Game Creator Documentation

From Zero to Hero

Catsoft Works

Table of contents

1. Game Creator

1.1 Welcome to Game Creator

Every game begins with an idea – a world to build, a compelling game mechanic, a feature that players are bound to fall in love with – but it takes a lot of work to bring that idea into fruition. Game Creator is a collection of tools to help make the journey from idea to playable game a lot smoother.

Acronym

Game Creator is sometimes informally abbreviated as **GC**.

1.1.1 Who is it for?

Game Creator is the perfect tool for both beginners and experienced users.

- Newcomers will find an easy-to-use tool with a very smooth learning curve, thanks to the small amount of concepts one has to learn in order to get started.
- Experienced users will find that these small set of tools have a lot of depth and can be synergically used to create any mechanic with ease, while favoring quick iteration.

Game Creator also has a very straight-forward API for programmers, from which they can extend the tools with new features and seamlessly integrate them with the rest of the ecosystem of tools. Level and art designers can quickly test their environments, creating a playable character and a camera type that fits their game with just a couple of clicks. And game designers will be delighted with a pletora of tools that they can use and exploit to create intrincate game mechanics.

1.1.2 How to get started

The easiest way to start learning how to use **Game Creator** is to jump to the Getting Started section. It overviews everything you need to know to get up to speed and assumes you have no technical knowledge. It also contains links to other learning resources from where to learn more.

1.1.3 What is it?

The **Game Creator** package comes with a slew of tools that help you very easily make the game of your dreams. These tools have been carefully crafted to be as flexible and intuitive as possible. Each tool takes care of dealing with the heavy-math under the hood and present it to you in a very human-friendly form, so you can focus on what really matters: Making games.

- **Characters**: Characters are entities living in your scene. These come loaded with common features, such as inverse kinematics, obstacle avoidance navigation, user input, jumps, footstep sound effects and animation systems.
- **Cameras**: Cameras allow to control how your game is framed. From an orbiting third-person perspective with zoom and geometry clipping avoidance to more traditional fixed camera angles, top-down perspectives or first-person views.
- **Visual Scripting**: Visual Scripting in Game Creator is very unique: Instead of using a typical node graph, it borrows the concept of task lists. This makes it really easy to read, organize and keep all interactions under control without the project quickly becoming a spaghetti mess.
- **Variables**: Variables allow to keep track of the game's progress and storing it when the user saves the game.

More tools

Game Creator comes with more tools than the aforementioned above. However, we recommend beginners focus on understanding these first. Experienced users and programmers can jump to the Advanced section to know more about the rest.

1.1.4 Modules

Game Creator is built to be extremely flexible and extensible. **Modules** are add-on packages that extend the features provided even further. For example, the **Inventory** module allows to easily define items with different properties, which can later be equipped, consumed, crafted, dropped, sold, bought or stored in chests.

- Inventory: Manage and equip items, craft new ones and trade them with other merchants.
- Dialogue: Create conversations with other characters with branching narratives.
- Stats: Make complex RPG interactions with intertwined stats, attributes and status effects.
- Quests: Keep your game's progress and lore under control with a mission manager.
- Behavior: Easily manage character's AI using Behavior Trees and other mechanisms.
- Perception: Allow entities to use sight, smell or hearing to understand the world.
- Shooter: Create long-ranged shooting mechanics.
- Melee: Define close quarter combat mechanics wih parries and combos.
- Traversal: Give characters the ability to climb and other traversing skills.

Modular synergy

Modules do not just extend **Game Creator**'s capabilities, but can also communicate with other **Modules**. This allows to intertwine their features and develop even more complex game mechanics.

Example of use case

A very common case is using the **Dialogue** module along with the **Stats**. The first one allows to easily manage conversations between characters, where the player is prompted with choices and characters react to these. The **Stats** module, on the other hand, allows to define RPG traits to objects.

By combining these two modules you can create more interesting mechanics, such as displaying an option during a conversation with a character, where trying to intimidate it will only yield in success if the player has a certain stat (for example `strength`) above a certain value.

1.1.5 Documentation

If you're reading this from a PDF file, make sure you're reading the latest version of the documentation. Click Download PDF to get the latest version.

However, we recommend you read this documenation from the website itself, which contains GIFs, higher quality images and better navigation options. PDF should only be used as an offline alternative.

The documentation is structured as follows:

documentation structure

1. The top navigation shows a list of all the available **Modules** with their own documentation.
2. The central page is dedicated to the content of the current page.
3. The left side-bar shows the current page you are reading.
4. The right side-bar shows the table of contents of the current page.

Game Creator 1.x Support

Game Creator 2.0 is not compatible with **Game Creator 1.x** because its code base has been rearchitctured. However, most concepts are identical or very similar.

Each module has one or multiple pages dedicated to the description of what each sub-system does, with clear examples, tips and tricks. Moreover, for those who want to go one step further, all sub-systems have an *Advanced* chapter with more technical details on how it works and how it can be extended through the exposed scripting API.

1.1.6 Errata

If you find a mistake or omission in the documentation, please send us an email at docs@gamecreator.io with a link to the relevant entry and an explanation what you think is wrong. We'll take a look and make any necessary updates.

1.2 Getting Started

1.2.1 Getting Started

Welcome to the Getting Started section. Here you will find all necessary resources to get you started with **Game Creator**.

- **Installation**: Learn how to install Game Creator from the Unity Asset Store.
- **First Steps**: Get to know the basic first steps towards using Game Creator.

Once you are comfortable with the core concepts, we recommend checking the **Examples** that come with Game Creator and the free **Courses** available on the website. If you prefer to learn in non-written format, you can also check our Youtube channel, where we upload new video tutorials.

- **Examples**: Discover examples to learn from and production-ready templates.
- **Courses**: A collection of courses you can take at your own pace.
- **Video Tutorials**: A collection of courses you can take at your own pace.

We also recommend checking out the **Game Creator Hub**: It's a community-driven platform where anyone can download further free Instructions, Conditions and Events.

- **Game Creator Hub**: Explore how the Hub can help you connect with other developers and expand the tools at your disposal.

1.2.2 Installation

This guide explains how to set up your Game Creator project from scratch. It includes information about prerequisites, installing the package, creating an initial workspace and verify your setup.

Creating a new project

Start by downloading the Unity Hub software and install the latest Unity version. Create a new blank project and choose the rendering pipeline that suits you best.

Rendering Pipeline

We recommend using the **Built-in Rendering Pipeline** (BRP) if it's the first time you're using Unity or you just want to try out Game Creator. If you want to use **URP** or **HDRP**, convert the materials automatically clicking on *Edit Rendering Pipelines* *Upgrade Project Materials to URP/HDRP Materials*

Get the **Game Creator** core package from the Unity Asset Store following the link below:

Get Game Creator

Once you have purchased it, click on the "Import" button on the website and the Unity Editor's **Package Manager** window should appear with the **Game Creator** package selected. Click on *Download* and *Import* afterwards.

Package Manager

Let the process complete and if everything went fine, your console shouldn't have any errors. If you do, please feel free to reach out to our support email.

Verify installation

If you have successfully installed Game Creator you should see a new "Game Creator" menu at the top-toolbar with a set of options. You'll also have access to a new "Game Creator" section right clicking on both the *Hierarchy* panel and the *Project* panel.

Setting up for Git

We highly recommend using GitHub or GitLab for backing up your projects. If you use Git as your main repository source be sure to add the following snippet at your `.gitignore` file:

```
# Game Creator
/Assets/Plugins/GameCreator/Documentation.pdf
/Assets/Plugins/GameCreator/Packages
```

This willl avoid adding the offline documentation file to your git repository as well as the examples & code from the Game Creator asset. The reason why the code can be ignored is that it can be easily downloaded from the Asset Store. If you prefer to save a local copy of the current version of your Game Creator package, skip the last two lines and only include the following on your `.gitignore` file:

```
# Game Creator
/Assets/Plugins/GameCreator/Documentation.pdf
```

1.2.3 First Steps

In this section you'll learn to setup a very simple example that uses some of the core features of Game Creator. It shouldn't take you more than 5 minutes to have it up and running.

Preparing the scene

Let's start creating the geometry that will hold the scene. Right click on the *Hierarchy Panel* and select 3D Object   Plane. This is going to be the floor.

If the scene doesn't have a light, create one right clicking again on the *Hierarchy Panel* and select Light   Directional Light and place it somewhere that shines downwards towards the plane.

Finally, if the scene doesn't have a camera object, create one clicking on the *Hierarchy Panel* and select Create   Camera. Select it and, in the upper-part of the *Inspector* window, change its tag from `Untagged` to `MainCamera`. You should also change the camera's position and rotation so it points towards the center of the plane, in order to visualize what happens in it.

Geometry Setup

Creating the Player

To create a player character, open the *Hierarchy Panel* context menu and select Game Creator   Characters   Player. This should have created a character object in the scene in T-pose. If you click play, you should be able to control the default player using the WASD keys or a controller, if you have one plugged in.

Player Setup

Creating a camera

Game Creator uses Camera Shots to tell the main camera how to behave and which target/s to follow. The easiest way to follow the player character is to use the Third-Person camera shot, which automatically orbits around the player using the mouse's movement and allows to zoom in/out.

To create a **Camera Shot** open again the **Hierarchy Panel**'s context menu and select Game Creator   Cameras   Camera Shot.

Automatic camera detection

Creating a new Camera Shot will automatically add the Main Camera component on the scene's main camera, if any at all. If the main camera doesn't have any Camera Shot assigned, it will assign this newly created shot.

The default Camera Shot is the Fixed one. However, we want to use the Third-Person Orbit shot. To change the type of camera shot, click on its name and select **Third Person** from the dropdown menu.

New options should appear now. We need to specify the target at which the camera will look at and orbit around. In both cases, this is the Player, so choose the "Player" option from the `Look Target` and `Orbit Target` fields.

Enter Play-Mode and you should be able to move the player like before, but the camera should also track it and orbit around it using the mouse or controller's right stick.

Complete Setup

Next Steps

Check out Game Creator's free courses for more step-by-step tutorials

1.2.4 Toolbar

Since version 2.3.15, **Game Creator** comes with a dockable **Toolbar** that can be used to create common components in the scene view.

Game Creator Toolbar

Display Toolbar

If the Toolbar is not displayed by default, focus on the scene view and press the [Space] key. This will pop a vertical menu that allows to show/hide different toolbars. Click on **Game Creator** to enable its visibility.

Show Toolbar

The toolbar can be docked as any other toolbar. Simply drag the handles and drop them on any corner or edge.

The orientation can also be changed to fit the position. To do so, right click the handles and select one of the following options:

- **Panel**: Displays an horizontal stripe with the name and icons for each button
- **Horizontal**: Shows an horizontal stripe with just the icons
- **Vertical**: Similar to Horizontal, but displays each button vertically stacked

Tooltips

We recommend using either Horizontal or Vertical layouts. Hovering over any of the icons will display a small tooltip with a description of what that button does.

1.2.5 Examples

Game Creator comes packed with a collection of examples that have been carefully hand-crafted to speed up your development process even further with common mechanics. You can think of them as *templates* of game mechanics you can use for your projects.

To install an example, head to the top toolbar and click **Game Creator** **Install...**. A window will appear with a collection of available examples to install. Select one that you want to add and click *Install*.

Install Window

Dependencies

An example may or may not have a list of dependencies. The **Install** window will display a green icon if the example dependency is installed or a red icon if it is not. Installing a module with dependencies will install and update all dependencies.

Once you do that, the example will appear under `Assets/Plugins/GameCreator/Installs/` or you can simply click the *Select* button to automatically select the example's folder.

Example Path

When installing an example, it is located at the `Plugins/GameCreator/Installs/` directory. The name of the example's folder is the `[name of the module]` followed by a dot, the `[name of the example]` followed by an `@` (at) symbol and the version number. For example, Game Creator's *Example 1* with version 1.2.3 will be located at: `Plugins/GameCreator/Installs/GameCreator.Example1@1.2.3/`.

Uninstalling an Example

If you want to uninstall an example, simply delete root folder of the example. For instance, if you installed a Game Creator example called "Example 1", you can right click the folder at `Assets/Plugins/GameCreator/Installs/GameCreator.Example1@1.0.0/` and choose *Delete*. This will permanently delete the example from your project. However, you can still reinstall it again from the **Install** window.

