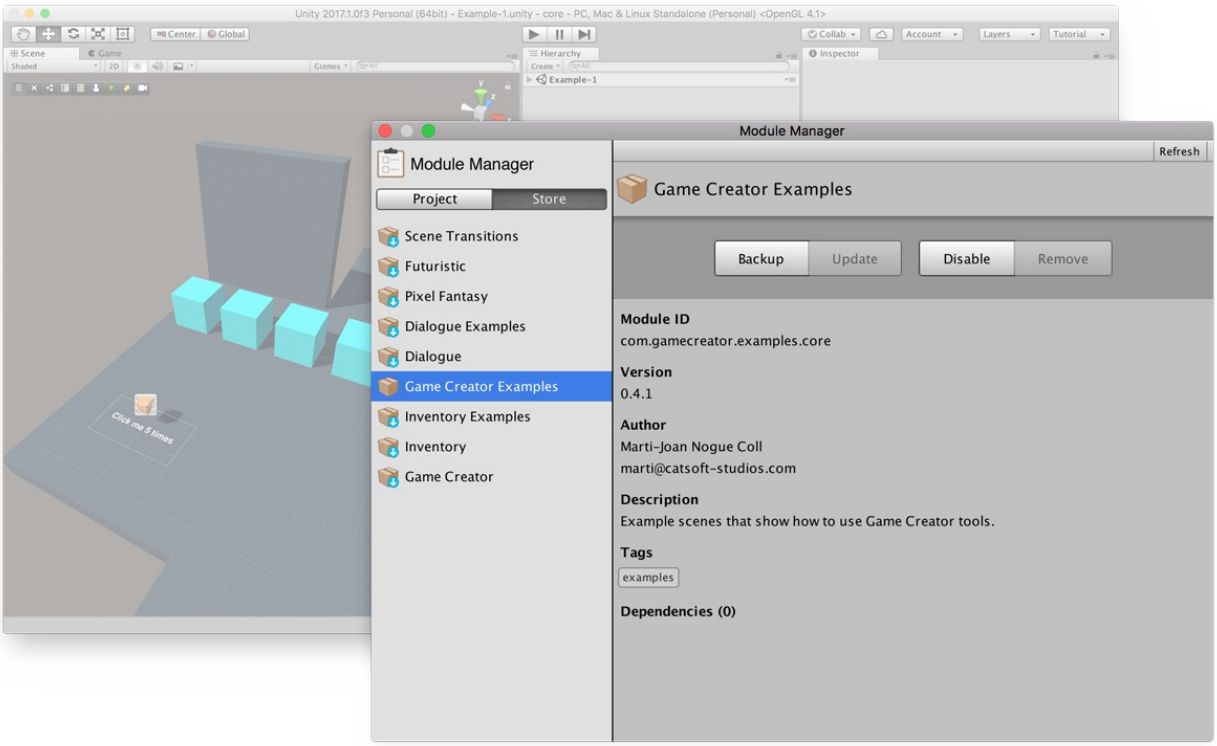
Best Practices

- Don't set any text as translatable until the very end of the project. You can always do that later.
- Keep a copy of all translation documents or save them inside the project. better yet, use *git* to keep older copies.
- Use the **Post Process** tool to transform the output of the text to fit your game. That way, translators don't need to keep in mind what should be written in capital letters and what not.
- Regularly check the texts from the *Texts* tab. Sometimes some texts that have been removed bypass **Game Creator**'s detection and still appear.

Module Manager

Game Creator is a large ecosystem of different parts brought together. Instead of shipping **Game Creator** as a single tool with lots of functionalities, it's split into what is called **Modules**.

A **Module** extends or adds new functionalities to the core tools. Their complexity can range from a simple new **Action** to a complete RPG toolbox with stats, buffs and animations.



> All modules are available for free at store.gamecreator.io

Import a Module

To add or import a module:

- If you download it from the Unity Asset Store skip to **Managing Modules**
- If you download it from our store.gamecreator.io, double click the `.unitypackage` file.

Managing modules

Once you've imported a **Module**, you can enable, disable or remove it. You can also create a backup copy of an enabled module if you don't use `git` or any repository management tool.

Backup

You can create a backup file from the select active **Module** at any time. Backups are located at the root of the project, inside a folder called `Backups/[backup-date]/`. The name of the file is the **id** of the module followed by its **version** number.

Enable

Enabling a **Module** allows you to work with the content of the module. In some cases, a **Module** might override some of your data (such as **Variables**). This happens mostly when using example modules that teach you how to use a certain tool (**Dialogue Examples**, **Game Creator Examples**, ...). If that's the case, a dialog box will prompt warning you of the consequences of enabling the module.

> Before enabling or disabling a **Module** it's a good practice to make a backup of your project.

Disable

Disabling a **Module** will remove all functionalities and scripts from your project, but as long as the installer is still present in your project, you can re-enable it at any time.

> Upon disabling a **Module** all data generated by this module will be removed. It's a good practice to make a backup before disabling it, and it's just one click away. For example, the **Inventory** module will remove all generated **Items** and **Recipes** from the project when disabling it.

Remove

When a **Module** is disabled, you'll have the option of deleting its *installer*. This is perfectly safe and will allow you to free some space from your project. Once a **Module** is removed, it will no longer appear in your **Module Manager** and you'll have to import it again to enable it.

Overview

Inventory is a complete solution exclusively developed for **Game Creator** that allows to have a fully functional inventory system in your game in just 10 seconds.

Key features

- Create **Items** from the *Items Catalogue* and define what happens when consumed.
- Create **Recipes** of two objects and customize the output using **Actions**.
- Use brand-new **Actions** to add, substract, buy, sell and combine items.
- Built-in currency system to trade with other NPCs.
- Easy to customize **Inventory**'s **UI**.

> **Inventory** comes with two examples of Inventory systems: One that emulates a typical RPG and another that looks like the old-school adventure games from *Lucas Arts*.

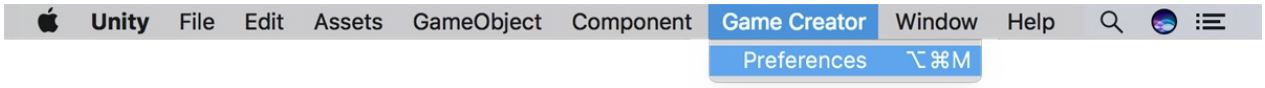
Setup

Import the package and wait for the scripts to compile. *Voilá!* You don't need to do enything else!

> **Important!** This module requires **Game Creator** and won't work without it

Preferences

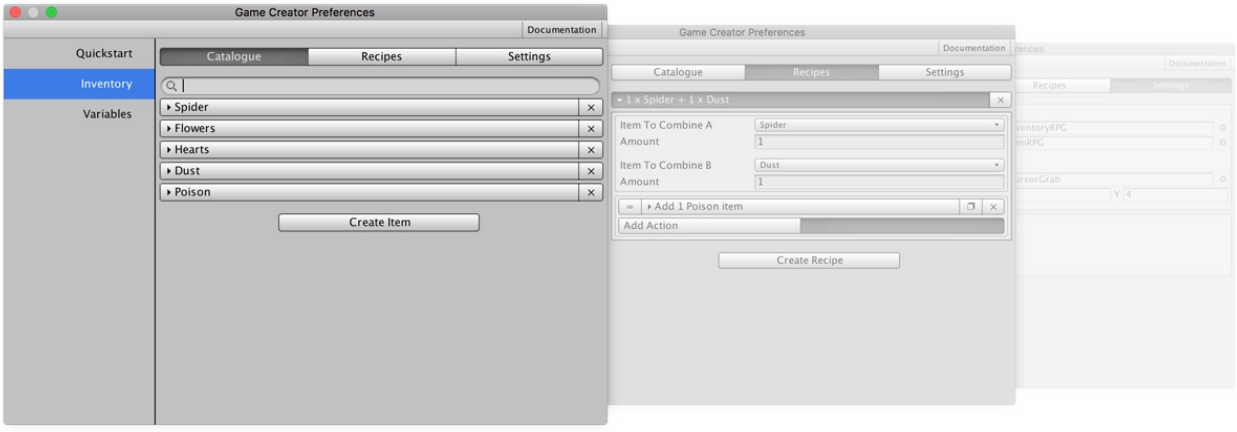
Before jumping into the *game view* you'll have to first define at least one **item**. **Items**, **Recipes** and everything related to the **Inventory** is located at the **Game Creator**'s preferences menu. To open it, simply click on the toolbar's *Game Creator* button and select *Preferences*.



> You can also open the **Preferences Window** using the keyboard shortcut `Alt + Ctrl + M` if you're on a Windows machine or `Alt + CMD + M` in macOS.

Preferences Window

Navigate to the **Inventory** left-sidebar button and you'll be presented with a screen with 3 options:

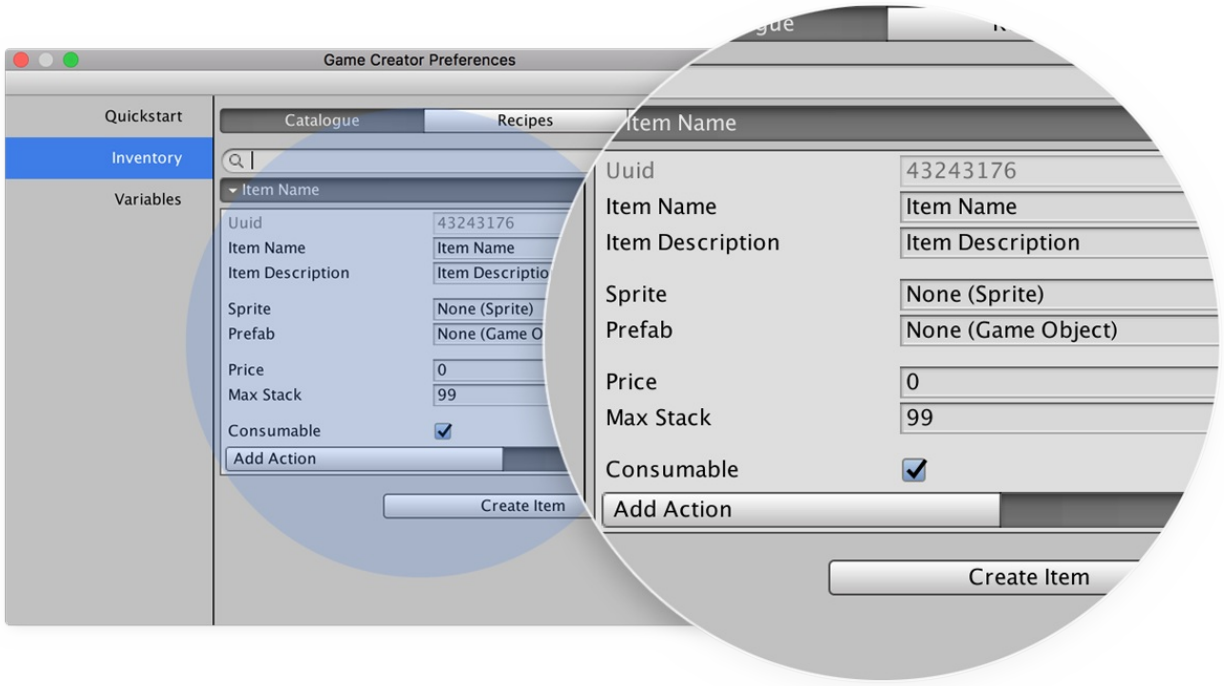


- Catalogue: Here you can create the game **Items** and give them properties
- Recipes: In this tab you can specify what happens when combining two items.
- Settings: Everything related to the **Inventory**'s settings is here.