1.3 Characters

1.3.1 Characters

One of **Game Creator**'s main systems is the **Character**. It represents any interactive playable or non-playable entity and comes packed with a collection of flexible and independent features that can be used to enhance and speed up the development process.

Main Features

A **Character** is defined by a `Character` component that can be attached to any game object. It is organized into multiple collapsable sections, each of which controls a very specific feature of this system.

Some of the most noticeable features are:

- **Player Input:** An input system that allows to change how the Player is controlled at any given moment. Including directional, point & click, tank-controls, and more.
- **Rotation Modes:** Controls how and when the character rotates. For example facing the camera's direction, its movement direction or strafing around a world position.
- **World Navigation:** Manages how the character moves around a scene. It can use a Character Controller, a Navigation Mesh Agent, or plug-in a custom controller.
- **Gestures & States:** An animation system built on top of Unity's Mecanim which simplifies how to play animations on characters.
- **Inverse Kinematics:** An extendable IK system with feet-to-ground alignement or realistic body orientation when looking at points of interest.
- **Footstep Sounds:** A very easy to use foot-step system that mixes different sounds based on the multiple layers of the ground's materials and textures
- **Dynamic Ragdoll:** Without chaving to configure anything, the Ragdoll system allows a character to seamlessly transition to (and from) a ragdoll state.
- **Breathing & Twitching:** Procedural animations that can be tweaked at runtime which change a character's perceived exertion and breathing rate and amount.

Player Character

The Player character uses the same **Character** component as any other non-playable character but with the difference that it has the `Is Player` checkbox enabled. A **Character** with this option enabled processes the user's input based on its Player section.

Shortcut Player

Note that when creating a **Player** game object from the Hierarchy menu or the Game Creator Toolbar, it ticks the **Is Player** checkbox by default.

1.3.2 Component

The **Character** system is built using a single component called `Character` component and handles everything a character can do; From playing animations to footstep sounds, modifying animations though inverse kinematics and much more.

Character Component

General Settings

This block includes the big *mannequin* icon and two fields:

- **Is Player:** Determines whether this character is a Player character or not. A Player character processes input events and makes the character respond accordingly.
- **Update Time:** Indicates whether the character should work with the internal game's clock the real-life clock.

Character Component

Game Time vs Unscaled Time

By default all characters should use the game's clock. Setting the game's time scale to zero will freeze the game, which is useful for pausing it. However if your game has a mechanic where a character ignores the time scale, you can use the unscaled real-life clock.

The *mannequin* icon isn't just an aesthetic ico, but a debugging tool. When the game is running, the icon will change into a green colored one and will turneach of its limbs red every time the character performs a blocking action that prevents that limb from doing something else. For example, performing a jump makes the legs be *busy* for a little less than a second, as well as landing.

The *mannequin* icon will change into a red skull when the character is considered dead.

Kernel Settings

This block is the most important one. A **Character** behavior is divided into 5 main categories (known as **Units**) and each one can be changed individually without affecting the rest.

Names

This settigs block is called the **Kernel** of the character and each individual row is called a **Control Unit** or **Unit** for short.

Character Component

To change each type of **Unit** click on the right-most icon of each and choose the implementation you want. Clicking on the name of the **Unit** will expand/collapse its available options.

Custom Character Controllers

Game Creator comes with a collection of **Units** so you can customize how you want your characters to work. However, these lists are not fixed and can be extended via code. As **Game Creator** grows, so will the amount of options available. If you are a programmer you can create **Unit** that integrates a third-party character system. To know more about extending the `Character` component see the Character Controller section.

PLAYER

The **Player** unit controls how the character is controlled by the user. It only affects the character if its `Is Player` checkbox is enabled. **Game Creator** comes with a bunch of different **Player** units the user can choose from:

- **Directional:** The character moves relative to the main camera's direction and reacting to the keyboard's WASD keys or any Gamepad's Left Stick. This is the most common control scheme for most games.
- **Point & Click:** The character moves towards the point in space click with the mouse cursor. If the Driver is set to Navigation Agent, the character will try to reach the clicked position avoiding any obstacles along its path.
- **Tank:** Pressing the *advance* key will make the character move forward in their local space, regardless of the main camera orientation. This option requires the *Tank* option as its Rotation unit.

MOTION

The **Motion** unit defines a character's properties and what it can or can't do. It comes with a list of options that can be modified both in the editor and at runtime.

Singular Unit

Game Creator comes with just a single **Motion** unit called **Motion Controller**. Unless the character is implementing a custom character controller, the **Motion** unit shouldn't be changed to anything else.

Character Component

These options are:

- **Speed:** The maximum velocity at which the character can move. In Unity units per second.
- **Rotation:** The maximum angular speed at which the character can rotate. In degrees per second.
- **Mass:** The weight of the character. In kilograms.
- **Height:** How tall the character is. In Unity units.
- **Radius:** The amount of space the character occupies around itself. In Unity units.
- **Gravity:** The pull force applied to the character that keeps it grounded.
- **Terminal Velocity:** The maximum speed reached by a character when falling.
- **Use Acceleration:** Determines if the character accelerates/decelerates when moving. If set to false, the character will start moving at full speed.
- **Acceleration:** How fast the character increases its velocity until it reaches its maximum speed.
- **Deceleration:** How fast the character decreases its velocity until it stops.
- **Can Jump:** Determines if the character can execute a jump.
- **Air Jumps:** The number of *double jumps* the character can perform in mid-air. Most games allow zero or up to one air-jump.
- **Jump Force:** The vertical force used when executing a jump.
- **Jump Cooldown:** The minimum amount of time that needs to pass between each successive jump. Useful to prevent the user from spamming jumps.

The **Motion** unit also has the Interaction section at the bottom, which allows to configure how the character can interact with elements from the scene.

DRIVER

The **Driver** unit is responsible for translating the *math* of the processed motion data into actual movement. Depending on the controller type the character will move slightly different.

- **Character Controller:** The default unit. It uses Unity's default Character Controller which provides a versatile controller which should work fine for most cases.
- **Navmesh Agent:** It uses Unity's Navmesh Agent as the character controller. It allows to avoid obstacles when moving a character to a point in space but has the con that prevents the character from being able to jump.
- **Rigidbody:** It uses Unity's Rigidbody component so the character is affected by external forces using Unity¡s Physics Engine.

ROTATION

The **Rotation** handles how the character rotates and its facing direction at any time. There are multiple **Units** available by default although the most common one is the **Pivot**.

- **Pivot:** The character rotates towards the direction it last moved to.
- **Pivot Delayed:** Very similar to **Pivot** but the character waits a few seconds before it starts rotating towards the direction it's moving. This option looks best for slow-paced movements, like walking slowly, sneaking or crawling.
- **Look at Target:** The character always faces towards an object in the scene and wil strafe when moving sideways relative to the object. This option is most used when locking onto enemies.
- **Object Direction:** The character faces the direction of another object. This is mostly used third and first person shooting games where the character must look straight towards where the camera aims so the weapon's direction is aligned with the camera's point of view.
- **Towards Direction:** The character faces a 3D world-space direction. Mostly used in games *on-rails* or *infinite runners*.
- **Tank:** The character pivots around itself when pressing the specified buttons.

Switching at Runtime

It's important to highlight the fact that these options can be changed at runtime. For example, the player can use the **Pivot** unit when wandering the world but switch to a **Look at Target** unit when encountering an enemy. The character will seamlessly transition between them.

ANIMATION

The **Animation** unit controls how the character model moves as a reaction of any internal or external stimulus and also manages the representation of the character's 2D or 3D model.

Character Component

Just like the **Motion** unit, there is one single **Animation** unit option available called **Kinematic** which controls any generic character model's animations. There are different configuration blocks within the **Kinematic** animation unit:

- **Smooth Time:** Determines how long it takes to transition between most character's animations, in seconds. Higher values make transitions look smoother but also take longer and feel less responsive. Lower values closer to zero make the character feel more responsive but also snappier.
- **Animator:** The Animator component of the character's 3D or 2D model.

Runtime Animator Controller

The character's model **Animator** component should use **Game Creator**'s Locomotion *runtime animator controller* or a custom controller that follows the same parameter names. To use a custom *runtime animator controller* it is necessary to implement a custom `IAnimim` unit (see Character Controller for more information).

- **Skeleton:** The skeleton object field is a **Game Creator** asset that defines multiple bounding volumes of each major part of a model. It is primary used to automatically build a Ragdoll system, but can be used for other things, such as detecting head-shots and so on. For more information about how to use and setup a *Skeleton* head to the advanced Skeleton version.
- **Start State:** Optional field that allows to set an initial character State. The starting state is set to layer number -1.
- **Breathing & Twitching:** These two blocks of data allow to make humanoid character models feel more alive, by additively playing subtle animations on top of any others.
- **Breathing:** Allows to control the breathing rate and amount of exertion. The higher the *Rate* the faster and more often the character will breathe. The *Exertion* field controls how deep each breadth is.
- **Twitching:** This is a very subtle animation that is usually not noticed, but perceived. Twitching adds random limb and finger movement to all humanoid character. This allows to have a consistent animation being played between each animation and transition. The *weight* field controls how much of the twitching animation affects the character.

Still pose animations

Combining the **breathing** and **twitching** systems allows using single-frame still poses feel like fully-fledged animations, thanks to the additive *breathing* and *twitching* animations. In fact, **Game Creator**'s default idle poses have a duration of a single frame. It's the twitching and breathing animations that make the pose look like it's real.

Extra Settings

The `Character` component has 3 extra sections at the bottom of the component which allow to control more specific parts of the character.

INVERSE KINEMATICS

Inverse Kinematics (IK for short) allow characters to change their bone rotations in order to transform the overall structure and reach with the tip a targeted position and rotation. A common use of Inverse Kinematics is making sure the character correctly align their feet to the steepness of the terrain.

Character IK

Game Creator allows to dynamically add or remove new IK systems onto each character individually and are processed from top to bottom. To add a new IK system simply click onto the *"Add IK Rig Layer"* button and select the option you want from the list.

Custom IK Rigs

You can also create your own custom IK systems. Check out the Custom IK section for more information.

The `Character` component comes with some common IK systems used on most games:

- **Look at Target:** This IK system allows characters to slightly rotate their head, neck, chest and spine chain in order to look at a specific point of interest. This is specially useful when paired with the Hotspots component. Requires the character model to be *Humanoid*.
- **Align Feet with Ground:** This IK system allows a character to automatically detect when the character is touching the ground and smoothly align their feet with the inclination of the ground. It can also lower the position of the hip so both feet touch the ground, in case the ground is very steep and one foot is higher than the other.

FOOTSTEPS

The Footstep system allows the character to signal when it has performed a step. This is useful when you want a character to leave a trail of footprints, play some particle effects simulating the dust of each step or playing a sound effect.

Character Footsteps

Humanoid and Generic characters

The Footstep system doesn't require the character model to be humanoid. It uses an array of objects that identify the character's feet bones. By default it assumes the character is a human and has two feet, but this can be easily customized clicking on the *"Add Foot"* button.

- The **Ground Threshold** field determines the minimum height a character's foot must elevate in order to consider the movement as a *step*.
- The **Sound Asset** field references a *Footstep Sounds* asset that determines which textures play which sound effects. For more information about how to configure this asset see Footstep Sounds section.

Physically accurate sounds

The *Footstep Sounds* does not play the raw step sound effect but automatically distorts it in order for the player to hear different slightly different sounds each time. It also changes the pitch of the sound if there are multiple layers of textures, muffling those that are less prominent.

RAGDOLL

The `Character` component comes with a built-in Ragdoll physics system that allows to quickly turn any character into an inanimate object that reacts to physics with a set of constraints on each of its limbs.

Character Ragdoll

Skeleton asset

The **Ragdoll** system uses the Skeleton configuration asset to determine which parts of the model correspond to which bone. It can't work without one.

- **Transition Duration:** When a character recovers from a ragdoll state, it plays an animation based on the direction its body faces. This field determines the time it takes to blend between the ragdoll position to the animation clip being played when recovering.

Give plenty of transition time

It is recommended to use large transition values, above 0.5 seconds. The character's limbs can be in very awkward positions that doesn't match the initial pose of the recovery animation clip; so having small transitions will make the character appear to snap into an animation, instead of smoothly blending into it.

- **Recover Face Down:** The recovery animation played when the root of the character's ragdoll faces downwards.
- **Recover Face Up:** The recovery animation played when the root of the character's ragdoll faces upwards.

For more information check its dedicated Ragdoll section.