Items Catalogue

Items are the main object of the **Inventory** module. The *Items Catalogue* tab in the *Preferences Window* allows to create and set the **Item**'s properties.

Here's an example of an empty **Item**:



An **Item** has quite a lot of options

- **Name**: The name of the item (displayed in the UI)
- **Description**: The description of the item (displayed in the UI)
- **Sprite**: The visual representation of the item (as seen in the inventory UI)
- **Prefab**: When dropping an item or instantiating it, this prefab will be used
- **Price**: The price this value costs (used when buying or selling an item)
- **Max Stack**: Maximum number of this items the player can carry in its inventory
- **Consumable**: Defines whether an item is consumable or not

If an item is marked as *consumable* then the **Actions** defined below will be executed when the player clicks on the item in the **Inventory**'s **UI** or another **Action** forces to consume the item.

For example, if you create a **Health Potion** you'll want to restore some player's health (which in this example will be saved in a **Variable**). Then the **Item** would look like this:

> **Note**: Notice that you are not restricted to add effects to the player when consuming items. You could, for example, instantiate another **Character** in front of the player when consuming an item named *Magic Lamp* which could be used to start a dialogue. The possibilities are only limited by your imagination!

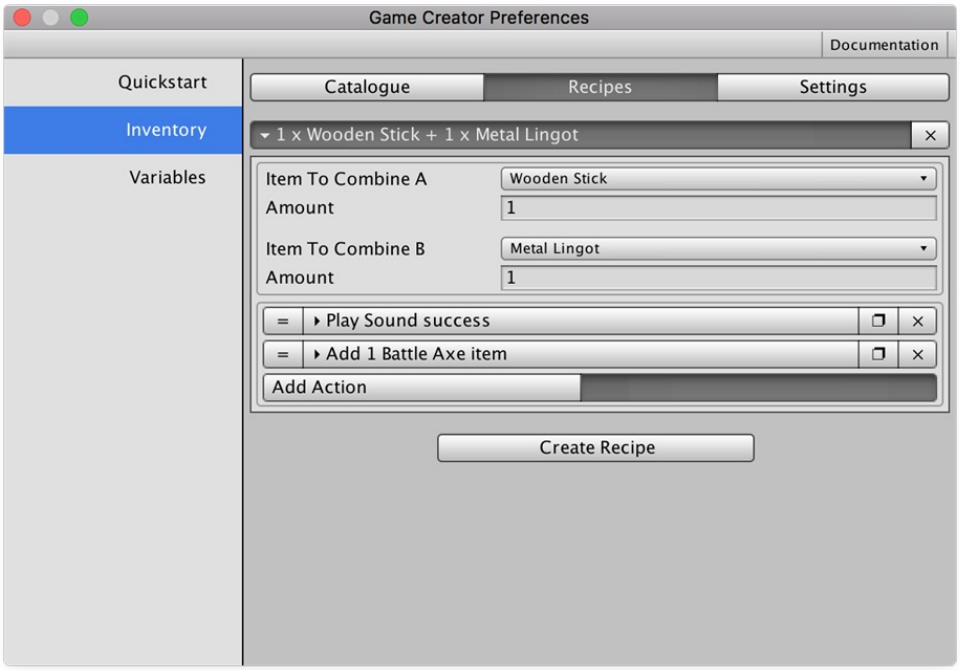
Recipes

A **Recipe** is the definition of the output of combining two **Items**. For example, a **recipe** for creating a *sharp sword* would be the combination of a *worn out sword* **item** with a *grinder* item.

> Of course, similarly to what happens with **Items**, you are not limited to creating other items when combining two. You can do whatever you can think of as long as there's an **Action** that allows it.

Creating a **Recipe** is as easy as clicking on the "*Create Recipe*" button on the *Recipes* tab in the *Preferences Window*. Then, select the two items you want to combine and the amount that are needed.

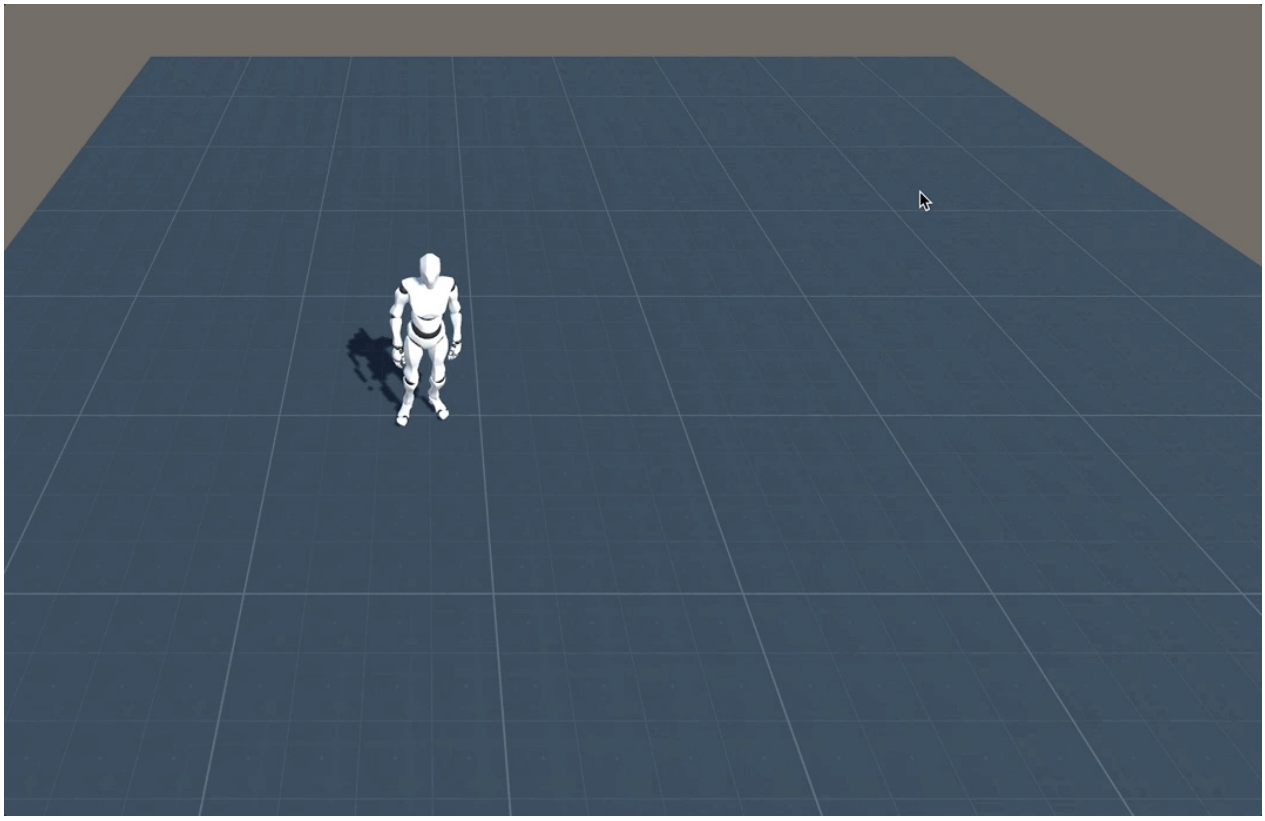
All that's left to do is specify the effects of combining two items.



In the previous screenshot, we can see how combining a **Wooden Stick** and a **Metal Lingot** will execute the **Actions** sequence defined below:

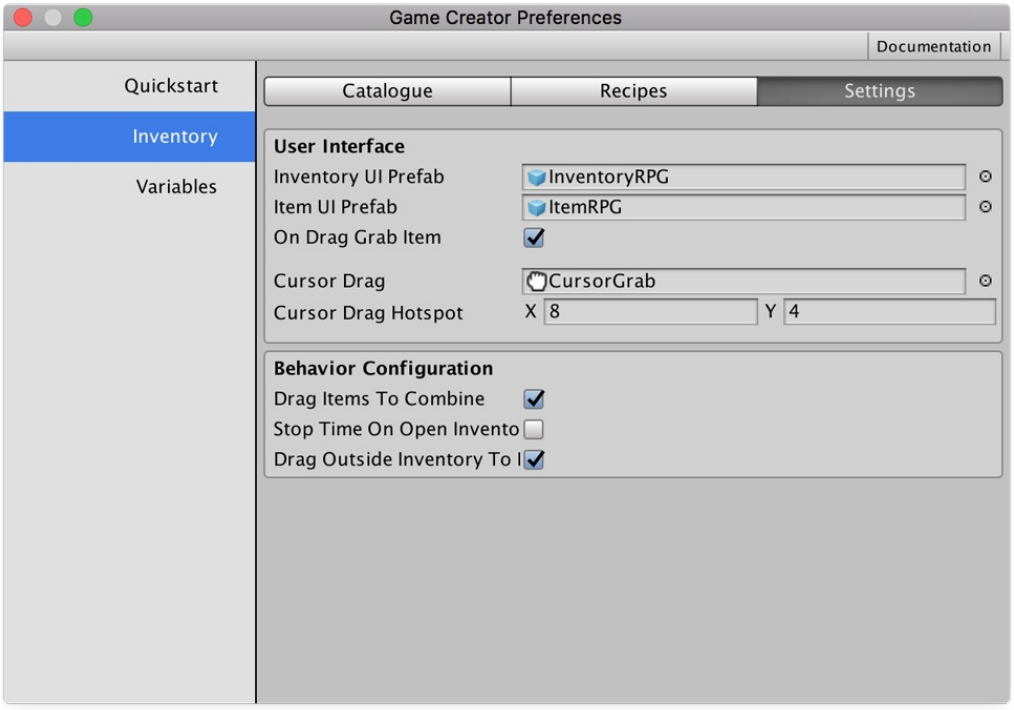
- Play a *success* sound effect
- Give the player an item called *Battle Axe*

You can tell the game to combine two **Items** using the **Recipe Action** that comes with the **Inventory** module or dragging and dropping one item onto another one in the **Inventory UI**.