1.3.3 Interaction

Game Creator comes with a built-in interaction system that lets characters (both Players and NPCs) dynamically focus on a scene element and decide whether to interact with it or not.

Character setup

How a **Character** interacts with scene objects is specified in the **Motion** unit.

Character Interaction

The **Radius** option determines the minimum distance an object has to be in order for the character to focus on it.

The **Mode** option allows to determine how to prioritize how objects are focused:

- **Near Character:** Picks the closes object to the character's interaction center, which can be offset by a certain amount. This option is best for console and games that require a controller.
- **Screen Center:** Interactive objects closer to the center of the screen have higher priority. This is the best option for first person games.
- **Screen Cursor:** Interactive objects closer to the cursor take precendence. This option is best for point and click adventures.

Interact

The character will automatically focus and unfocus any interactive object. To interact with the currently focused object, use the **Interact** instruction.

Interactive Objects

Any game object with the **On Interact** event on a **Trigger** component will be automatically marked as an interactive one.

This event will be fired every time a character attempts to interact with this trigger.

Trigger On Interact

If a character attemps to interact, but there is no *Interactive* object available, it will simply ignore the call.

Detect new Interaction

Apart from the **On Interact** event, one can also detect when a Trigger becomes focused or loses focus (also known as *blur*). This can be tracked using the **On Focus** and **On Blur** events.

Hotspots can also display a text or activate a prefab when the game object is focused by a character. To do so, you can add the **Text on Focus** spot on a **Hotspot** component and it will display the chosen text every time the selected character focuses on this interactive element.

Hotspot Interactive has Focus

1.3.4 Animation

Animation

Game Creator has a built-in custom animation system built on top of Unity's Mecanim that makes it easier and faster to manage character animations.

It introduces the concept of **Gestures** and **States**, which are two mechanisms that allow to play different types of animations without having to previously register them inside an Animator Controller graph.

Mecanim vs Gestures & States

It is preferable that users use the **Gestures** and **States** system to manage and play all their animations. However if a user prefers to use a more traditional approach, there's a base Mecanim layer that allows to use Unity's runtime controller workflow. Check the Animator section to know more about this.

Animation Flow

Gestures are animations that are played once and are removed from the animation graph when finished. For example, an animation of a character throwing a punch can be played as a *Gesture*; This will make a character play the *punch* animation and smoothly restore its previous animation after the animation finishes.

States are animations that are played on a repeating loop. For example, a character sitting on a chair is an *Animation State* while a character moving crouched is a *Locomotion State*.

- **Animation States** play a single animation clip over and over again, until told to stop.
- **Locomotion States** are more complex states that react to certain parameters such as caracter speed. Can have multiple clips transitioning and blending with each other.

Click on Gestures and States to know more about how to use them in your game.

Animator

`Character` components reference a child game object called the *Model* which contains an `Animator` component. This component must referece a Runtime Animator Controller graph, that determines which animations are played when and how these transition between them.

CUSTOM MODEL

Game Creator makes it very easy to change the 2D or 3D model from a character. All that needs to be done is to open the **Animation** section of the `Character` component and drag and drop the Character prefab onto the indicated drop zone.

Change Character Model

Changing model at runtime

To change the character model at runtime use the **Change Model** instruction.

LOCOMOTION RUNTIME ANIMATOR

Game Creator comes with a default **Runtime Animator Controller** called the *Locomotion* controller. It comes packed with a collection of animations and features that fit most projects.

Changing the Locomotion controller

It is not recommended modifying the Locomotion controller. In most cases using a custom State is easier and provides enough flexibility to create new simple or complex locomotion animations.

However if you need to use a custom **Runtime Animator Controller** you must also creata new class that implements the `IAnimim` interface to feed the Character's data onto your custom controller. See Character Controller section for more information.

Gestures

The **Gesture** system allows characters to play a single animation that stops after it finishes. This is specially useful for animations such as a character throwing a punch, vaulting an obstacle or waving a hand.

These animations are always played on top of any other animations.

Character Gesture Waving

PARAMETERS

The easiest way to play a **Gesture** animation is using the Play Gesture instruction, which has a few configuration parameters.

Character Play Gesture instruction

Too many options?

It may seem a bit overwhelming the amount of parameters available for a single animation. Note that the most important ones are the **Character** and **Animation Clip** fields. The rest can be left with their default values and should work on most cases.

Character

The **Character** field determines the object that the animation clip will be played. The game object referenced must contain a `Character` component in order to work. Otherwise the instruction will be skipped.

Animation Clip

The **Animation Clip** references an animation asset. Without this field the instruction will not work.

Avatar Mask

The **Avatar Mask** is an optional field that determines which parts of a character will play the animation and which won't. If this field is left empty the whole body will play the animation. For more information about masking animations, see the Unity documentation about Avatar Masks.

Blend Mode

The **Blend Mode** field determines whether the animation clip overrides or adds up its movement on top of any other animations being played.

- **Blend:** The default parameter. Blend overrides any animations and plays the animation clip on top of them. This is the most common option for most animations.
- **Additive:** This blend mode allows to play an animation by adding up the motion on top of any other clips being played.

Delay

The **Delay** field allows to start playing the animation after a certain amount of seconds have passed. If the value is set to zero the animation will start to play immediatelly.

Speed

The **Speed** field is a coefficient that determines the speed at which the animation is played. A value of 1 plays the animation at its original speed. Higher values will play the animation faster while lower ones will play the animation slower. For example a value of 2 will play the animation twice as fast.

Root Motion

Determines whether this animation should take control over the character and use its root motion to also move and rotate it. Notice that using *root motion* takes control of the character while the animation plays and the user's input will be ignored.

Transitions

The **Transition In** field determines the amount of seconds the animation will take to blend between the current animation and the new Gesture animation clip.

Animation Gesture Transition

Similarly, the **Transition Out** field determines how much time, in seconds, it takes to blend out the current gesture animation to the animation being played underneath.

Wait to Complete

The **Wait to Complete** checkbox allows the instruction to be put on hold and only continue once the animation finishes. This is specially useful when chaining multiple gestures one after another.

About Instructions

For more information about how to use instructions to interact with other systems, see the Visual Scripting section.

States

The **States** system allows to dynamically blend in/out arbitrary animations or entire animator controllers at runtime. All that needs to be done is to specify which animation or controller a character should play, and which layer should it be assigned to.

Character Animation State Asset

Mecanim vs States

It is important to note that the **States** system is built on top of Unity's Mecanim and it complements it; It does not prevent or restrict from using any of its features. It simply adds a new and more flexible workflow on top of it.

TYPES OF STATES

There are primarily two types of States, but both work the same way: An instruction feeds a State to a Character and this one plays the animation/s based on the behavior of the State.

Animation States

Animation States are single animation clips that are played over and over again, until told to stop and blend out.

For example a character playing a single looped animation of sitting on a chair is an *Animation State*. These are the most common and basic forms of **States**, where an Animation Clip must be provided and the Character plays it in a loop.

It is also possible to create an *Animation State* asset that allows to play a looped animation as well as providing a fields for gestures that are played when entering and exiting the State. To do so, right click on the Project Panel and select *Create* *Game Creator* *Characters* *Animation State* and drop the Animation Clip file onto the corresponding field.

Character Animation State Asset

The **State Clip** field determines which animation is played in a loop, while **State Mask** discerns which body parts are affected by the animation. Note that this last field only works with Humanoid characters. See Avatar Mask for for information about masking animations.

The **Entry** and **Exit** sections contain optional fields that allow to play a Gesture right before entering or exiting the current State. For example, you may want a character to play the *unsheathe sword* animation every time it enters a sword combat stance, and play the *sheathe* animation when exiting the combat stance state.

Locomotion States

These are more complex States that react to certain parameters such as the speed of a character, its direction and fall velocity. Locomotion States have multiple clips transitioning and blending with each other.

For example a character that idles in a prone position and crawls when the character moves is a *Locomotion State*.

To create a Locomotion State, right click anywhere on the *Project Panel* and select *Create* *Game Creator* *Characters* *Locomotion Basic State* or *Create* *Game Creator* *Characters* *Locomotion Complete State*.

Character Locomotion State Asset

The **Locomotion State** asset may seem a bit daunting at first, but it's fairly straight forward. There are two types of **Locomotion States** and those are:

- **Basic States:** Have an idle and an 8-axis directional animation clip fields for moving
- **Complete States:** Have an idle and a 16-axis directional animation clip fields for moving: 8 for moving at half speed and another 8 for moving at full speed.

The first fields, **Airborne Mode**, controls the amount of animation clips available and can take one of the following values:

- **Single**: Displays a single animation clip for that particular phase.
- **Circular 8 Point**: Displays animation clip fields for the 8 cardinal directions: Forward, Backwards, Right, Left and each of the diagonals.
- **Circular 16 Points**: Displays animation clip fields for the 8 cardinal directions, and another 8 for half-way points between the first and the origin.

8 Points vs 16 Points

This decision comes down to the type of controller and animations available. If your game is meant to have analogic controls, the user might slightly push the movement joystick forward, making the character move slow. In this case, it is recommended using the **Complete Locomotion State**, as it allows to have both running and walking animations in a single State.

LAYERS

The **States** system is built around the concept of *Layers*, which is similar to the concept found in image editing tools, such as *Photoshop*. The idea is that any **State** is assigned a layer number. With higher numbers taking higher priority when playing an animation.

Example

Let's say we have a character with three Layers, each one with a single **State**, numbered **1**, **2** and **10** respectively.

Character States Layer

In this case, the animation played would be be the one found at the layer number **10**. However, if this layer was to be removed, the animation at layer **2** would be the next one with highest priority and thus, its **State** would be played.

It is recommended to add a transition time when adding or removing a **State** from a *Layer* in order to smoothly blend between the new animation and the one underneath.

Animation Gesture Transition

When adding a new **State** onto a *Layer* that already has a **State**, this last one will be smoothly faded out taking into account the new **State**'s transition time, until it is replaced by the new one. After that happens, it will be automatically disposed.

Gestures and States

Note that although **States** can have different priorities, a **Gesture** animation will always have higher priority than any **State** and will play on top of it.

WEIGHTS

Setting a new State is not an all-or-nothing operation and the new animation can be blended by a percetage with any other animations playing underneath the stack.

For example, if a character is currently playing a *running upstraight* animation, a *running crouched* animation can be blended at 50% to to make the character look like it's running halfway between standing and crouched.

Weight at runtime

The weight can be modified at runtime using the Change State Weight instruction.

ENTERING A STATE

The easiest way to make a character enter an Animation or Locomotion State is using the Enter State Instruction.

Instruction Enter State

The **Character** field references the targeted character game object that enters the state. The **State Type** field determines whether the State is an *Animation Clip*, a *State* asset or a Runtime Animation Controller.

Runtime Animation Controller as a State

Game Creator allows to use a Runtime Animation Controller as a State. However, this is an advanced feature and should only be used if one understands how Gestures & States work under the hood.

The **Layer** field allows to determine which layer this State occupies in the Character's layer stack. **Blend Mode** by default is set to *Blend*, which overrides the underlying animation with the animations provided by the State. If set to *Additive* it adds up the new State's animation as a delta movement on top of any other animation being played.

The **Delay** field allows to delay in a few seconds the time to start playing the State. **Speed** is a coefficient value that determines how fast the State plays. For example, a value of 1 makes the State play its animation at its default speed. A value of 0.5 plays the animation at half speed and a value of 2 plays it twice as fast.

The **Weight** field determines the opacity of the State. A value of 1 plays the animation as it is. Lower values allow any previous animations to bleed through and mix the effect between the new State and any other animation being played in lower layers.

The **Transition** field is the time in seconds that the new State takes to fade in.

EXITING A STATE

The instruction Stop State can be used to smoothly stop playing a State on a character.

Instruction Enter State

The **Character** field determines the targeted game object that stops playing a State found at the layer identified by the **Layer** number field.

Similarly, the **Delay** and **Transition** fields allow to delay the fading of the State by a certain amount of seconds.

1.3.5 Inverse Kinematics

Inverse Kinematics

Inverse Kinematics (IK for short) is the process of calculating the rotation of bones from a chain of bones, in order for the leading one to reach a desired position. **Game Creator** makes use of both limbic and full-body IK.

Character Feet IK

A common case scenario is adjusting the bending of the knees so the character naturally plants its feet on the ground.

MANAGE IK RIGS

The `Character` component has a section at the bottom that allows to manage which rigs affect the character and change their properties.

Character Feet IK

Rig order matters

The IK Rigs are excuted from top to bottom. So if two IK systems affect the same bone chains, the last rig will override any previous ones.

To add a new Rig, click on the *Add IK Rig* button and choose one from the dropdown list.

RIGS

Game Creator comes with a few IK rigs that work out of the box:

- **Feet Align**: Allows to align a Character's feet to uneven terrain.
- **Look at Target**: Allows a Character to use the Look At system from Hotspots.

Feet Align

This **IK Rig** allows a character to plant their feet and adjust the rotation on uneven terrain. This rig also allows the hips to be lowered by a certain amount if the height difference between both feet is very large.

Character Feet IK

Only for Humanoids

The **Feet Align** rig only works with *Humanoid* characters.

Character Feet Align

The **Feet Align** rig has the follow options:

- **Foot Offset**: An optional vertical offset applied to each foot. This is useful in cases where the foot penetrates the ground or floats above it, due to differences between the bone's tip position and skin mesh bounds.
- **Foot Mask**: Allows to choose which Layers should the character consider when aligning with ground. For example, water typically has a collider component, but the character should not align its feet on its surface.
- **Align Hips**: If the character is perpendicularly aligned on a very steep ramp, oe feet will be much higher than the other, making the lower leg float above ground. Ticking this option allows the character's center to be lowered so both feet touch the ground.

Look at Target

The **Look at Target** rig allows a character to rotate their head, neck, chest and body in order to look at a Hotspot.

Character Feet IK

Only for Humanoids

The **Look at Target** rig only works with *Humanoid* characters.

Character Look at Target

The **Look at Target** rig has the follow options:

- **Track Speed**: The angular speed at which each bone rotates to track the target. In degrees per second.
- **Max Angle**: The maximum peripheral angle, in degrees.
- **Head Weight**: The contribution of the head to the total rotation.
- **Neck Weight**: The contribution of the neck to the total rotation.
- **Chest Weight**: The contribution of the chest to the total rotation. Note that the Chest is an optional bone and some models may not have it.
- **Spine Weight**: The contribution of the spine bone to the total rotation.

Default values

The default parameters have been carefully picked to work for the majority of human-like characters.

1.3.6 Footstep Sounds

Game Creator's characters can mix and play multiple sound effects depending on the type of ground it's stepping on.

Character Footstep Sounds

Humanoid and Non-Humanoid

This system works for humanoid and non-humanoid characters alike. Though humanoids don't require any kind of setup and work out of the box.

Detecting Steps

The Footstep System detects when any identified bone passes through an horizontal threshold called *Ground Threshold*. When this occurs, the Footstep System raises a signal informing that a step has been taken.

Character Foot Ground Threshold

Playing Footstep Sounds

The **Footstep Sounds** system comes with a built-in tool for playing different sounds and sound variations depending on the surface the character is stepping onto. To create a material sound library, right click on the *Project Panel* and select `Create - Game Creator - Common - Material Sounds`.

Character Footstep Reaction

The **Material Sounds** asset allows to define which textures produce which sound effects. Each texture can have multiple sound effects, which will be picked up randomly every time the character takes a step.

Pseudo-Random Sound Picking

Note that although it's completely random, two sound effects will never be played in succession in order to avoid repetition.

The **Material Sounds** asset also allows to instantiate a game object from a pool of prefabs at the impact position. The instantiated object is aligned with the incision angle. This is very useful when spawning particle effects of dust.

The human hearing quickly recognizes sound patterns. To avoid hearing the same sound effects over and over again, the Footstep Sound System intelligently shifts the pitch and speed of each audio clip every time it's played. By doing so, a single clip can be played hundreds of times with various nuances that tricks the human hearing into perceiving each clip as a different sound effect.

Gradient Footstep Sounds

Floors are not always composed of discreet materials. For example, there might be a sound effect for when the player steps on shallow water and another one when steps on sand. However, if the character runs along the shore, where there's a blend between the water and sand textures, the resulting sound effect is a proportional mix between the two audio clips and their pitch is shifted to fit how real-life audio blending occur.

Drop the **Material Sounds** asset onto the **Character**'s `Sound Asset` to link them.

Reacting to Footsteps

The **Footstep** system also allows Characters to react every time a step is taken. Using the **On Step** Trigger, which is executed every time a defined Character takes a step. This is useful for things like leaving footsteps behind.

Character Footstep Reaction

1.3.7 Ragdoll

A *Ragdoll* system lets characters react to physics and external forces without any direct input from itself. This is commonly used for enemies that have been defeated or when the player falls unconscious due to a strong attack or a big fall.

Character Ragdoll

A **Character** requires a Skeleton definition asset in order to correctly identify the size of each of its bones and how they form the joint connection chain.

Quickly generate a Skeleton

Defining all **Skeleton** volumes and how these relate to their parent bones is tedious and time consuming process. Luckily **Game Creator** makes it very easy to automatically generate a humanoid Ragdoll asset. With the Skeleton asset selected, drag and drop any *Humanoid* 3D model onto the bottom drop-zone and it will generate the structure for you. You can then tweak the values to perfectly match your model.

Starting and Stopping

To initiate a ragdoll state, simply use the *Instruction* **Start Ragdoll** and select the targeted character. Notice that the player's input will still be in effect though. This is why Game Creator's default character comes with 2 Triggers that make it even easier to handle Ragdolls: When a character is considered to be *dead* it will automatically trigger the *Start Ragdoll* instruction on the character. When a character is revived, it will also automatically handle playing the correct animation and get the character up from the floor.

This means that, in order to start and stop the ragdoll effects, all that needs to be done is to use the *Instruction* **Kill Character** to disable any interactions from a character and it will automatically enter ragdoll-mode. On the other hand, using the **Revive Character** *Instruction* will give back control to the character and get it up from the floor using the correct animation.

Getting up

The character will automatically handle transitioning from its ragdoll pose to the default idle animation and pick up the most suitable gesture, depending on whether its currently facing down or up.

Configure Ragdoll Animations

To setup the *getting up* animations, select the Character and drag and drop the desired animations onto the **Recover Face Down** and **Recover Face Up** clip fields.

Ragdoll Animations Setup

The **Transition Duration** field allows to specify the duration between the time the character is not controllable due to being in ragdoll-mode and recovered. Ideally this value will be a few milliseconds shorter than both recover animations.

The most important part of a ragdoll is knowing the length and size of each of its physical bones and how they interact with the rest of the body. This is done using the Skeleton asset file. To know more about configuring a Skeleton asset and associate it with a Character, see the Skeleton section.

1.3.8 Markers

A **Marker** is a component that is used by **Characters** as destination points. It allows to define a target position and rotation so the **Character** is at the correct location before doing something else, like opening a door.

Marker Gizmo in Scene

A **Marker** has a yellow shaped arrow that indicates the direction the Character will face after moving towards it.

Marker Gizmo in Scene

Optionally, a Marker can specify a **Stop Distance** threshold from which a Character is considered to have reached its destination.

By default it's zero, but if the destination is a very crowded, there might not be enough space for a character to be at the exact marker's position. Having some error threshold allows Characters to *more or less* reach their destination without getting stuck or pushing other characters around.

The **Type** field allows to determine how the Marker works. By default its set to *Directional* which forces the character to end at the same position and rotation as the arror-shaped gizmo in the scene.

Another available mode is *Inwards* which tells the character to move to the closest point around a circle and rotate towards its center. This is specially useful when you want the character to pick up an item and you don't care from which angle it is picked up.

1.3.9 Advanced

Advanced

This section covers topics that require some degree of programming knowledge and assumes certain level of coding expertise.

- **Skeleton**: What a Skeleton volume asset is and how to configure one.
- **Character API**: How to interact with the default **Character** system.
- **Character Controller**: How to customize or integrate other character controllers with the default one.
- **Custom IK**: How to construct new inverse kinematic character rigs.

Skeleton

A **Skeleton** asset is a scriptable object asset that contains all the necessary information to identify the bounding volume of a character's bones and how these form a chain of joints that conforms the whole body.

What is it used for?

The **Skeleton** asset is used on multiple systems, such as the Ragdoll system, or the Melee and Shooter hit detection systems.

Link Skeleton to Character

CREATE A SKELETON

To Create a Skeleton asset, right click on the *Project Panel* and select **Create** **Game Creator** **Characters** **.Skeleton**

To assign a **Skeleton** asset to a **Character** simply select the desired **Character** and expand the *Animation* tab. Drag and drop the **Skeleton** asset onto its corresponding field.

Link Skeleton to Character

CONFIGURE SKELETON

The **Skeleton** asset is divided in to sections: The first one determines the Physical Material and collision detection mode of the rigidbody system stemmed from the volumes. The second one defines the properties of each of the character's volumetric bones.

Readme!

To more easily configure the volumetric bounds of a humanoid character, see the next section.

Configure Skeleton

To create a volumetric bone, click on *Add Volume* and select the type of bone to create:

- **Box:** A cubic volume. Mostly used for chest and flat surfaces.
- **Sphere:** A spherical volume. Used for hands and head mostly.
- **Capsule:** The most widely used volume bone. Used for most limbs.

Configure Skeleton

A *Volumetric Bone* is composed of a **Bone Type**, a volume definition and an optional **Joint**.

The bone type can be specified by setting the humanoid bone from a dropdown list or from a path. For example, to reference the front right foot of a model of a Dog, the bone could be `Root/Spine/Collar/Right_Leg/Right_Foot` .

The volume definition depends on the type of volume created. For example, a Sphere volume bone contains a radius and a position offset field.

The Joint field allows to determine how a bone is related to other bones via a joint system. For example, a human right arm might be connected to the character's shoulder bone using a Fixed Character Joint, that allows it to rotate a certain amount of degrees.

More on Joints

For more information about character joints, visit this Unity documentation link.

SETTING UP A HUMANOID SKELETON

Game Creator comes with a tool that makes it much easier to automatically *guess* and extract the bounding volumes of a humanoid model. To use it, simply drag and drop the 3D model from the Project or Scene view model onto the bottom-most window and it will auto-magically approximate a Skeleton for you. you can then go ahead and tweak it to your game needs.

Configure Skeleton

Character API

This section covers the inners of the `Character` component and which tools are exposed for programmers to use.

LOCOMOTION

To move a character to a certain location, you'll need to access the `IMotion` unit, which handles the response to locomotion signals. To know more about Kernel Units, visit the Character Controller page.

There are 3 movement types that a character can perform:

- **Move to a position:** Which is done using the `MoveToLocation(...)` method.
- **Move towards a direction:** Which is done executing the `MoveToDirection(...)` method.
- **Start/Stop following a target:** Which is done using the `StartFollowingTarget(...)` and `StopFollowingTarget(...)`.

For example, to force a character to move to a target's transform position, the following snipped should be used:

```
Location location = new Location(target.position);
character.Motion.MoveToLocation(location, 0f, null);
```

RAGDOLL

As long as the character has a Skeleton, a ragdoll state can be triggered. To make a character enter the *ragdoll* state use the `character.Ragdoll.StartRagdoll()` method. To recover from a *ragdoll* state, execute the `character.Ragdoll.StartRecover()` method.

Ragdoll and Death

We recommend setting the character as dead before entering the ragdoll state. Otherwise the ragdoll animation might want to perform actions only available to non-Ragdoll characters (such as running, shooting, jumping, ...).

ANIMATIONS

To play an animation **Gesture** you can access the `Gestures` property and trigger the `CrossFade(...)` method, which handles creating a new layer (if necessary) on top of Unity's Mecanim and play the desired animation.

To enter or exit an animation **State** you can access the `SetState(...)` and `Stop(...)` methods from the `State` property.

Note that all animation methods are *async*. This means that your code can yield until the animation has finished executing. For example, to play a gesture animation and print a console message right after the animation has finished you can use:

```
Debug.Log("Start playing a new animation gesture")
await character.Gestures.CrossFade(myAnimationClip, ...);
Debug.Log("The previous animation has finished")
```

CHANGE MODEL

To change a character model, call the `ChangeModel(...)` method. Its signature contains 2 parameters:

- A prefab object reference, which should be the FBX model
- A configuration struct of type `ChangeOptions`

This last optional parameter allows to define the new model's footstep sounds, its skeleton's bounding volumes as well as a new animator controller and an offset. For example, to change the player's model without any optional parameters:

```
GameObject instance = character.ChangeModel(prefab, default);
```

BUSY

Accessed from the `Busy` property, it allows to query whether a specific limb of the character is being used or not. This allows other systems to determine whether an action can be performed or not.