Settings

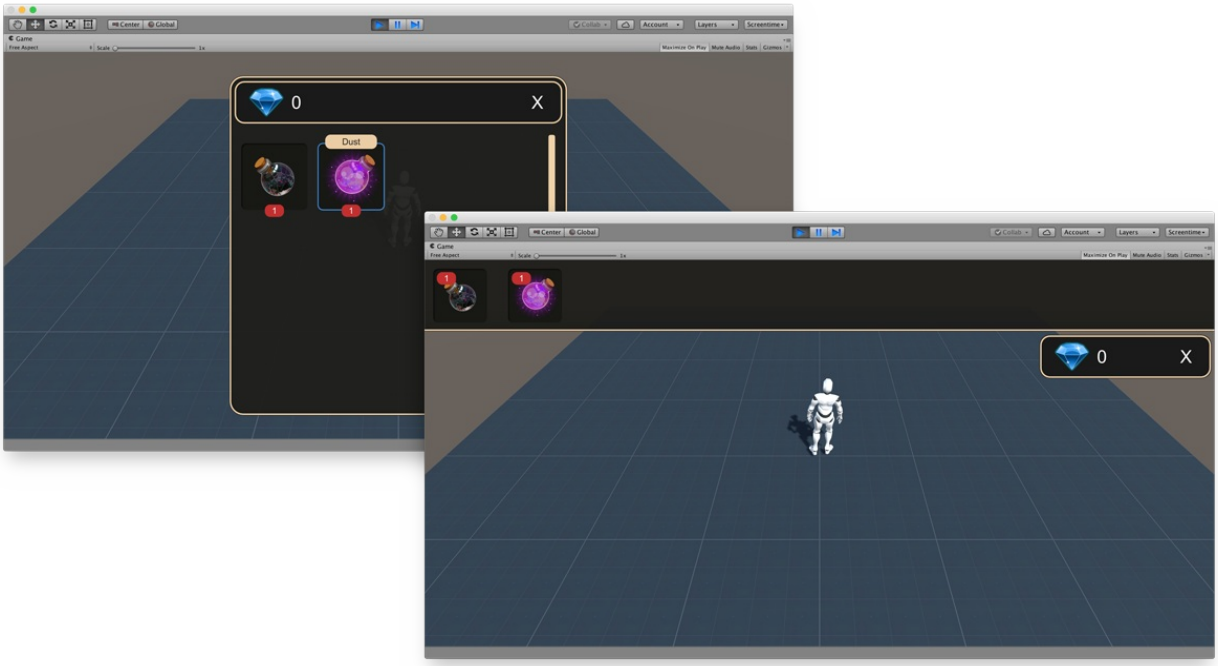
In the *Settings* tab you can configure all the necessary information related to the **Inventory** module. It's divided into different sub-sections such as *User Interface* and *Behavior Configuration*.



User Interface

In this section you can define how the **Inventory** looks like in the game.

Inventory UI Prefab and **Item UI Prefab** fields tell **Game Creator** what should be instantiated when the game requests to open the inventory. There's a complete chapter dedicated to this topic, but as a teaser, **Game Creator** comes with a couple of **Inventory** interfaces that can be seen in the image below. You can switch between them by setting the *InventoryRPG* or the *InventoryAdventure* prefabs in the **Inventory UI Prefab** field.



> See the Custom User Interface section for more information

The **On Drag Grab Item** toggle allows to enable/disable showing the **Item** picked below the cursor. This is only useful if you intend to disable all **Items** combinations. Otherwise, we recommend leaving it enabled.

Cursor Drag and **Cursor Drag Hotspot** are, as their name implies, the cursor used when dragging an item around.

Behavior Configuration

In the *Behavior Configuration* section you'll see options to set or unset **Inventory** features.

Drag Items To Combine option allows to enable or disable dragging and dropping an **Item** onto another to trigger a **Recipe** process. For example, if there's a **Recipe** with items *A* and *B* and the user drags *A* onto *B* (or the other way around), then they both will be destroyed and the **Recipe** actions set will be executed. If **Drag Items to Combine** is set to false, they will never be combined by dragging one onto the other.

Stop Time On Open Inventory allows to pause the game (the `TimeScale` is set to `0`) when opening the inventory, and resume when closed.

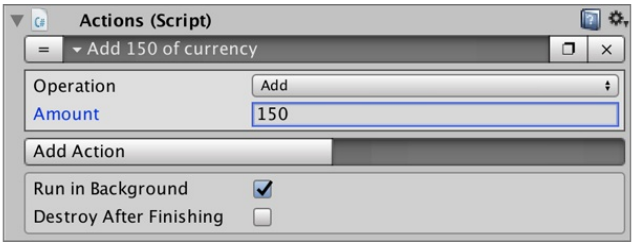
Drag Outside Inventory to Instantiate allows to drag items outside the **Inventory UI** window and instantiates the associated prefab in the scene. This feature is very useful. You can, for example, drop items from the **Inventory** and pick them up again from the scene, or you can use it to place mines in the scene for them to explode when a *Character* is nearby. Again, we provide the tools, you the creativity!

Actions

The **Inventory** module includes a set of **Actions** that complement **Game Creator**'s and allow to *do* things with **Items**, **Recipes** and the player's **Currency**.

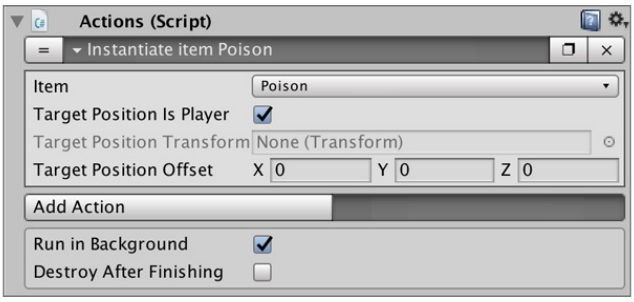
> **Note:** The *Player's Inventory* is where all the items being carried are stored. this inventory is automatically saved when calling the **Game Creator**'s **Save** Action so you don't need to worry about losing data between play sessions.

Action Currency



Allows to *Add* or *Substract* currency from the Player's Inventory.

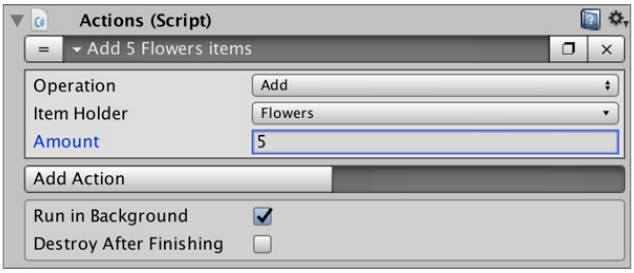
Action Instantiate



Instantiates (adds to the scene) a copy of the *Prefab* associated to the **Item**.

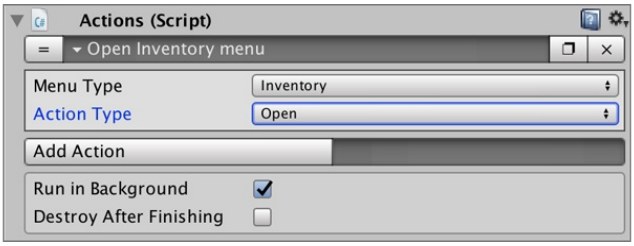
> For example, you can instantiate a 3D model of a potion that adds 1 Potion item to the players to simulate dropping and picking it again.

Action Item



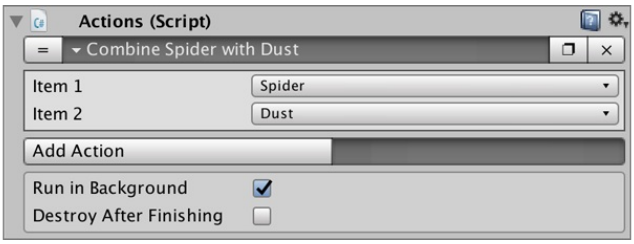
Allows to manipulate an **Item**. You can either *Add* the item to the Player's Inventory, *Substract* a certain amount of *Consume* the ones it has.

Action Inventory UI



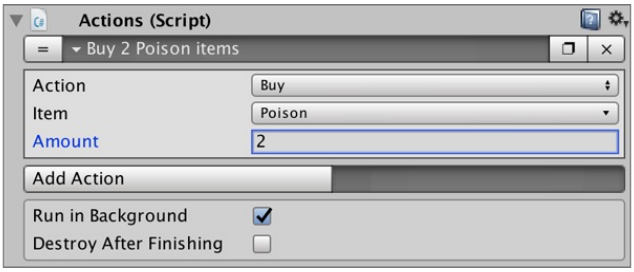
Opens or closes the Player's Inventory UI.

Action Use Recipe



Tries to find a **Recipe** with the two select **Items**. If it successfully finds one, it combines those objects executing the previously defined **Actions** associated to the **Recipe**.

Action Shop

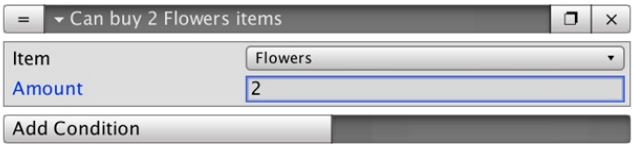


Allows to *Buy* or *Sell* an **Item** (and a defined amount of these). The difference between *Buying* and *Adding* is that an **Item** has a price. If the player doesn't have enough **Currency** to purchase the **Item/s**, then it won't do anything. The same applies to *Selling*. If the player is not carrying enough **Items** in his inventory, the transaction won't be fulfilled.

Conditions

The **Inventory** module includes a set of **Conditions** that complement **Game Creator**'s and allow to *check* whether an **Item** can be bought, if the player has a certain amount of items, etc...

Condition Can Buy

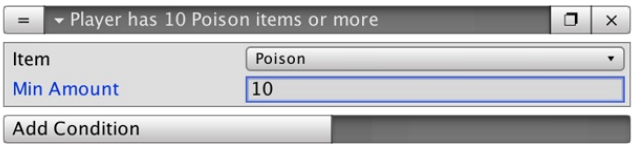


Checks whether an **Item** or an amount of **Items** can be bought with the current amount of currency.

Condition Enough Currency

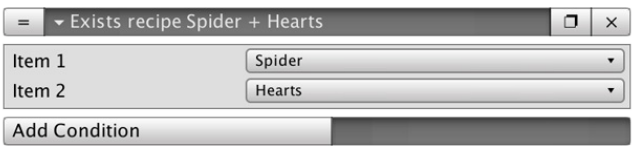
Checks whether the player has at least the amount specified of currency.

Condition Enough Items



Checks whether the player is carrying at least the amount of specified of items.

Condition Recipe Exists



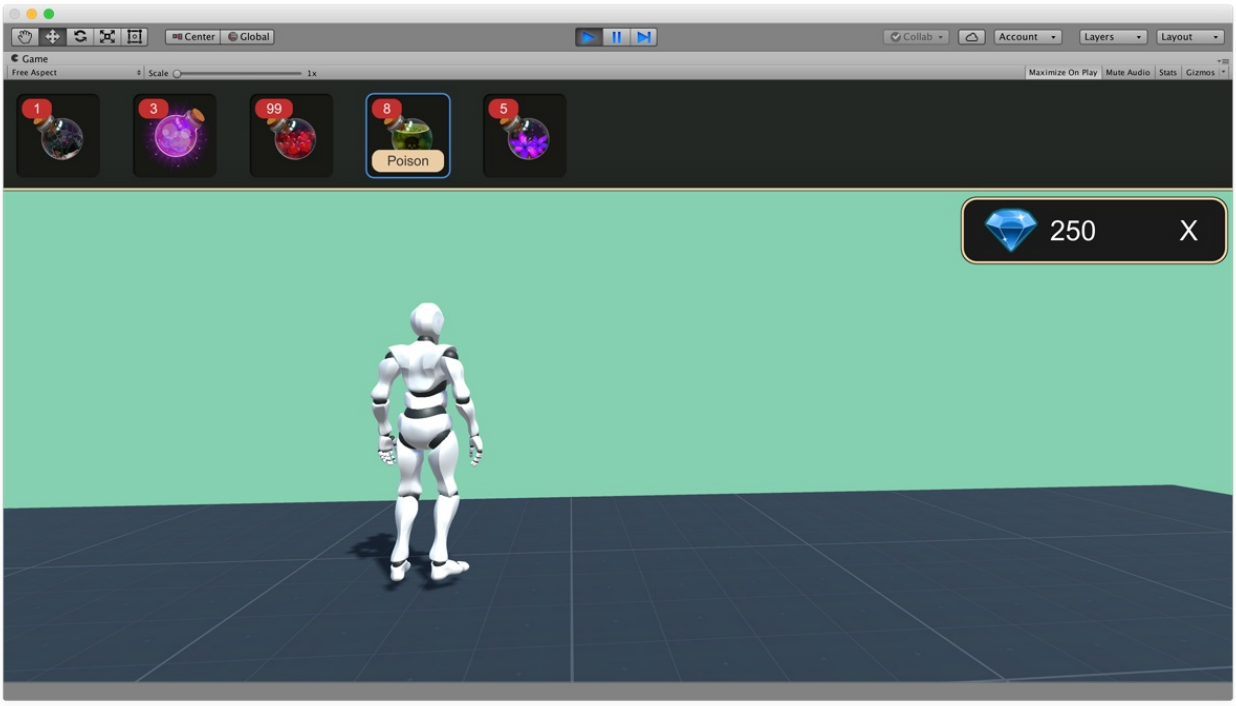
Checks whether a combination of two **Items** will result into something. If no **Recipe** is found to contain these two **Items** then it returns false.