Busy and Available limbs

For example, if a character has both of its arms set as unavailable, trying to execute an action that involves the hands won't be possible, such as grabbing a ladder.

The follow properties can be queried and inform of the availability state of the limb or group of limbs:

```
IsArmLeftBusy : boolean
IsArmRightBusy : boolean

IsLegLeftBusy : boolean
IsLegRightBusy : boolean

AreArmsBusy : boolean
AreLegsBusy : boolean

IsBusy : boolean
```

Additionally, limbs can be marked as busy or make them available using the `MakeLimbXXX()` method, where XXX is the limb of the body. For example, to set the *Left Leg* as busy, call the `MakeLegLeftBusy()` method.

All available methods

For more information about all the available methods on the Busy system, check the script under `Plugins/GameCreator/Packages/Core/Runtime/Characters/Busy`.

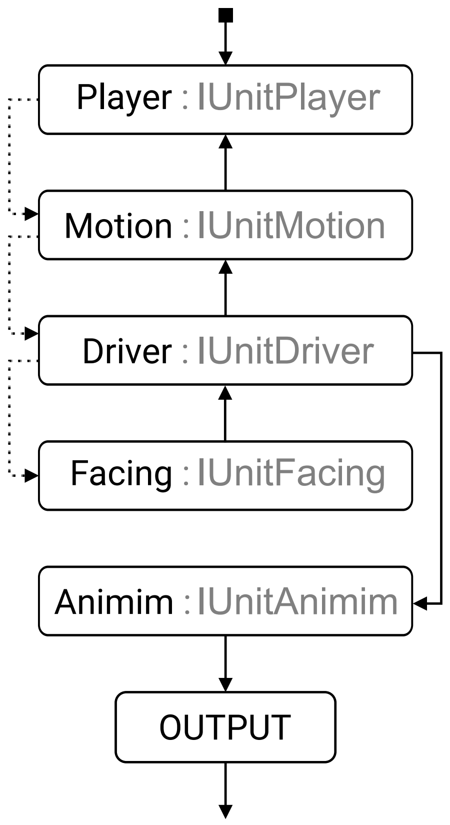
Character Controller

Game Creator Characters have been build to be easy to use and highly customizable. This section go over what a **Character** does every frame cycle. This will put you in perspective in order to create a custom Character that works with Game Creator or you want to integrate a Character system from another package into Game Creator.

The **Character** component is composed of 5 different **Units** which conform the **Kernel**. These units can be changed at runtime without affecting the rest:

- **Player:** Defines whether the Character is a playable one and how the user can interact with it. If you want to create a custom Character input system, you'll need to implements the `IUnitPlayer` interface.
- **Motion:** Acts as an interface between the scene and the Character. All movement commands are relayed through this system and also takes into account the *Player*'s information. It decides which locomotion system should be used. If you want to create a different motion system for your characters, create a class that implements the `IUnitMotion` interface.
- **Driver:** Manages how the Character moves around the scene based on the *Motion*'s input. If you want to integrate another Character system from another Asset Store package, create a new class that implements from `IUnitDriver`.
- **Facing:** Is responsible for rotating the character towards a desired direction. For example, the default behavior is to have the character look towards where it's moving. If you want to customize where the character faces, create a custom class that implements the `IUnitFacing` interface.
- **Animim:** This system takes the *Driver*'s input and tells the Animator component which animation should be played via Mecanim parameters. If you want to use a custom Animator for your Character, crete a class that implements `IUnitAnimim` interface.

Every new cycle tick the **Character** updates all these systems in a very specific order.



It starts by calling the **Player**'s system `Update()` method. This takes the user's input and calls one of the **Motion**'s public movement methods:

- `MoveToDirection()`
- `MoveToPosition()`

After the **Player**'s system has been processed, the **Character** calls the **Motion** system's `Update()` method. This is where external forces are calculated, such as gravity, sliding through slopes, dashing, jumping, ...

Communication between systems

The **Motion** system takes into account the **Player**'s system before running the update. A system can access any of the other's systems data before processing its `Update()` cycle.

After the final **Motion** movement is calculated, the **Character** executes the **Driver**'s `Update()` method. This is where the *Transform* component is updated based on the movement type provided by the **Motion** parameter.

After the **Driver** system is completed, the **Facing** system starts. Based on the information provided by the **Driver** and **Motion** systems it calculates the direction in which the Character should be facing at.

Finally, the **Character** system calls the **Animim**'s `Update()` method, which feeds the **Animator** component with the necessary parameter values based on the information of the rest of the systems.

Modular design

It is important to highlight the fact that each system is independent of the other. You can create a custom animation system by implementing a `IUnitAnimim` interface and still use the default **Player**, **Motion** and **Driver** systems.

PLAYER

The **Player** unit handles how the user interacts with the Player character. If the Character does not have the `Is Player` field checked, this unit is skipped entirely.

The Player also contains the `IsControllable` flag that defines whether a character processes the input received or not. This is very useful when a character is in the middle of a cutscene and you don't want the user to have control over the player.

MOTION

The **Motion** unit is the brain of the character. It contains all of its quirks, such as its height, its move speed, terminal velocity and so.

The **Motion** unit also is in charge of receiving any locomotion commands:

- `MoveToDirection` defines a direction towards where the character must go. This method has to be called every frame or the character will stop.
- `StopToDirection` stops the character's movement. Useful when the character moves due to its deceleration value.

A character can also be instructed to move to a certain position:

- `MoveToLocation` instructs a character to move to a specific location. The `Location` class accepts a position and/or a rotation.
- `MoveToTransform` instructs the character to move to a specific transform's position. If the transform changes its position, the character will follow it until it reaches the target.
- `MoveToMarker` is similar to the previous method, but also takes into account the marker's rotation and forces the character to end facing the same direction as the navigation marker.

A character can also follow another target without an end condition:

- `StartFollowingTarget` starts following a target and stays within a `minRadius` and `maxRadius` distance.
- `StopFollowingTarget` instructs a character to stop following a target.

The **Motion** unit is also responsible for dealing with character's jumps. The `Jump()` method will instruct a character to perform a jump (or air jump), if it's possible.

DRIVER

The **Driver** unit controls *how* a character moves around the scene: Whether it's using Unity's Character Controller, the Navigation Mesh Agent for obstacle avoidance or a physics-based rigidbody entity.

This unit recieves the locomotion information of *Motion* and *Facing*, and transforms it into a physical translation and rotation.

FACING

The **Facing** unit controls where the body of the character (not the head) points at. By default all characters do not rotate their body unless they are moving; in which case the body rotates towards where the character is moving.

However, there are certain situations where the character might want to temporary face at a certain direction. For example, when the character aims with the gun at a certain object, or when talking to a character. **Game Creator** comes with a layer system that provides a neat solution for these cases.

Recommendation

If you plan on creating your own facing system, we recommend creating a class that inherits from `TUnitFacing` instead of the interface `IUnitFacing`. This base class comes with the layer system built out of the box, so you don't have to recode it.

The **Facing** system interfaces provides access to 3 methods:

- `int SetLayerDirection(int key, Vector3 direction, bool autoDestroyOnReach)`
- `int SetLayerTarget(int key, Transform target)`
- `void DeleteLayer(int key)`

The first two methods, `SetPlayerDirection` and `SetLayerTarget` allow to make the character look at a certain direction or keep track of a particular scene object. Making the character change its default direction is done using a layer system.

When any of these methods is called for the first time, it creates a new entry in the layer system and returns its identifier: an integer known as `key`. To subsequently update a particular layer, simply pass as the `key` argument the resulting key from the previous iteration.

For example, if you want to make a character look at a certain character (defined by the variable `lookAtTransform`), you'll simply need to call:

```
private int key = -1;
public Character character;
public Transform lookAtTransform;

public void StartFacing()
{
    IUnitFacing face = this.character.Facing.Current;
    this.key = face.SetLayerTarget(this.key, this.lookAtTransform, false);
}

public void StopFacing()
{
    IUnitFacing face = this.character.Facing.Current;
    face.DeleteLayer(this.key);
}
```

No Exceptions

It is important to note that the layer system won't throw any exceptions. If you try to attempt to delete a layer but the key doesn't exist, it will simply do nothing.

When calling the `StartFacing()` method, the character will smoothly rotate towards the target defined until the `StopFacing()` method is called.

However, in some cases, you may not want to manually remove the facing layer, but instead stop facing a particular direction when the character reaches its target direction. For these cases, simply set the `SetLayerDirection` method's last parameter to `true`. This will tell Game Creator to automatically remove the layer when the character reaches its target direction.

For example:

```
public Character character;
public Vector3 direction;

public void LookAt()
{
    IUnitFacing face = this.character.Facing.Current;
    face.SetLayerDirection(-1, this.direction, true);
}
```

ANIMIM

The **Animim** unit handles everything related to the visual representation of a character: From its appearance to its animations.

Animator required

This unit requires an Animator component reference in order to deal with animations

The default character system comes with a set of procedural animations played on top that add subtle but consistent movement across different animations, such as breathing and exertion. The breathing rate and exertion amount can be modified using the `HeartRate`, `Exertion` and `Twitching` proprerties.

Custom IK

Characters in **Game Creator** have a layered *Inverse Kinematic* system that can be stack one after another in order to modify the animation of a character. The most common form of inverse kinematics is the Feet IK, which makes sure a character's feet are correctly placed and aligned with the floor below it.

ACCESSING A RIG

Accessing a rig is done using the IK property of the Character's component. To deactivate the rig that aligns the feet on the ground, for example, can be done using:

```
character.IK.GetRig<RigFeetPlant>().IsActive = false;
```

Note that `character.IK.GetRig<RigFeetPlant>()` returns an instance of that particular rig (null if it can't be found).

CREATING A CUSTOM RIG

Game Creator offsers two types of IK system wrappers:

- Riggings powered by DOTS
- Riggings powered by the `AnimatorIK` method

To create a new IK system you must crete a class that inherits from either `TRigAnimationRigging` (for DOTS) or `TRigAnimatorIK` (for AnimatorIK). We recommend using the new DOTS-based approach when possible, as it's more performant.

In either case, you should override the `DoStartup(...)` and `DoUpdate(...)` methods, which are called once at the beginning and every frame respectively.

```
public class MyCustomRig : TRigAnimationRigging
{
    protected override bool DoStartup(Character character)
    { }

    protected override bool DoEnable(Character character)
    { }

    protected override bool DoDisable(Character character)
    { }

    protected override bool DoUpdate(Character character)
    { }
}
```

1.4 Cameras

1.4.1 Cameras

Cameras are devices that capture and display the world to the user. **Game Creator** uses two components to determine how the action is framed:

- **Camera Controllers:** A component attached to the camera. For itself it does nothing but mimic the behavior that its active camera shot feeds. By default, the `Main Camera` component is the primary camera controller.
- **Camera Shot:** A component that has multiple configurations, depending on which, its associated camera controller will respond in one way or another.

For example, if the camera controller `Main Camera` has the *Third Person* Shot associated with it, the main camera will mimic the behavior of that shot, which is to follow and look at a target, while the user can orbit around it.

A camera controller can transition to another camera shot. This transition can either happen over time, or instantly.

1.4.2 Camera Controller

A **Camera Controller** is a component attached to a camera object that has a associated at most one **Camera Shot** reference. This associated camera shot can be changed at runtime and will dictate the behavior of the camera controller.

Main Camera

Most games will only have one single camera. The camera in these cases will have the `Main Camera` component attached, which is a camera controller that can be accessed globally by any script.

Creating the Main Camera

To creata a main camera, right click on the *Hierarchy Panel* and select Game Creator Cameras Main Camera from the dropdown menu.

Main Camera

The **Main Camera** component has three distinct sections:

- **Game Time:** Defines the time mode used to update the camera. By default it uses the Game Time option, which can pause time when the time scale is set to zero.
- **Shot:** Determines the **Camera Shot** associated with this camera controller. If none is set, the camera won't have any behavior.
- **Avoid Clipping:** Allows the camera to avoid clipping through the geometry of the scene.

Smooth Camera Movement

The Shot's smoothing options determine how much the camera lags from the Shot's behavior. It's recommended to add some lag to avoid any jittering. However, introducing too much lag will make controls feel a a bit unresponsive.

Transition to a new Shot

To transition a Camera Controller from one Camera Shot to another one, it's recommended to use the **Change Shot** instruction.