Custom User Interface

Every game is unique in its own way and even though two games of the same genre might share some similarities, the truth is that there aren't two games with the same **inventory** visuals.

Inventory module comes with a couple of inventory representations. The first one is called **RPG Inventory UI** and looks very similar to the ones you find in most Western RPG games, such as *The Witcher 3: Wild Hunt* or *The Elder Scrolls: Oblivion* (*Skyrim* had a vertical layout). These types of inventories occupy most of the screen space and are perfect to *do things* within the inventory, such as combining items, consuming them, and so.

The other inventory representation reminds more of old-school adventure games such as *Day of the Tentacle*, *TellTale Game's Tales from the Borderlands* or any *Room Escape* game. This types of inventories leave as much screen space as possible and allow to drag and drop items onto the screen. These inventories are more fit if you intend to add more interaction outside the inventory than inside.



UI Overview

The only difference between these two inventories is their layout. Yes, seriously. We first created the **RPG Inventory**, we duplicated it and then, in less than 2 minutes, we had finished the **Adventure Inventory**.

An Inventory representation is composed of two prefabs:

- **Inventory UI Prefab**: Which is the actual representation of the inventory.
- **Item UI Prefab**: Which is the representation of an item that will be instantiated inside the **Inventory UI Prefab**.

We recommend that, if you want to create your own custom inventory, you duplicate any of the default prefabs we've set up.

> You can find these prefabs at: Assets/Plugins/GameCreator/Inventory/Prefabs/

Once you duplicate any of the **Inventory UI** and **Item UI** prefabs, drag and drop them in the Settings menu of the Inventory's Preference Window.

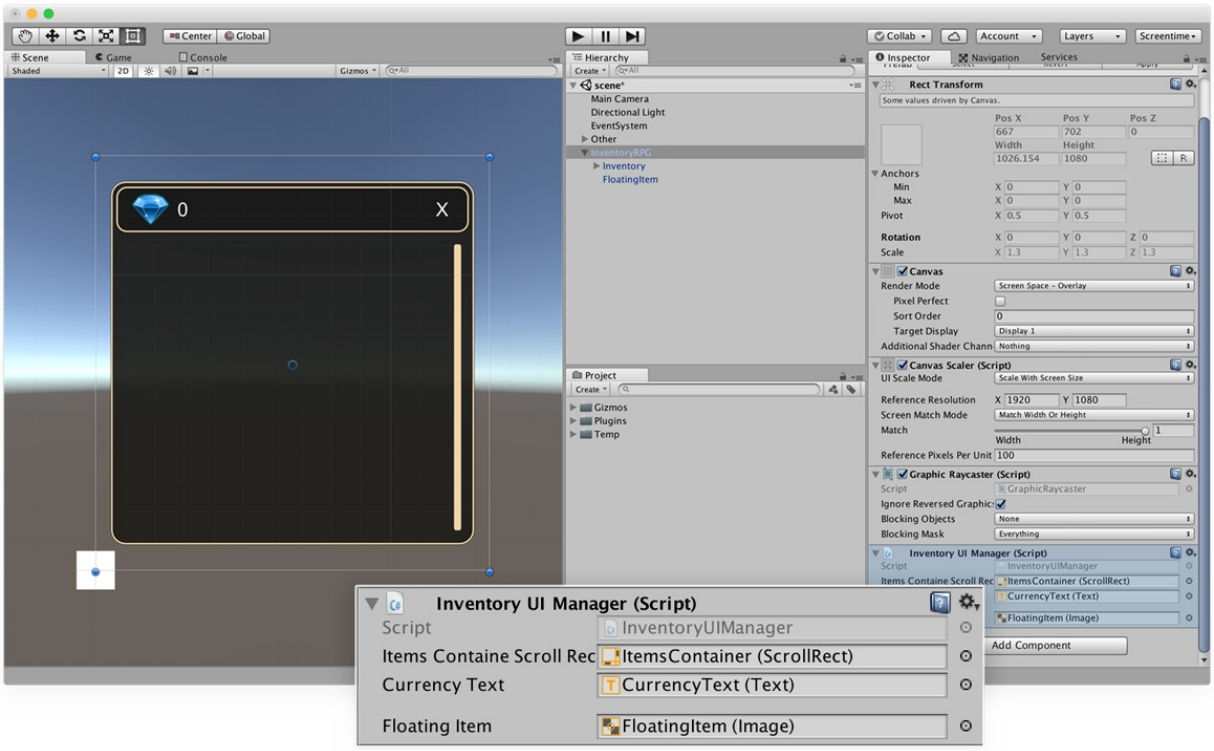
It's show time!

Customizing the Inventory UI

Customizing how the Inventory looks like is very easy. Just drop the duplicated prefab in the scene and change the sprites and layouts in order to look like how you want it.

The *items container* (where the items will be placed) should have an AutoLayout component. An auto-layout component can be either a HorizontalLayoutGroup, a VerticalLayoutGroup or a GridLayoutGroup.

> Note that the **RPG Inventory UI** has a `GridLayoutGroup` while the **Adventure Inventory UI** has a `HorizontalLayoutGroup` for their items container.



In the prefab root object, you'll see a component named `Inventory UI Manager`. This script is responsible for managing the inventory's UI. Drag the `Scroll Rect` of your prefab's item container inside the scroll field.

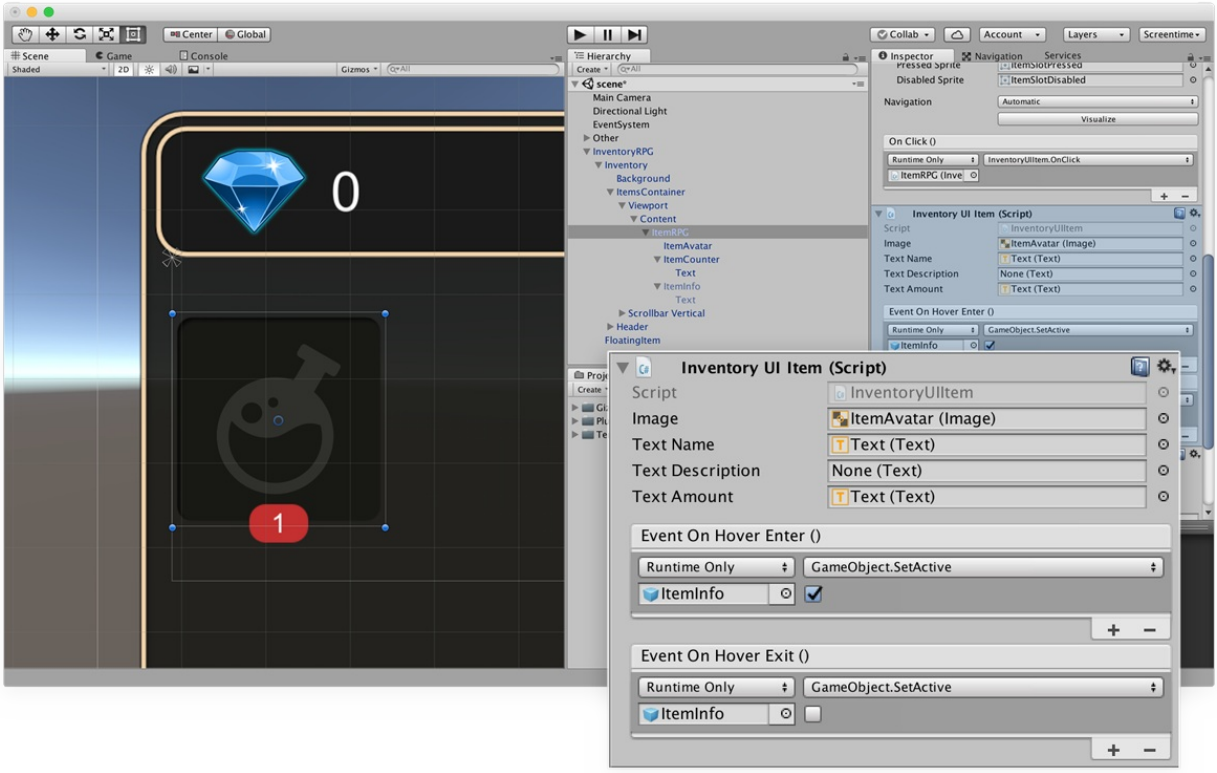
The `Currency Text` field is optional. This will update the `Text` component every time the currency amount the player has changes.

The `Floating Item` field is used when an item is dragged around the inventory and is positioned below the cursor. You should leave this unchanged. If you feel the dragged item image is too small, you can modify the white sprite at the bottom of the inventory and make it bigger.

Voilá! That's all you need to know about customizing your inventory!

Customizing the Items UI

Customizing how the items appear is even easier than the inventory. To begin with, drop the duplicated prefab of the **RPG Item** in the *items container* of the duplicated prefab of the **RPG Inventory** (see the image below). That way, you'll have a clean view of how it will look like.



The main component here is the `Inventory UI Item`. This scripts is responsible for showing how the item looks like. All its fields are optional, though it is recommended that you fill as much of them as possible.

For example, if you want to show the **Name** of the item, you can drag the `Text` component where name should appear and *Bam!* magic. The name will appear during gameplay in that component.

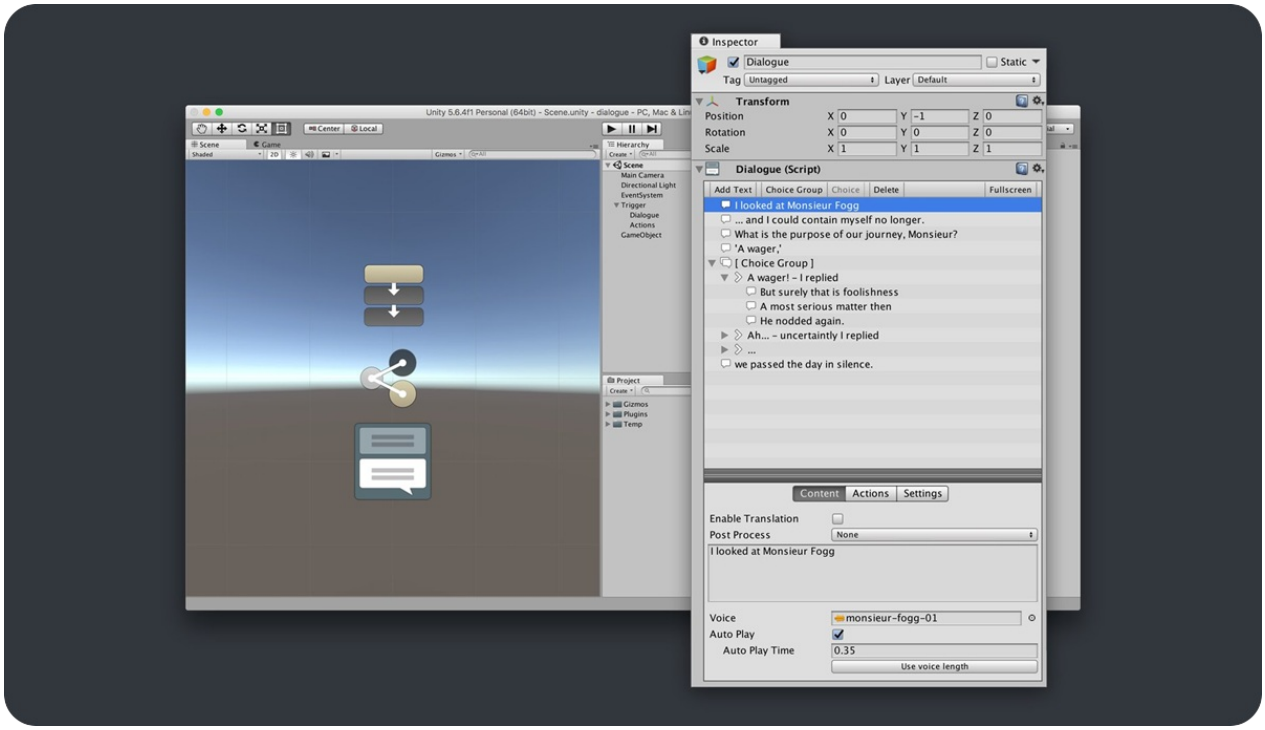
You can choose to show different properties of an **Item**:

- The sprite associated with this item
- The item's name
- The item's description
- The amount of items you have in the inventory

Don't forget to save your prefab (click *Apply*) and delete it from the scene view.

Overview

The **Dialogue** module is a **Game Creator** extension that allows to create and display in-game cutscenes and branching conversations between characters.



Key features

- Easily create **conversations** between Characters
- Add custom skins or download pre-made ones from the Game Creator Store
- Add **Conditions** to each line of dialogue
- Choose to execute Dialogue **Actions** first, last or simultaneously with the conversation line

> **Dialogue** comes a minimalist skin. You can create your own by duplicating the default one or import one from the store.

Setup

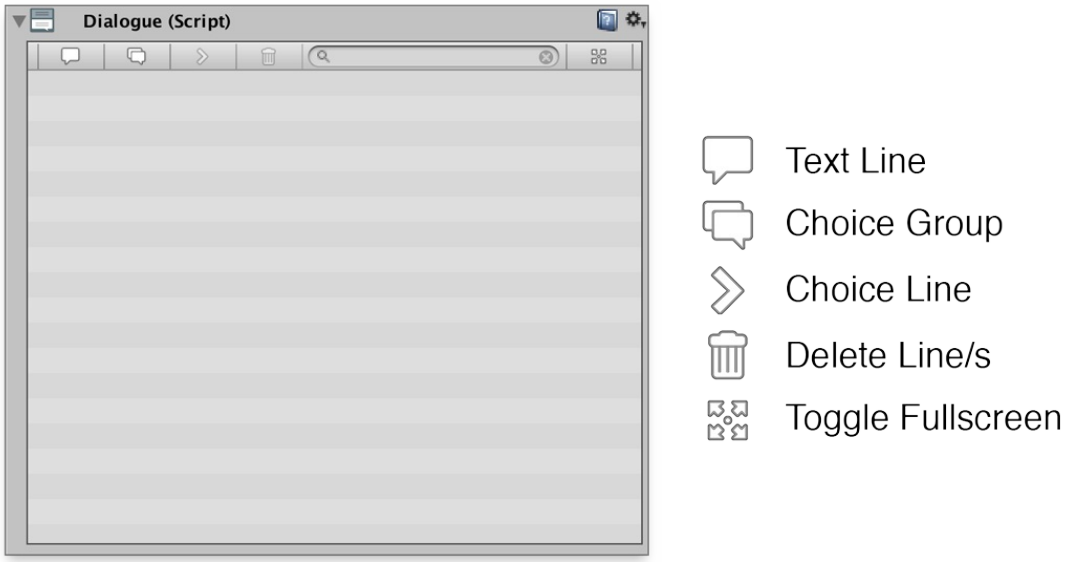
Download the package from the Game Creator Store or the Unity Asset Store. You'll first need to have Game Creator installed.

Then, bring up the *Preferences Window* clicking on the Game Creator option in the toolbar. Head to the **Modules** tab and click the Dialogue's **Enable** button.

> **Important!** This module requires **Game Creator** and won't work without it. Don't attempt to extract the package inside the Plugins/ folder as it will throw some errors.

Dialogue Anatomy

The Dialogue module is a **Component** that can be attached to any scene Game Object or *prefab*. To create one, right click on the *Hierarchy Panel* and select `Game Creator → Other → Dialogue`.



The Toolbar

The **Toolbar** is where all the Dialogue items are created. It's located at the top of the Dialogue component.

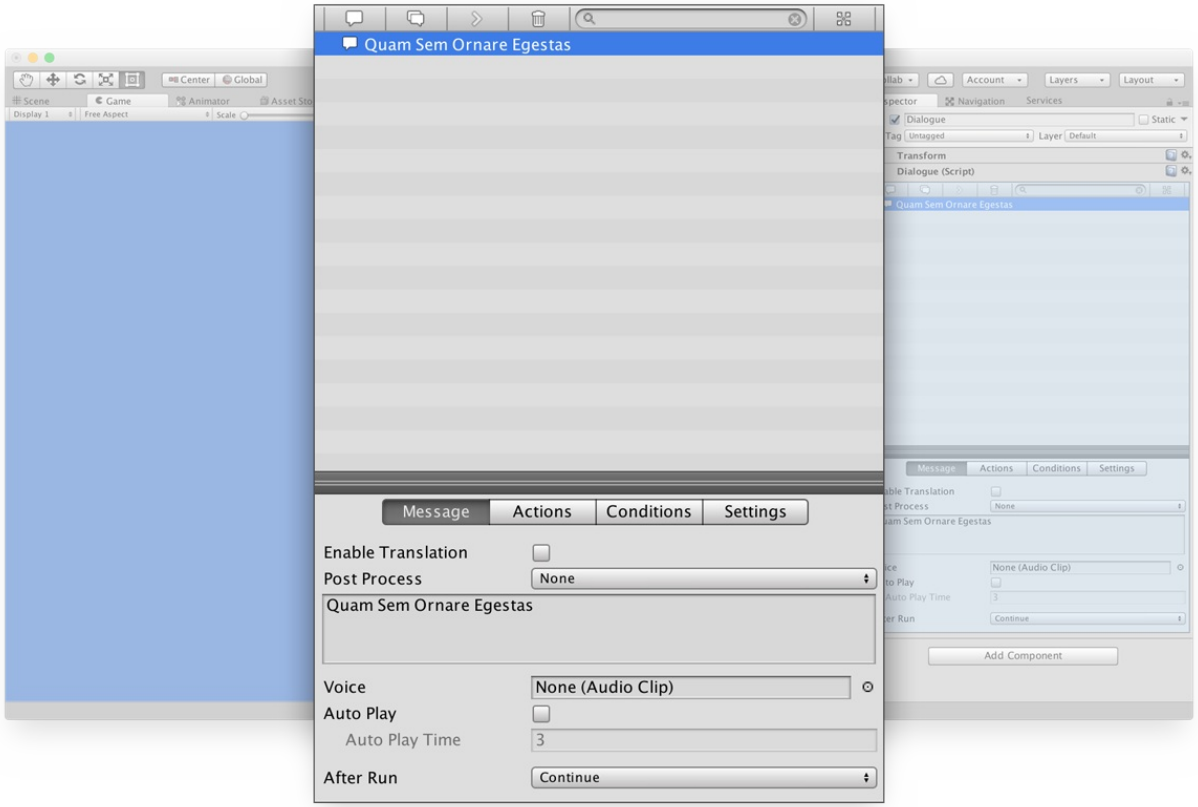
- **Add Text:** Create a dialogue line.
- **Add Choice Group:** Prepare the Dialogue for a set of choices.
- **Add Choice:** Can only be added below a Choice Group. Represent a choice line.
- **Delete element/s:** Removes all selected items.
- **Search:** Type to search a specific line of dialogue.
- **Maximize/Minimize:** Toggle between fullscreen and normal mode. Useful for large conversations.

Dialogue Elements

Any **Dialogue** is composed of the combination of 3 types of elements: **Text**, **Choice Groups** and **Choices**.

Text Elements

The most common element on a conversation between characters. The Text element allows to display a text following a set of rules defined by the **Conditions** tab and execute a set of **Actions** (if any) that can be found in its homonym tab. If a custom skin is provided, it will use it for this specific line of text. Otherwise, it will use the one set in the *Preferences Window*.



> It is highly recommended that you play a little with the different options and familiarize with can be done with the **Dialogue** module.

Additionally, you can also specify what happens after executing a **Text** element. The default behavior is that it continues to the next one (which is the most common option), but sometimes you might want to end the conversation upon reaching a certain point or jump to a previous node.

To do so, change the **After Run** property to **Exit** to end the dialogue or **Jump** to jump to another dialogue line. **Continue** will jump to the next element (if any).

Choice Group & Choices

Choice Group and **Choice** elements go hand in hand. A **Choice Groups** behaves like a **Text** element, but also allows to specify the characteristics and settings of the following choices presented to the player.