Change Camera Shot instruction

Simply drop in the Camera Shot you want the Camera Controller mimic and how long should it take to transition. **Game Creator** will handle the rest.

1.4.3 Camera Shots

Camera Shots are components that provide the **Camera Controller** (or **Main Camera**) information about how they should move and behave.

Camera Shots Analogy

Think of Shots as a collecion of camera angles scattered around the scene, each trying to frame the action as best as possible. Then you, the Director, decide which camera is visualized on the screen, for how long and when to swap to another shot.

Creating a Camera Shot

To create a **Camera Shot** right click on the *Hierarchy* panel and select Game Creator Cameras Shot Camera from the dropdown menu. This will place a new game object on the scene with the **Camera Shot** attached to it.

Camera Shot + Main Camera

If your scene doesn't have a Main Camera attached to the scene camera, creating a new **Camera Shot** will create one for you and link it to the newly created shot automatically for you.

Camera Shot

A **Camera Shot** component contains its shot type and a collection of parameters that can be modified to fine-tune its behavior. In the example above, the *Third Person* camera shot has 3 sections that allow to modify the target tracked, whether the user should be able to zoom in/out and how the orbit should be done. Clicking on each of these sections reveals or hides its content.

Camera Shot Types

To change a camera shot type, simply click on its type name. A dropdown menu will appear from which the new type can be selected.

Camera Shot

FIXED POSITION

This camera shot doesn't move from its place. However, it can be instructed to keep track of a target's position by pivoting around itself. Think of this camera's behavior as a security camera.

FOLLOW TARGET

This camera is very similar to the *Fixed Position* but also allows to follow the target from a certain distance. Useful for top-down view games like Diablo.

FOLLOW TRACK

This camera shot allows to track a target as well as move along a pre-defined rail-like path. This path's position is defined by the position of the targeted object along another path. This camera shot is useful for games that have very linear corridors but want to smoothly turn the camera around corners.

ANIMATION

This camera shot moves along a pre-defined path over a certain amount of time. When it reaches the end of the animation, it stops there and does nothing else. This shot is very useful for cinematic sequences where multiple animation shots can be chained together to dynamically follow the action.

FIRST PERSON

This shot is perfect for first person games. The target object (usually a humanoid) determines the position of the shot and follows it while allowing to spin the head around.

Comes with a vast collection of features such as:

- **Head Bobbing:** The amount of up and down and side movement due to the character's change of weight when walking or running.
- **Head Leaning:** A subtle rotation on the local X and Z axis that is applied when the character moves in order to display the impulse required to go towards that particular direction.
- **Noise:** Another subtle yet realistic random movement applied to both the rotation and translation of the shot to simulate restless idle motion and breathing.

All these parameters can be changed at runtime to accomodate to different situations, such as increasing the noise after sprinting and such.

THIRD PERSON

This shot is used on third person games where the camera follows a target but the user is free to orbit around it.

LOCK ON

This shot allows to follow a target's position while the rotation follows another one, always framing both targets on screen. This shot is perfect for locking on enemies when making an action game or hinting the player something they should not be missing.

ANCHOR PEEK

This shot anchors itself to the chosen game object and allows to pan and tilt the camera vertically and horizontally, up to a certain amount. The *restitute* field brings back the shot to the center if no further input is detected. This is specially useful when using a gamepad controller and you want the character to peek around corners.

1.5 Visual Scripting

1.5.1 Visual Scripting

Game Creator comes with a unique high-level and intuitive visual scripting toolset that makes it very easy to *code* interactions. It only consists of 3 components:

- **Actions**: A list of instructions that are executed one after another.
- **Triggers**: A component that listens to events in the scene
- **Conditions**: Branch off to instructions, depending on certain conditions.

Visual Scripting nomenclature

The **Actions** component consists of a list of **Instructions**. The **Conditions** component is made of **Branches**, which contain a list of **Conditions** and **Instructions**. Lastly, the **Trigger** component listens for a specific **Event** in the scene.

Apart from these three visual scripting components, **Game Creator** also includes **Hotspots**, which is a special type of component that doesn't directly affect gameplay, but highlights interactive objects in different ways: For example, making a character's head turn towards a point when near, showing a text above an interactive element, and so on.

High Level Scripting

A high-level scripting language is a methodology in which programming interactions is closer to what humans are used to use. For example, in **Game Creator** you can tell a character to follow a target object; freeing the user from having to think what it means to *follow* an object.

Game Creator Hub

Game Creator and each module comes packed with a unique set visual scripting tools. The Game Creator Hub is a web platform where community members upload free Instructions, Conditions and Events for everyone to download and use in their projects. Be sure to check it out!

Why not Playmaker

Why not both? Playmaker and Unity's Visual Scripting solution are graph-based, which tend to be closer to a programming language. If you're used to using these, you'll find these complement **Game Creator** very well.

On one hand, **Game Creator** makes it very fast and easy to structure common interactions without the need to *code* the low-level stuff. However, if you need more fine-grain control over some parts and you don't know how to code your own Instructions, you can use these graph-based solutions that perfectly complement the process of making games.

1.5.2 Actions

Actions

Actions are components that have a list of individual **Instructions** which are executed from top to bottom. It's important to note that an **Instruction** won't be executed until the previous one has finished.

Actions

Task List

Actions can be thought as task lists that must be completed from top to bottom.

CREATING ACTIONS

There are two ways to create an Actions object. One is to create an object that contains an Actions component, by right clicking on the *Hierarchy* panel and selecting *Game Creator* *Visual Scripting* *Actions* This creates a scene object with the component attached to it.

However, an Actions component can also be added to any game object. Simply click on any game object's *Add Component* button and type Actions.

Deleting Actions

To delete an Actions component, simply click on the component's little cog button and select "Remove Component" from the dropdown menu.

ADDING INSTRUCTIONS

To add an **Instruction** to an **Actions** component, click on the "Add Instruction" button to pop a dropdown list with a searchable field. Navigate through the different categories or search for a specific instruction and click it to add it at the bottom of the list.

Add a new Instruction

It is also possible to add **Instructions** at any point of the list. To do so, right click on any existing **Instruction** and choose "Insert Above" or "Insert Below" from the contextual menu that appears.

Accessible Fuzzy Search

Game Creator uses an advanced indexed search algorithm that allows to both syntactically and semmantically understand what the user is trying to search, even if the search contains mispelled words. For example, searching for "move" will display the "Move Character" instruction, but also the "Change Position" one.

BUILT-IN DOCUMENTATION

All **Instructions** have built-in documentation that explain what it does as well as a small description of each of its parameters. To access its documentation, either search for that particular instruction on the documentation, or right click it on the **Instruction** and select *Help*. A new floating window will appear with all the necessary information.

Instruction Documentation

DEBUGGING TOOLS

Actions come with built-in tools that allow to easily visualize and what's happening at runtime. Right click on any **Instruction** to pop a context menu with the *Disable* and add a *Breakpoint* options.

Disable Instruction

This option disables a particular instruction, as if it was not there.

Disable Instruction

The **Instruction** is greyed out and a special icon appears on its right side. Click the icon to enable the instruction again.

Add a Breakpoint

A breakpoint pauses the Unity Editor upon reaching a particular Instruction, right before executing it. This is very useful if you want to check the state of certain data before the execution progresses any further.

Breakpoint Instruction

When an **Instruction** has a breakpoint, it displays a red icon on its right side. Clicking it will remove the breakpoint from the Instruction.

Editor only

It is important to note that *breakpoints* only work on the Editor and have no effect when building the project as a standalone application.

Instructions

INSTRUCTIONS

Sub Categories

- Animator
- Application
- Audio
- Cameras
- Characters
- Debug
- Game Objects
- Lights
- Logic
- Math
- Physics 2D
- Physics 3D
- Renderer
- Scenes
- Storage
- Testing
- Time
- Transforms
- Ui
- Variables

ANIMATOR

Animator

Instructions

- Change Animator Float
- Change Animator Integer
- Change Animator Layer
- Change Blend Shape
- Play Animation Clip
- Set Animator Boolean
- Set Animator Trigger

Change Animator Float

```
Animator » Change Animator Float
```

Description

Changes the value of a 'Float' Animator parameter

Parameters

Name	Description
Parameter Name	The Animator parameter name to be modified
Value	The value of the parameter that is set
Duration	How long it takes to perform the transition
Easing	The change rate of the parameter over time
Wait to Complete	Whether to wait until the transition is finished
Animator	The Animator component attached to the game object

Keywords

`Parameter` `Number`

Change Animator Integer

```
Animator » Change Animator Integer
```

Description

Changes the value of a 'Integer' Animator parameter

Parameters

Name	Description
Parameter Name	The Animator parameter name to be modified
Value	The value of the parameter that is set
Duration	How long it takes to perform the transition
Easing	The change rate of the parameter over time
Wait to Complete	Whether to wait until the transition is finished
Animator	The Animator component attached to the game object

Keywords

`Parameter` `Number`

Change Animator Layer

```
Animator » Change Animator Layer
```

Description

Changes the weight of an Animator Layer

Parameters

Name	Description
Layer Index	The Animator's Layer index that's being modified
Weight	The target Animator layer weight
Duration	How long it takes to perform the transition
Easing	The change rate of the parameter over time
Wait to Complete	Whether to wait until the transition is finished
Animator	The Animator component attached to the game object

Keywords

`Weight`

Change Blend Shape

```
Animator » Change Blend Shape
```

Description

Changes the value of a Blend Shape parameter

Parameters

Name	Description
Skinned Mesh	The Skinned Mesh Renderer component attached to the game object
Blend Shape	Name of the Blend Shape to change
Value	The target value of the blend shape
Duration	How long it takes to perform the transition
Easing	The change rate of the parameter over time
Wait to Complete	Whether to wait until the transition is finished

Keywords

`Morph` `Target`

Play Animation Clip

```
Animator » Play Animation Clip
```

Description

Plays an Animation Clip on the chosen Animator

Parameters

Name	Description
Animation Clip	The Animation Clip that is played
Animator	The Animator component attached to the game object

Keywords

`Animate` `Reproduce` `Sequence` `Cinematic`

Set Animator Boolean

```
Animator » Set Animator Boolean
```

Description

Sets the value of a 'Bool' Animator parameter

Parameters

Name	Description
Parameter Name	The Animator parameter name to be modified
Value	The value of the parameter that is set
Animator	The Animator component attached to the game object

Keywords

`Parameter` `Bool`

Set Animator Trigger

```
Animator » Set Animator Trigger
```

Description

Sets the value of a 'Trigger' Animator parameter

Parameters

Name	Description
Parameter Name	The Animator parameter name modified
Animator	The Animator component attached to the game object

Keywords

`Parameter` `Once` `Flag` `Notify`

APPLICATION

Application

Sub Categories

- Cursor

Instructions

- Open Web Page
- Quit Application

Open Web Page

```
Application » Open Web Page
```

Description

Opens the specified URL with the default web browser

Parameters

Name	Description
URL	The route link to open. Must include the protocol prepended (http or https)

Keywords

`Site` `Internet`

Quit Application

```
Application » Quit Application
```

Description

Closes the application and exits the program. This instruction is ignored in the Unity Editor or WebGL platforms

Keywords

`Exit` `Close` `Shutdown` `Turn`

Cursor

Cursor Instructions

- Cursor Texture
- Cursor Visibility
- Lock Cursor

Cursor Texture

```
Application » Cursor » Cursor Texture
```

Description

Changes the image of the hardware cursor

Parameters

Name	Description
Texture	The new appearance of the cursor. The texture must be set to Cursor type
Tip	The offset from the top left of the texture used as the target point
Mode	Determines if the cursor is rendered using software or hardware rendering

Keywords

`Mouse` `Crosshair` `Click`

Cursor Visibility

```
Application » Cursor » Cursor Visibility
```

Description

Determines if the hardware cursor is visible or not

Parameters

Name	Description
Is Visible	If true the cursor is visible, unless it is set as Locked

Keywords

`Mouse` `FPS` `Crosshair`

Lock Cursor

```
Application » Cursor » Lock Cursor
```

Description

Determines if the hardware pointer is locked to the center of the view or not

Parameters

Name	Description
Lock Mode	The behavior of the cursor. The default value is None

Keywords

`Mouse` `State` `FPS` `Center` `Confine`

AUDIO

Audio

Instructions

- Audio Mixer Parameter
- Change Ambient Volume
- Change Master Volume
- Change Snapshot
- Change Sound Effects Volume
- Change Speech Volume
- Change Ui Volume
- Play Ambient
- Play Sound Effect
- Play Speech
- Play Ui Sound
- Stop Ambient
- Stop Sound Effect
- Stop Speech On Game Object