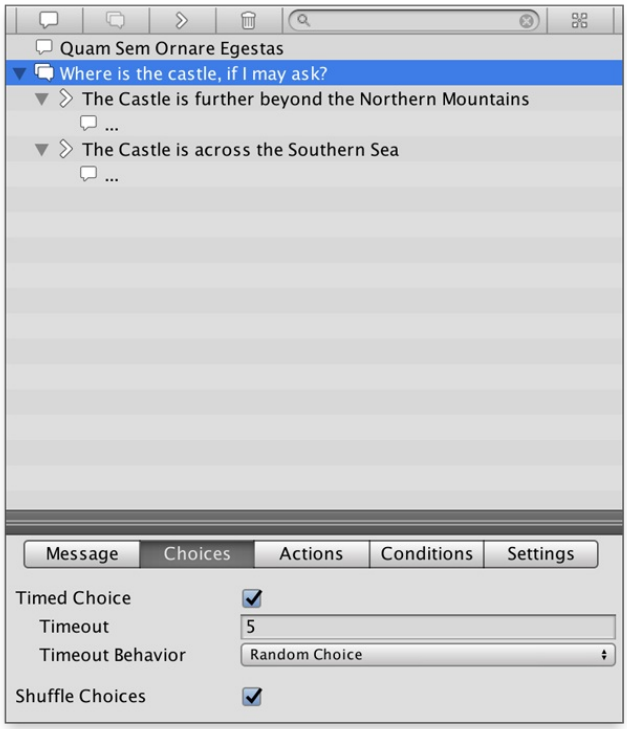
For example, imagine there's a *Merchant* that asks for directions to *The Castle*. The choices presented to the Player could be:

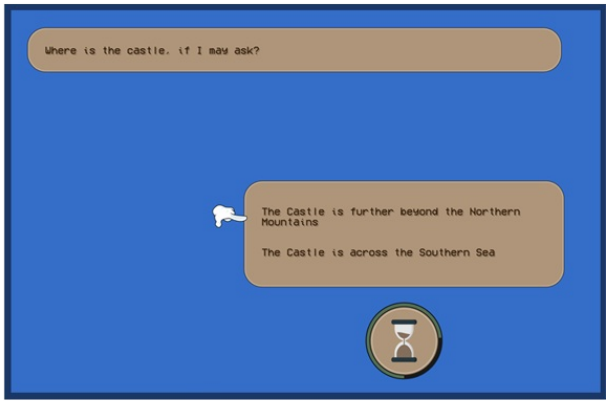
- "The Castle is further beyond the Northern Mountains"
- "The Castle is across the Southern Sea"

The **Choice Group** will allow you to configure if this choice is a **Timed Choice** as well as specify how much time has the player to make a choice. Also, if it's a timed choice, you can specify what happens if no choice is made:

- Select first choice
- Select a random choice
- Skip the entire choice group

A **Choice Group** will also allow you to randomly shuffle all choices at runtime.





On the other hand, a **Choice** element will work as a "gate" to a specific choice. They work more or less like a **Text** element. The difference is that only the player can use one.

Tips & Tricks

See there's a **Condition** tab inside a **Text Element**? These *Conditions* are checked before running the text line. If the result is negative, the line will be skipped.

> You can take advantage of this and skip entire conversations by dragging and dropping **Text Elements** inside another **Text Element**. If the upper-most element's *Conditions* evaluate as `false`, any of its children will be shown.

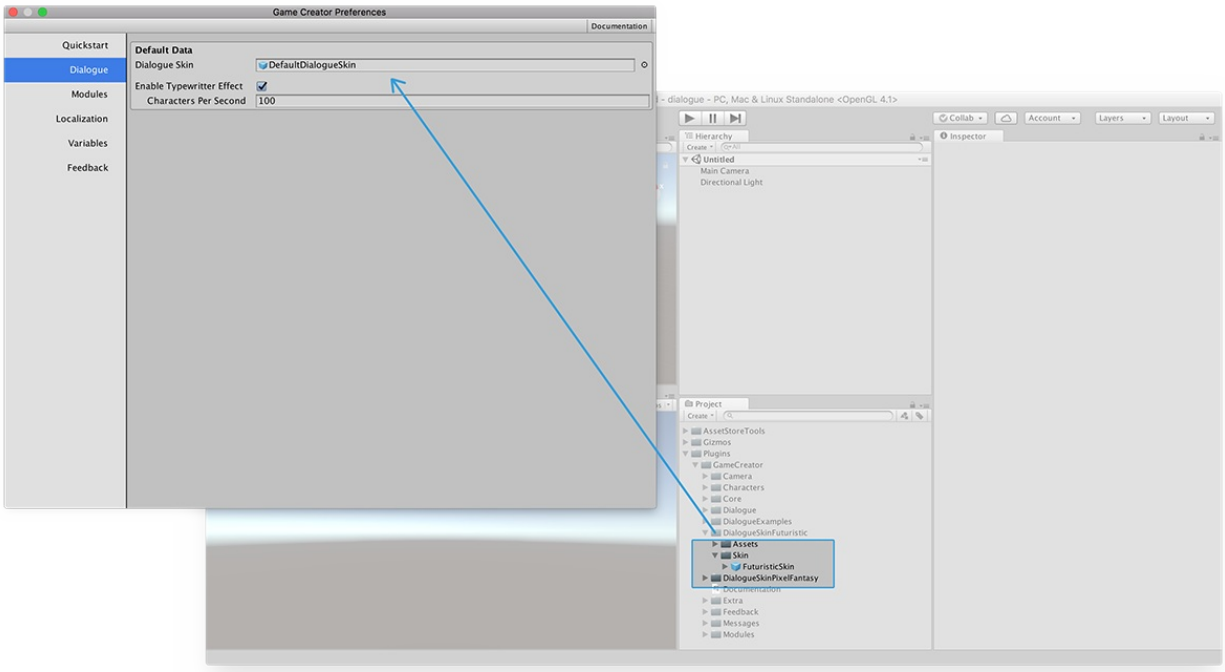
Skins

The **Dialogue** module is 100% customizable and isn't restricted by any type of UI style. You can head to the Game Creator Store and download any **Dialogue Skin** module and start using it right away.

Using a Skin

A **skin** is a prefab that is used whenever a conversation line or choice is displayed. They are automatically managed by the **Dialogue** module.

To test a **skin** all there is to do is to drag and drop the *Skin Prefab* onto the **Dialogue Skin** field found in the preferences menu.



> Additionally, each **Text Element** has its own unique **Skin** field in case you want to change the skin for a particular conversation line. Otherwise, the default skin will be used.

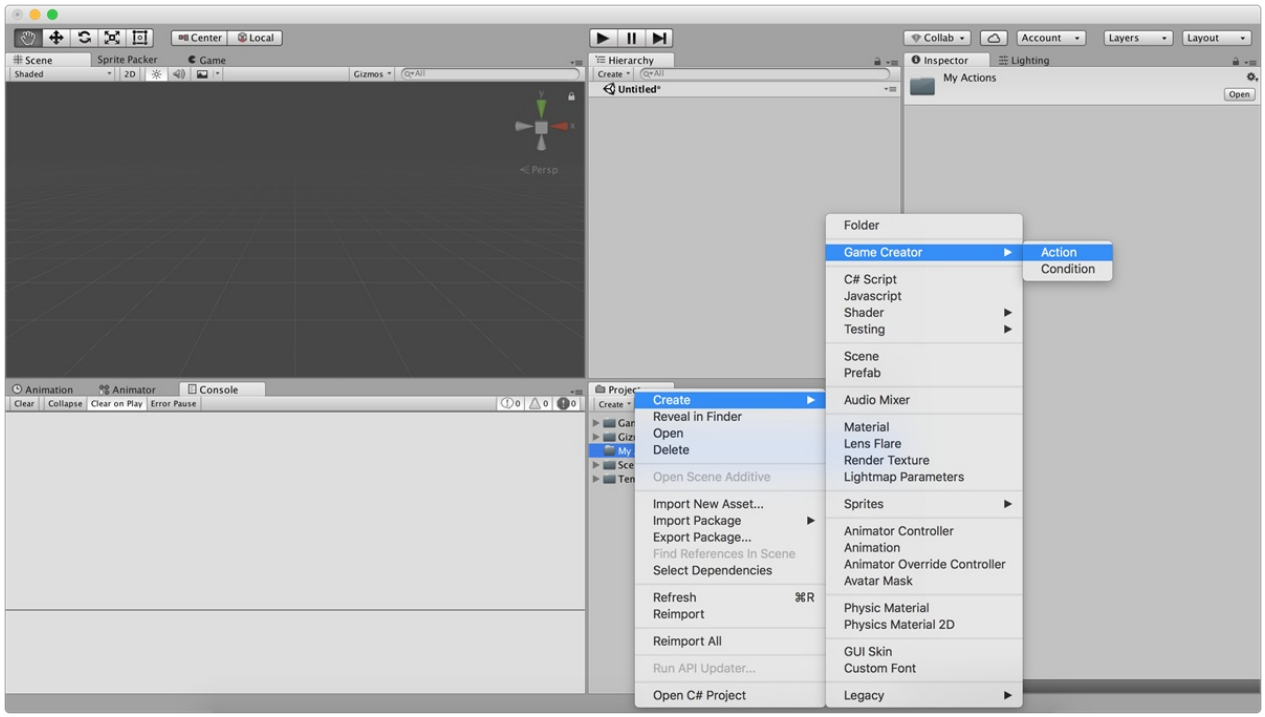
Custom Actions and Conditions

If you are a *programmer* you can extend the **Game Creator** functionality to fit your needs. This guide will cover the extendable components and how to use the **built-in tools** that fasten the development.

Custom Actions

Actions are the heart of **Game Creator** and thus is a process you'll be doing quite often if you intent to customize your game. Luckely we've created tools that allows you to easily create a *template* action which you can modify.

To create a new **Action** right-click on the *Project Panel* inside the *Unity Editor* and navigate to `Create → GameCreator → Action` and give the text file created a suitable name.



> We use the convention **"Action" + *name***, but you can use whatever you like.

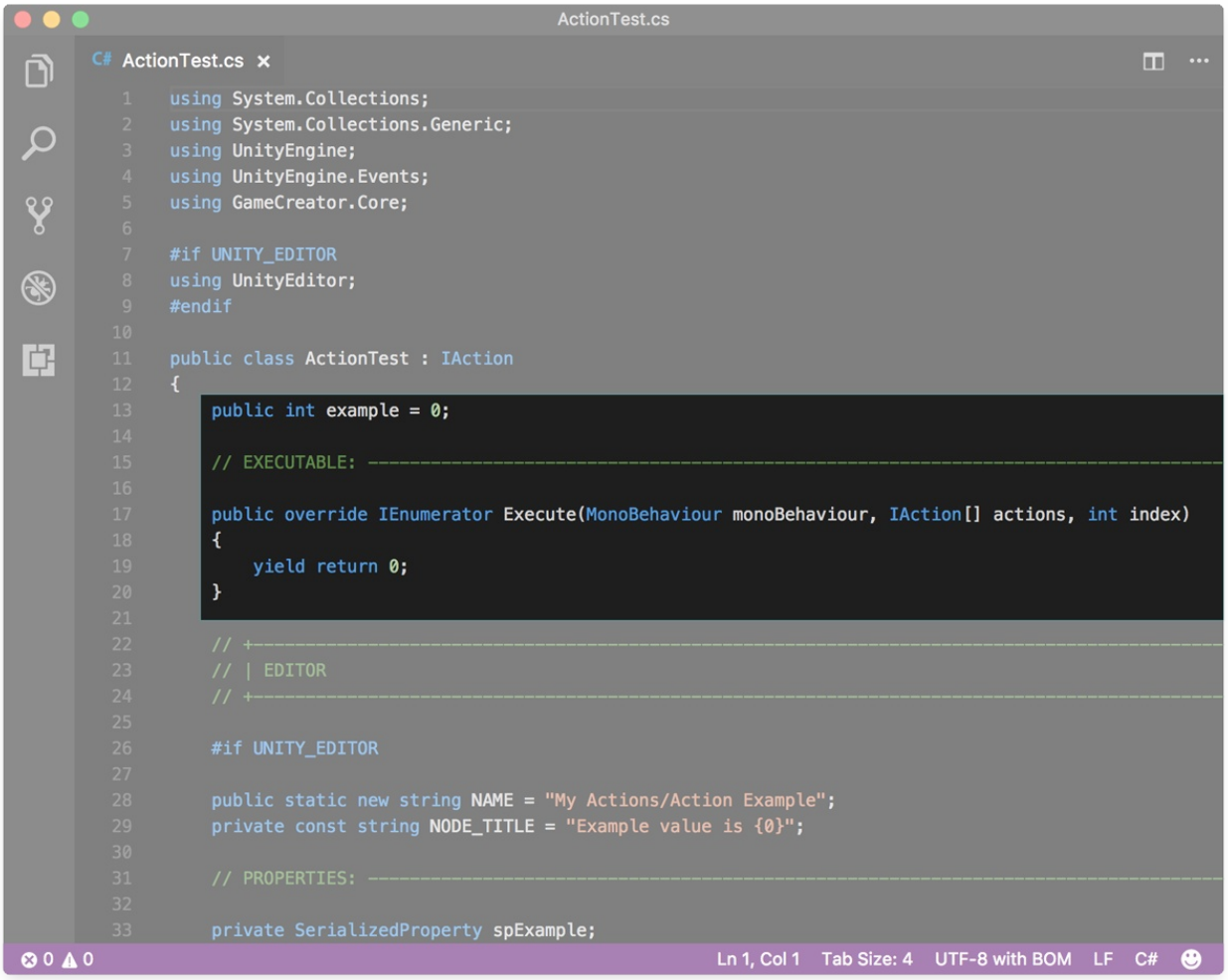
Open the new **Action** and see how there's already a bunch of code written. Notice how the **Action** inherits from an abstract class named `IAction`. To create a custom **Action** you don't need to know how they work. All you have to do is fill in the methods and you'll be good to go!

> All **Game Creator** scripts are organized inside the namespace `GameCreator` and each module has its own sub-namespace.

Creating a custom **Action** is divided in two parts. The first one, you'll be programming what the action actually does. The second part you'll need to tell Unity how you want this **Action** to be visualized in the *Inspector* and which properties are exposed. Let's begin with the first one.

Runtime body of an Action

The runtime body of the **Action** goes from the beginning of the action class definition till the platform compile condition pragma (where it's read `#if UNITY_EDITOR`)



You can declare your properties at the beginning of the class as you would normally do. The method `Execute` is called whenever an action is executed. Let's break down its structure and explain how it works.

```
public override IEnumerator Execute(MonoBehaviour mono, IAction[] actions, int index)
{
    yield return 0;
}
```

Notice how `Execute` returns an `IEnumerator` type. This is because this method is a coroutine. A coroutine works as any other function, except that it can halt its execution waiting for a value or for an amount of time. This is perfect for **Actions** as it allows to halt the execution independently of the cause.

> For example, in the **Character Navigation** module, there's an **Action** that allows to move a character from *A* to *B*. You can decide whether to wait for the character to arrive to its destination or continue the execution as soon as the character starts moving.
>
> The `Execute` method must always return A `yield return 0` (or any other integer number). This value is used to jump between instructions. Positive values skip forward instructions and negative ones re-execute previous instructions. By default you should always return 0.

The parameters are:

- `MonoBehaviour mono` : References the **Action** itself
- `IAction[] actions` : An array containing the full set of instructions
- `int index` : The index of the current **Action** in the previous parameter

Runtime body example

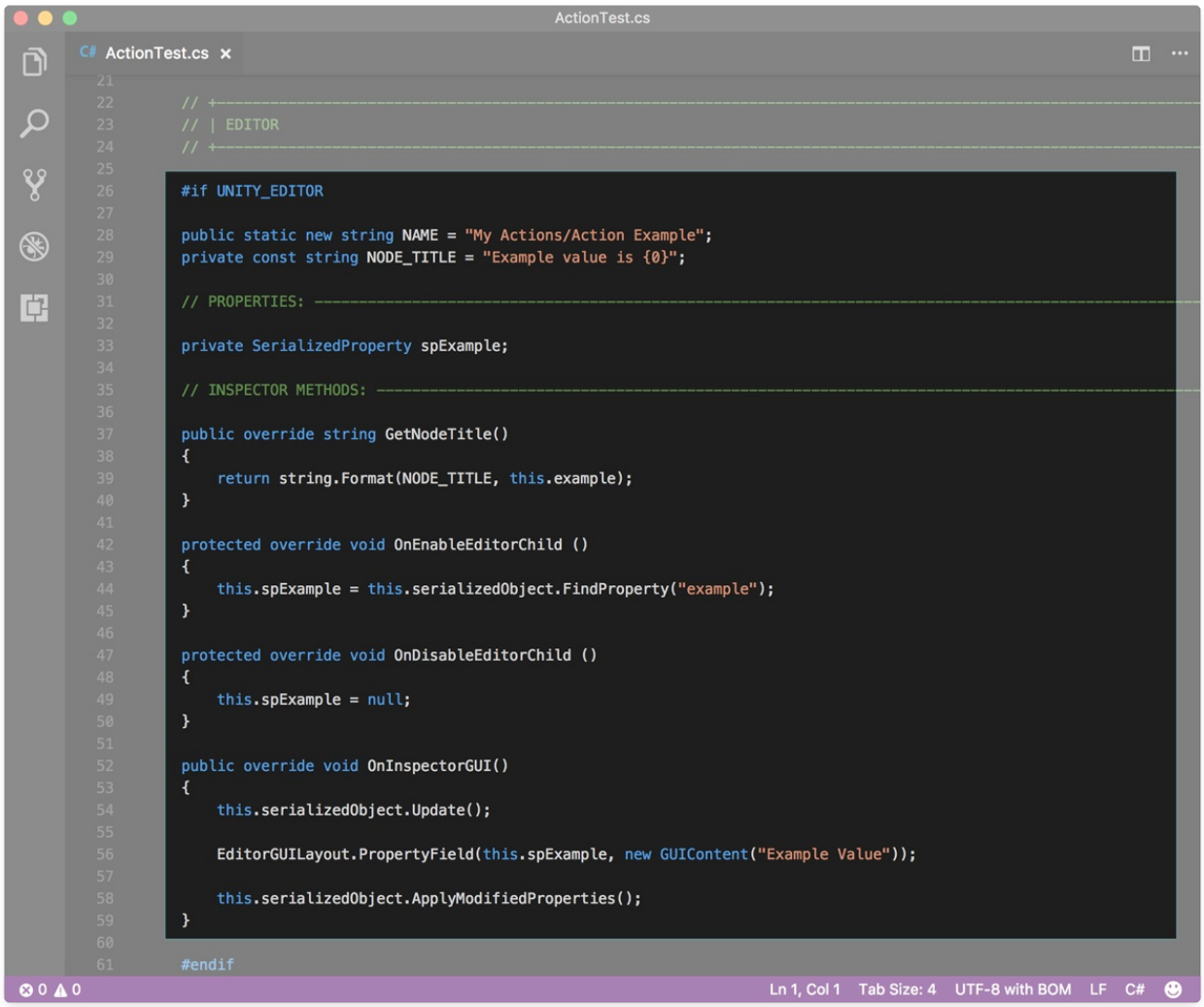
Here's an example of a custom **Action** called `ActionLookAtAndWait` . What it does is to make the `objectA` asset look at the `objectB` and wait 0.5 seconds before resuming the execution.

```
public Transform objectA;
public Transform objectB;

public override IEnumerator Execute(MonoBehaviour monoBehaviour, IAction[] actions, int
 index)
{
    this.objectA.LookAt(this.objectB, Vector3.up);
    yield return new WaitForSeconds(0.5f);

    yield return 0;
}
```

> Notice how after the `yield return new WaitForSeconds(0.5)` there's one last yield instruction that returns an integer (`yield return 0`)

Exposing an Action's options

Let's take a look at the second half of the code for creating a custom **Action**. Between the platform compile condition pragmas (`#if UNITY_EDITOR` ... `#endif`).



First of all, there is a *static* property with the *new* keyword called `NAME`. This property gives the **Action** a name in the **Action**'s selection dropdown list. Just change the name and you're good to go.

The second property is a *const* string named `NODE_TITLE`. This one is used in the method `GetNodeTitle()` which returns a string containing the description of the **Action** shown in the *Inspector*.

> Instead of using a constant string, **Game Creator** gets the description of an **Action** using the `GetNodeTitle` method so it can dynamically change its content based on the **Action**'s properties.

`OnEnableEditorChild()` and `OnDisableEditorChild()` are called when the *Instector* containing this **Action** is focused/unfocused. It's a good place to initialize the properties used in the `OnInspectorGUI()` method.

The `OnInspectorGUI()` is where the thick code goes. It is called every refresh frame. It should always start with `serializedObject.Update()` and finish with `serializedObject.ApplyModifiedProperties()`. Between these two lines you should show the serialized version of the properties.

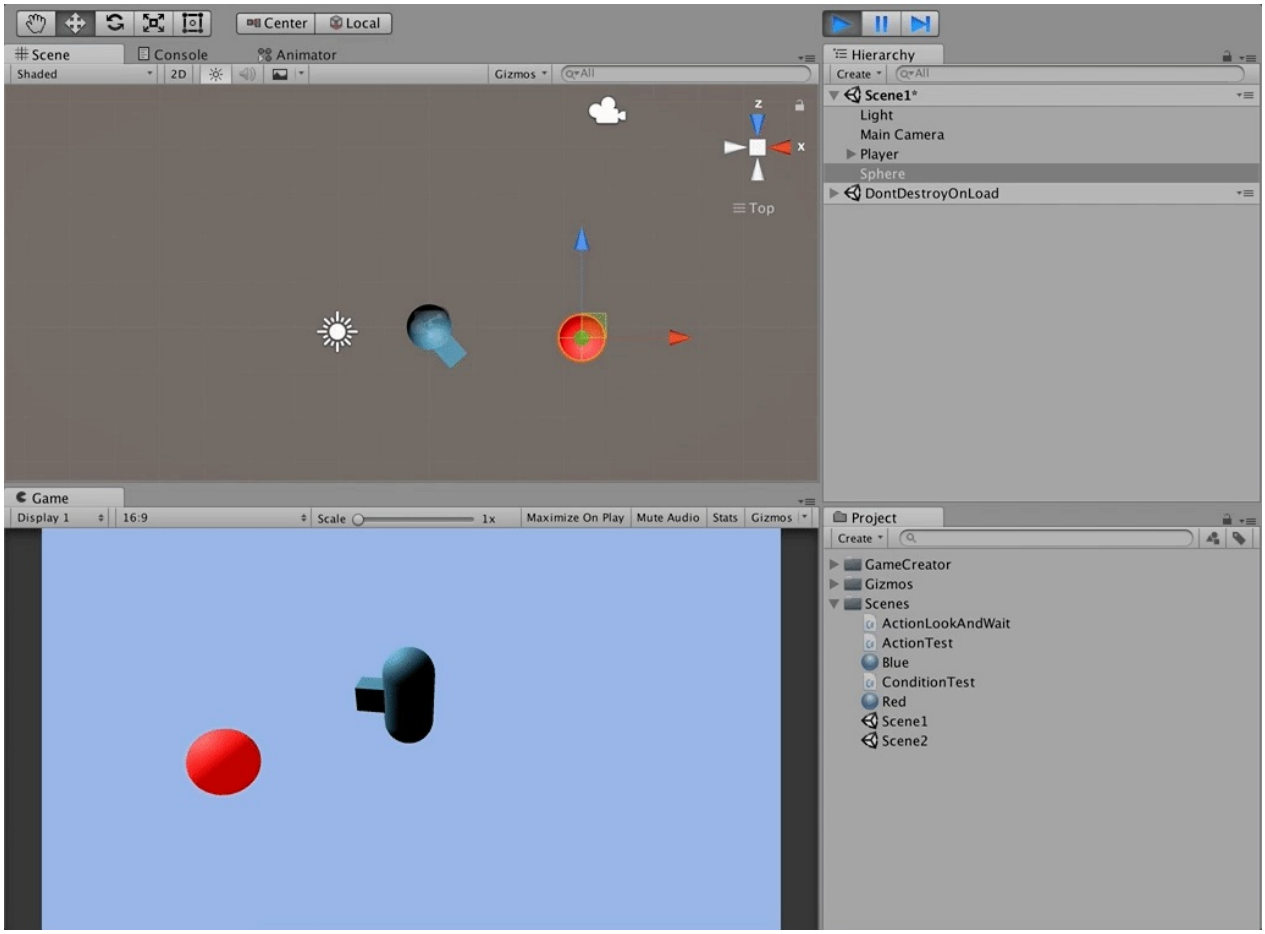
For more information on working with *SerializedProperties* follow this Unity guide.