Audio Mixer Parameter

```
Audio » Audio Mixer Parameter
```

Description

Changes the value of an Audio Mixer exposed parameter

Parameters

Name	Description
Audio Mixer	The Audio Mixer asset with the exposed parameter
Parameter Name	A string representing the name of the exposed parameter
Parameter Value	The value which the exposed parameter is set

Keywords

`Float` `Exposed` `Effect` `Change`

Change Ambient volume

```
Audio » Change Ambient volume
```

Description

Change the Volume of Ambient music

Parameters

Name	Description
Volume	A value between 0 and 1 that indicates the volume percentage

Keywords

`Audio` `Ambience` `Music` `Background` `Volume` `Level`

Change Master volume

```
Audio » Change Master volume
```

Description

Change the Master volume. The Master volume controls how loud all other channels are

Parameters

Name	Description
Volume	A value between 0 and 1 that indicates the volume percentage

Keywords

`Audio` `Sounds` `Volume` `Level`

Change Snapshot

```
Audio » Change Snapshot
```

Description

Smoothly transitions to a new snapshot over a period of time

Parameters

Name	Description
Snapshot	The Audio Mixer Snapshot that is activated
Transition	How long it takes to transition to the new Snapshot

Keywords

`Effect` `Transition` `Effect` `Change`

Change Sound Effects volume

```
Audio » Change Sound Effects volume
```

Description

Change the Volume of Sound Effects

Parameters

Name	Description
Volume	A value between 0 and 1 that indicates the volume percentage

Keywords

`Audio` `Sounds` `Volume` `Level`

Change Speech volume

```
Audio » Change Speech volume
```

Description

Change the Volume of character Speech

Parameters

Name	Description
Volume	A value between 0 and 1 that indicates the volume percentage

Keywords

`Audio` `Character` `Voice` `Voices` `Volume` `Level`

Change UI volume

```
Audio » Change UI volume
```

Description

Change the Volume of UI elements

Parameters

Name	Description
Volume	A value between 0 and 1 that indicates the volume percentage

Keywords

`Audio` `User` `Interface` `Button` `Volume` `Level`

Play Ambient

```
Audio » Play Ambient
```

Description

Plays a looped Audio Clip. Useful for background music or persistent sounds.

Parameters

Name	Description
Audio Clip	The Audio Clip to be played
Transition In	Time it takes for the sound to fade in
Spatial Blending	Whether the sound is placed in a 3D space or not
Target	A Game Object reference that the sound follows as the source

Keywords

`Audio` `Music` `Ambience` `Background`

Play Sound Effect

```
Audio » Play Sound Effect
```

Description

Plays an Audio Clip sound effect just once

Parameters

Name	Description
Audio Clip	The Audio Clip to be played
Wait To Complete	Check if you want to wait until the sound finishes
Pitch	A random pitch value ranging between two values
Transition In	Time it takes for the sound to fade in
Spatial Blending	Whether the sound is placed in a 3D space or not
Target	A Game Object reference that the sound follows as its source

Keywords

`Audio` `Sounds`

Play Speech

```
Audio » Play Speech
```

Description

Plays an Audio Clip speech over just once

Parameters

Name	Description
Audio Clip	The Audio Clip to be played
Wait To Complete	Check if you want to wait until the sound finishes
Spatial Blending	Whether the sound is placed in a 3D space or not
Target	A Game Object reference that the sound follows as its source

Keywords

`Audio` `Voice` `Voices` `Sounds` `Character`

Play UI sound

```
Audio » Play UI sound
```

Description

Plays a non-diegetic user interface Audio Clip

Parameters

Name	Description
Audio Clip	The Audio Clip to be played
Wait To Complete	Check if you want to wait until the sound finishes
Pitch	A random pitch value ranging between two values
Spatial Blending	Whether the sound is placed in a 3D space or not
Target	A Game Object reference that the sound follows as its source

Keywords

`Audio` `Sounds` `User` `Interface` `Beep` `Button`

Stop Ambient

```
Audio » Stop Ambient
```

Description

Stops a currently playing Ambient audio

Parameters

Name	Description
Audio Clip	The Audio Clip to be played
Wait To Complete	Check if you want to wait until the sound has faded out
Transition Out	Time it takes for the sound to fade out

Keywords

`Audio` `Music` `Ambience` `Background` `Fade` `Mute`

Stop Sound Effect

```
Audio » Stop Sound Effect
```

Description

Stops a currently playing Sound Effect

Keywords

`Audio` `Sounds` `Silence` `Fade` `Mute`

Stop Speech on Game Object

```
Audio » Stop Speech on Game Object
```

Description

Stops any Speech clips being played by a specific Game Object

Parameters

Name	Description
Target	A game object that is set as the source of the speech

Keywords

`Audio` `Voice` `Voices` `Sounds` `Character` `Silence` `Mute` `Fade`

CAMERAS

Cameras

Sub Categories

- Properties
- Shakes
- Shots

Instructions

- Change To Shot
- Revert To Previous Shot

Change to Shot

```
Cameras » Change to Shot
```

Description

Changes the active Shot for a particular camera

Parameters

Name	Description
Camera	The target camera component
Shot	The camera Shot that becomes active
Duration	How long it takes to transition to the new Shot, in seconds
Wait To Complete	If the instruction waits till the transition is complete

Keywords

`Cameras` `Render` `Switch` `Move`

Revert to Previous Shot

```
Cameras » Revert to previous Shot
```

Description

Reverts the active Shot of a particular camera to the previous one

Parameters

Name	Description
Camera	The target camera component
Duration	How long it takes to transition to the new Shot, in seconds

Keywords

`Cameras` `Render` `Switch` `Move`

Properties

Properties Instructions

- Change Culling Mask
- Change Field Of View
- Change Projection

Change Culling Mask

```
Cameras » Properties » Change Culling Mask
```

Description

Changes the camera culling mask

Parameters

Name	Description
Camera	The camera component whose property changes
Culling Mask	The mask the camera uses to discern which objects to render

Keywords

`Cameras` `Render`

Change Field of View

```
Cameras » Properties » Change Field of View
```

Description

Changes the camera field of view

Parameters

Name	Description
Camera	The camera component whose property changes
FoV	The field of view of the camera, measured in degrees

Keywords

`Cameras` `Perspective` `FOV` `3D`

Change Projection

```
Cameras » Properties » Change Projection
```

Description

Changes the camera projection to either Perspective or Orthographic

Parameters

Name	Description
Camera	The camera component whose property changes
Projection	Whether to change to Orthographic or Perspective mode

Keywords

`Cameras` `Orthographic` `Perspective` `3D` `2D`

Shakes

Shakes Instructions

- Shake Camera Burst
- Shake Camera Sustain
- Stop Camera Sustain Shake
- Stop Shake Camera Bursts

Shake Camera Burst

```
Cameras » Shakes » Shake Camera Burst
```

Description

Shakes the camera for an amount of time

Parameters

Name	Description
Camera	The camera that receives the burst shake effect
Delay	Amount of time in seconds before the shake effect starts
Duration	Amount of time the shake effect stays active
Shake Position	If the shake affects the position of the camera
Shake Rotation	If the shake affects the rotation of the camera
Magnitude	The maximum amount the camera displaces from its position
Roughness	Frequency or how violently the camera shakes
Transform	[Optional] Defines the origin of the shake
Radius	[Optional] Distance from the origin that the shake starts to fall-off

Keywords

`Cameras` `Animation` `Animate` `Shake` `Impact` `Play`

Shake Camera Sustain

```
Cameras » Shakes » Shake Camera Sustain
```

Description

Starts shaking the camera until the effect is manually turned off

Parameters

Name	**Description**
Camera	The camera that receives the sustain shake effect
Delay	Amount of time in seconds before the shake effect starts
Transition	Amount of seconds the shake effect takes to blend in
Shake Position	Whether the shake affects the position of the camera
Shake Rotation	Whether the shake affects the rotation of the camera
Magnitude	The maximum amount the camera displaces from its position
Roughness	Frequency or how violently the camera shakes
Transform	[Optional] Defines the origin of the shake
Radius	[Optional] Distance from the origin that the shake starts to fall-off

Keywords

Cameras Animation Animate Shake Wave Play

Stop Camera Sustain Shake

```
Cameras » Shakes » Stop Camera Sustain Shake
```

Description

Stops a Sustain Shake camera effect in a particular layer layer

Parameters

Name	Description
Camera	The camera target that stops a Sustain Shake effect
Layer	The camera layer from which the Sustain Shake effect is removed
Delay	Amount of time before the Sustain Shake effect starts blending out
Transition	Amount of time it takes to blend out the Sustain Shake effect

Keywords

`Cameras` `Animation` `Animate` `Shake` `Wave` `Play`

Stop Camera Shake Bursts

```
Cameras » Shakes » Stop Shake Camera Bursts
```

Description

Stops any ongoing camera Burst Shake effects

Parameters

Name	Description
Camera	The camera target that stops all its active Burst Shake effects
Delay	Amount of time before all Burst Shake effects start blending out
Transition	Amount of time it takes to blend out all Burst Shake effects

Keywords

`Cameras` `Animation` `Animate` `Shake` `Impact` `Play`

Shots

Shots Sub Categories

- Anchor
- Animation
- First Person
- Follow
- Head Bobbing
- Head Leaning
- Lock On
- Look
- Noise
- Orbit
- Zoom

Instructions

- Change Main Shot

Change Main Shot

```
Cameras » Shots » Change Main Shot
```

Description

Assigns as the Main Shot a new Camera Shot

Parameters

Name	Description
Shot	The new main Camera Shot

Anchor
Anchor Instructions

- Change Distance
- Change Offset
- Change Target
- Enable Anchor

Change Distance

```
Cameras » Shots » Anchor » Change Distance
```

Description

Changes the anchored position the Shot sits relative to the target

Parameters

Name	Description
Distance	The new distance relative to the target in local coordinates
Shot	The camera Shot targeted

Keywords

`Cameras` `View` `Cameras` `Shot`

Change Offset

```
Cameras » Shots » Anchor » Change Offset
```

Description

Changes the offset position of the targeted object

Parameters

Name	Description
Offset	The new offset in target local coordinates
Shot	The camera Shot targeted

Keywords

`Cameras` `Track` `View` `Cameras` `Shot`

Change Target

```
Cameras » Shots » Anchor » Change Target
```

Description

Changes the targeted game object

Parameters

Name	Description
Target	The new target
Shot	The camera Shot targeted

Keywords

`Cameras` `Track` `View` `Cameras` `Shot`

Enable Anchor

```
Cameras » Shots » Anchor » Enable Anchor
```

Description

Toggles the active state of a Camera Shot's Anchor system

Parameters

Name	Description
Active	The next state
Shot	The camera Shot targeted

Keywords

`Cameras` `Disable` `Activate` `Deactivate` `Bool` `Toggle` `Off` `On` `Cameras` `Shot`

Animation
Animation Instructions

- Change Duration
- Enable Animation

Change Duration

```
Cameras » Shots » Animation » Change Duration
```

Description

Changes the duration it takes for the Animation shot to complete

Parameters

Name	Description
Duration	The new duration in seconds
Shot	The camera Shot targeted

Keywords

`Cameras` `Track` `View` `Cameras` `Shot`

Enable Animation

```
Cameras » Shots » Animation » Enable Animation
```

Description

Toggles the active state of a Camera Shot's Animation system

Parameters

Name	Description
Active	The next state
Shot	The camera Shot targeted

Keywords

`Cameras` `Disable` `Activate` `Deactivate` `Bool` `Toggle` `Off` `On` `Cameras` `Shot`

First person
First Person Instructions

- Change Max Pitch
- Change Offset
- Change Sensitivity
- Change Smooth Time
- Change Target
- Enable First Person

Change Max Pitch

```
Cameras » Shots » First Person » Change Max Pitch
```

Description

Changes the maximum rotation (up and down) allowed

Parameters

Name	Description
Max Pitch	The amount the Shot is allowed to look up and down, in degrees
Shot	The camera Shot targeted