> Unlike the `Update` method, *Editor* scripts are only refreshed when they detect a change

Here's the *Editor* code section of the **ActionLookAtAndWait**

```
#if UNITY_EDITOR

public static new string NAME = "My Actions/Look and Wait";
private const string NODE_TITLE = "Object {0} looks at {1}";

private SerializedProperty spObjectA;
private SerializedProperty spObjectB;

public override string GetNodeTitle()
{
    return string.Format(
        NODE_TITLE,
        (this.objectA == null ? "none" : this.objectA.gameObject.name),
        (this.objectB == null ? "none" : this.objectB.gameObject.name)
    );
}

protected override void OnEnableEditorChild ()
{
    this.spObjectA = this.serializedObject.FindProperty("objectA");
    this.spObjectB = this.serializedObject.FindProperty("objectB");
}

protected override void OnDisableEditorChild ()
{
    this.spObjectA = null;
    this.spObjectB = null;
}

public override void OnInspectorGUI()
{
    this.serializedObject.Update();

    EditorGUILayout.PropertyField(this.spObjectA);
    EditorGUILayout.PropertyField(this.spObjectB);

    this.serializedObject.ApplyModifiedProperties();
}

#endif
```

Wrapping and putting everything together, you can see this custom **Action** in a real project. Notice how the square looks at the objectB (red ball) and waits 0.5 seconds before looking at it again.

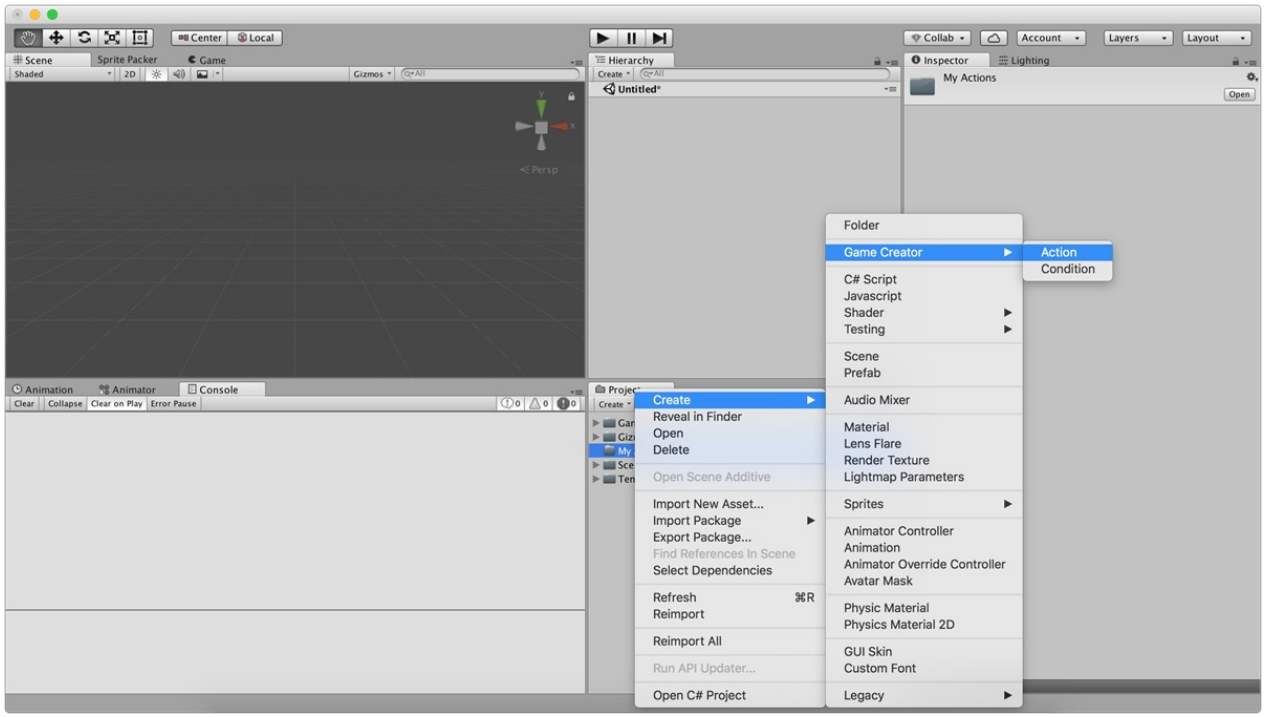


Custom Conditions

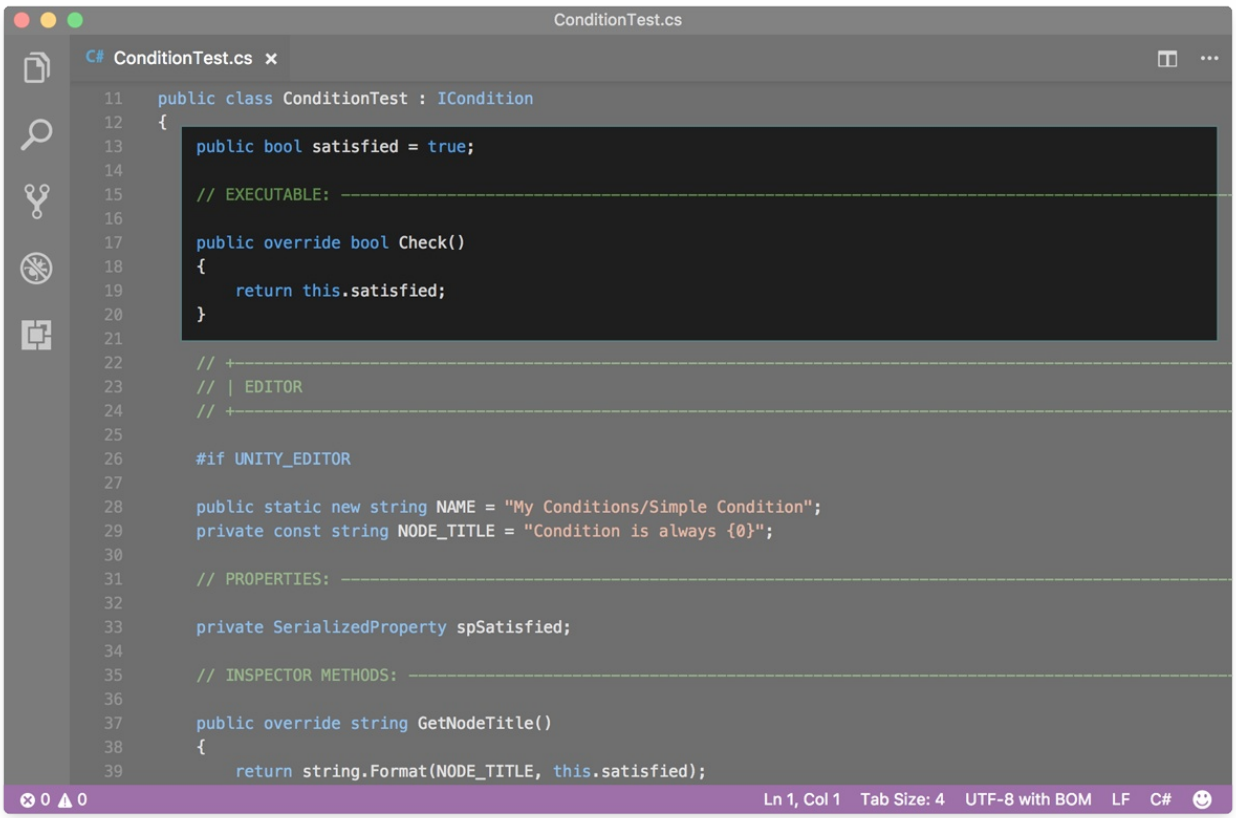
Creating custom **Conditions** is much like creating custom **Actions** but even easier. If you haven't taken a look at how to create **Actions** it is encourage that you do so, as the process is practically the same.

To create a custom **Condition** right-click in the *Project Panel*, select `Create → GameCreator → Condition` and give it a suitable name. Because the template file is a too simple **Condition** example, we're going to create a more complex one through this manual. We're going to create a **Condition that checks whether two objects are within a certain radius.

> We use the convention **"Condition" + *name***, but you can use whatever you like.



A **Condition** code is divided in a *runtime* section and an *editor* section.



The runtime section describes how it checks whether *something* is true or false. To do so, declare your properties below the class definition and override the `Check` method. To create the new **Condition** let's remove the `isSatisfied` property and create 3 others:

```
public Transform objectA;
public Transform objectB;
public float maxDistance = 5.0f;
```

And finally fill the `Check()` method. To check whether `objectA` is within a distance of `maxDistance` respect `objectB` we calculate the distance between the two objects and return if the value is lower than the threshold.

```
public override bool Check()
{
    if (this.objectA == null) return false;
    if (this.objectB == null) return false;

    float distance = Vector3.Distance(
        this.objectA.position,
        this.objectB.position
    );

    return (distance <= this.maxDistance);
}
```

> The *Editor* section of a **Condition** is exactly as the **Action**'s. For the sake of space, we're not covering the same thing again.

Custom Hooks

Hooks are *Unity* components that allow you to easily access unique objects such as the *Player* or the *Main Camera*. Their use is not required but useful if you want to simplify the process of selecting common objects.

> Let's say you want to detect when the player enters a certain area with a **Trigger**. Instead of setting the **Trigger** to *Enter* and dragging and dropping the *Player* game object into the target field, you can simply mark the `collideWithPlayer` checkbox and it automatically looks for the *Player*.

Creating a custom hook is really easy. **Game Creator** already provides a hook for some components but if you want, for example, to create one for the player's *Dragonfly*, you can create a class that inherits from `IHook` as follows:

```
public class HookDragonfly : IHook<HookDragonfly> {}
```

> Notice that using **Hooks** requires to use the `GameCreator.Core.Hooks` namespace

That's all. Add this component to you game object and you'll be able to access it using `HookDragonfly.Instance`.