Keywords

`Cameras` `Shot`

Change Offset

```
Cameras » Shots » First Person » Change Offset
```

Description

Changes the offset position of the targeted object

Parameters

Name	Description
Offset	The new offset in self local coordinates
Shot	The camera Shot targeted

Keywords

`Cameras` `Shot`

Change Sensitivity

```
Cameras » Shots » First Person » Change Sensitivity
```

Description

Changes how sensitive the Shot reacts to input

Parameters

Name	Description
Sensitivity	Input sensitivity for X and the Y axis
Shot	The camera Shot targeted

Keywords

`Cameras` `Shot`

Change Smooth Time

```
Cameras » Shots » First Person » Change Smooth Time
```

Description

Changes the maximum rotation (up and down) allowed

Parameters

Name	Description
Smooth Time	How smooth the camera operates when rotating
Shot	The camera Shot targeted

Keywords

`Cameras` `Shot`

Change Target

```
Cameras » Shots » First Person » Change Target
```

Description

Changes the targeted game object to view from

Parameters

Name	Description
Target	The new target
Shot	The camera Shot targeted

Keywords

`Cameras` `Track` `View` `Cameras` `Shot`

Enable First Person

```
Cameras » Shots » First Person » Enable First Person
```

Description

Toggles the active state of a Camera Shot's First Person system

Parameters

Name	Description
Active	The next state
Shot	The camera Shot targeted

Keywords

`Cameras` `Disable` `Activate` `Deactivate` `Bool` `Toggle` `Off` `On` `Cameras` `Shot`

Follow
Follow Instructions

- Change Distance
- Change Target
- Enable Follow

Change Distance

```
Cameras » Shots » Follow » Change Distance
```

Description

Changes the offset distance between the Shot and the targeted object

Parameters

Name	Description
Distance	The new offset distance in world coordinates
Shot	The camera Shot targeted

Keywords

`Cameras` `Track` `View` `Cameras` `Shot`

Change Target

```
Cameras » Shots » Follow » Change Target
```

Description

Changes the targeted game object to Follow

Parameters

Name	Description
Follow	The new target to follow
Shot	The camera Shot targeted

Keywords

`Cameras` `Track` `View` `Cameras` `Shot`

Enable Follow

```
Cameras » Shots » Follow » Enable Follow
```

Description

Toggles the active state of a Camera Shot's Follow system

Parameters

Name	Description
Active	The next state
Shot	The camera Shot targeted

Keywords

`Cameras` `Disable` `Activate` `Deactivate` `Bool` `Toggle` `Off` `On` `Cameras` `Shot`

Head bobbing

Head Bobbing Instructions

- Enable Head Bobbing

Enable Head Bobbing

```
Cameras » Shots » Head Bobbing » Enable Head Bobbing
```

Description

Toggles the active state of a Camera Shot's Head Bobbing system

Parameters

Name	Description
Active	The next state
Shot	The camera Shot targeted

Keywords

`Cameras` `Disable` `Activate` `Deactivate` `Bool` `Toggle` `Off` `On` `Cameras` `Shot`

Head leaning
Head Leaning Instructions

- Enable Head Leaning

Enable Head Leaning

```
Cameras » Shots » Head Leaning » Enable Head Leaning
```

Description

Toggles the active state of a Camera Shot's Head Leaning system

Parameters

Name	Description
Active	The next state
Shot	The camera Shot targeted

Keywords

`Cameras` `Disable` `Activate` `Deactivate` `Bool` `Toggle` `Off` `On` `Cameras` `Shot`

Lock on
Lock On Instructions

- Change Anchor
- Change Distance
- Change Offset
- Enable Lock On

Change Anchor

```
Cameras » Shots » Lock On » Change Anchor
```

Description

Changes the targeted game object to Lock On

Parameters

Name	Description
Anchor	The new target to Anchor onto
Shot	The camera Shot targeted

Keywords

`Cameras` `Track` `View` `Cameras` `Shot`

Change Distance

```
Cameras » Shots » Lock On » Change Distance
```

Description

Changes the distance from the anchor point

Parameters

Name	Description
Distance	The new distance in self local coordinates
Shot	The camera Shot targeted

Keywords

`Cameras` `Track` `View` `Cameras` `Shot`

Change Offset

```
Cameras » Shots » Lock On » Change Offset
```

Description

Changes the offset position of the targeted object

Parameters

Name	Description
Offset	The new offset in self local coordinates
Shot	The camera Shot targeted

Keywords

`Cameras` `Track` `View` `Cameras` `Shot`

Enable Lock On

```
Cameras » Shots » Lock On » Enable Lock On
```

Description

Toggles the active state of a Camera Shot's Lock On system

Parameters

Name	Description
Active	The next state
Shot	The camera Shot targeted

Keywords

`Cameras` `Disable` `Activate` `Deactivate` `Bool` `Toggle` `Off` `On` `Cameras` `Shot`

Look
Look Instructions

- Change Offset
- Change Target
- Enable Look

Change Offset

```
Cameras » Shots » Look » Change Offset
```

Description

Changes the offset position of the targeted object

Parameters

Name	Description
Offset	The new offset in self local coordinates
Shot	The camera Shot targeted

Keywords

`Cameras` `Track` `View` `Cameras` `Shot`

Change Target

```
Cameras » Shots » Look » Change Target
```

Description

Changes the targeted game object to look

Parameters

Name	Description
Target	The new target
Shot	The camera Shot targeted

Keywords

`Cameras` `Track` `View` `Cameras` `Shot`

Enable Look

```
Cameras » Shots » Look » Enable Look
```

Description

Toggles the active state of a Camera Shot's Look system

Parameters

Name	Description
Active	The next state
Shot	The camera Shot targeted

Keywords

`Cameras` `Disable` `Activate` `Deactivate` `Bool` `Toggle` `Off` `On` `Cameras` `Shot`

Noise
Noise Instructions

- Enable Noise

Enable Noise

```
Cameras » Shots » Noise » Enable Noise
```

Description

Toggles the active state of a Camera Shot's Noise system

Parameters

Name	Description
Active	The next state
Shot	The camera Shot targeted

Keywords

`Cameras` `Disable` `Activate` `Deactivate` `Bool` `Toggle` `Off` `On` `Cameras` `Shot`

Orbit
Orbit Instructions

- Change Alignment
- Change Max Pitch
- Change Max Radius
- Change Offset
- Change Sensitivity
- Change Smooth Time
- Change Target
- Enable Orbit

Change Alignment

```
Cameras » Shots » Orbit » Change Alignment
```

Description

Changes whether and how the Shot aligns behind the targeted object

Parameters

Name	Description
Align with Target	If the Shot should move behind the target after some idle time
Delay	If the Shot should move behind the target after some idle time
Smooth Time	The speed at which
Shot	The camera Shot targeted

Keywords

`Cameras` `Shot`

Change Max Pitch

```
Cameras » Shots » Orbit » Change Max Pitch
```

Description

Changes the maximum rotation (up and down) allowed

Parameters

Name	Description
Max Pitch	The amount the Shot is allowed to look up and down, in degrees
Shot	The camera Shot targeted

Keywords

`Cameras` `Shot`

Change Max Radius

```
Cameras » Shots » Orbit » Change Max Radius
```

Description

Changes the maximum rotation (up and down) allowed

Parameters

Name	Description
Max Radius	The amount the Shot is allowed to look up and down, in degrees
Shot	The camera Shot targeted

Keywords

`Cameras` `Shot`

Change Offset

```
Cameras » Shots » Orbit » Change Offset
```

Description

Changes the offset position of the targeted object to orbit

Parameters

Name	Description
Offset	The new offset in self local coordinates
Shot	The camera Shot targeted

Keywords

`Cameras` `Track` `View` `Cameras` `Shot`

Change Sensitivity

```
Cameras » Shots » Orbit » Change Sensitivity
```

Description

Changes how sensitive the Shot reacts to input

Parameters

Name	Description
Sensitivity	Input sensitivity for X and the Y axis
Shot	The camera Shot targeted

Keywords

`Cameras` `Shot`

Change Smooth Time

```
Cameras » Shots » Orbit » Change Smooth Time
```

Description

Changes how smooth the orbit responds to input

Parameters

Name	Description
Smooth Time	How smooth is the orbital translation
Shot	The camera Shot targeted

Keywords

`Cameras` `Shot`

Change Target

```
Cameras » Shots » Orbit » Change Target
```

Description

Changes the targeted game object to orbit around

Parameters

Name	Description
Target	The new target
Shot	The camera Shot targeted

Keywords

`Cameras` `Track` `View` `Cameras` `Shot`

Enable Orbit

```
Cameras » Shots » Orbit » Enable Orbit
```

Description

Toggles the active state of a Camera Shot's Orbit system

Parameters

Name	Description
Active	The next state
Shot	The camera Shot targeted

Keywords

`Cameras` `Disable` `Activate` `Deactivate` `Bool` `Toggle` `Off` `On` `Cameras` `Shot`

Zoom
Zoom Instructions

- Change Level Zoom
- Change Min Distance
- Change Smooth Time
- Enable Zoom

Change Level Zoom

```
Cameras » Shots » Zoom » Change Level Zoom
```

Description

Changes the targeted zoom level percentage

Parameters

Name	Description
Level	The zoom level value between zero and one
Shot	The camera Shot targeted

Keywords

`Cameras` `Shot`

Change Min Distance

```
Cameras » Shots » Zoom » Change Min Distance
```

Description

Changes the targeted zoom level percentage

Parameters

Name	Description
Min Distance	The minimum zoom distance between the target and the Shot
Shot	The camera Shot targeted

Keywords

`Cameras` `Shot`

Change Smooth Time

```
Cameras » Shots » Zoom » Change Smooth Time
```

Description

Changes how smooth the zoom responds to input

Parameters

Name	Description
Smooth Time	How smooth is the zoom transition
Shot	The camera Shot targeted

Keywords

`Cameras` `Shot`

Enable Zoom

```
Cameras » Shots » Zoom » Enable Zoom
```

Description

Toggles the active state of a Camera Shot's Zoom system

Parameters

Name	Description
Active	The next state
Shot	The camera Shot targeted

Keywords

`Cameras` `Disable` `Activate` `Deactivate` `Bool` `Toggle` `Off` `On` `Cameras` `Shot`

CHARACTERS

Characters

Sub Categories

- Animation
- Footsteps
- Interaction
- Navigation
- Player
- Properties
- Ragdoll
- Visuals

Animation

Animation Instructions

- Change Exertion
- Change Heart Rate
- Change Smooth Time
- Change State Weight
- Change Twitching
- Enter State
- Play Gesture
- Stop Gesture
- Stop State

Change Exertion

```
Characters » Animation » Change Exertion
```

Description

Changes the Exertion value of a Character over time

Parameters

Name	Description
Exertion	The target Exertion value between 0 and 1. Default is 0.25
Duration	How long it takes to perform the transition
Easing	The change rate of the parameter over time
Wait to Complete	Whether to wait until the transition finishes
Character	The game object with the Character target

Example 1

The Heart Rate value goes hand in hand with the Exertion. The Heart Rate controls the speed that the breathing animation plays. The Exertion controls the magnitude of the breathing animation.

Keywords

`Tire` `Effort` `Struggle` `Sweat` `Exercise`

Change Heart Rate

```
Characters » Animation » Change Heart Rate
```

Description

Changes the Heart Rate value of a Character over time

Parameters

Name	Description
Heart Rate	The target Heart Rate value between 0 and 2. Default is 1
Duration	How long it takes to perform the transition
Easing	The change rate of the parameter over time
Wait to Complete	Whether to wait until the transition finishes
Character	The game object with the Character target

Example 1

The Heart Rate value goes hand in hand with the Exertion. The Heart Rate controls the speed that the breathing animation plays. The Exertion controls the magnitude of the breathing animation.

Keywords

`Breathe` `Pump` `Beat` `Pulse`

Change Smooth Time

```
Characters » Animation » Change Smooth Time
```

Description

Changes the average blend time between locomotion animations

Parameters

Name	Description
Smooth Time	The target Smooth Time value. Values usually range between 0 and 0.5
Duration	How long it takes to perform the transition
Easing	The change rate of the parameter over time
Wait to Complete	Whether to wait until the transition is finished
Character	The game object with the Character target

Example 1

The Smooth Time controls how fast a Character animation blends into another when reacting to external factors. A value of 0 makes the Character react instantly whereas a value of 0.5 takes half a second to completely blend in. A value between 0.2 and 0.4 usually provide the best results, though it depends on the look and feel the creator wants to achieve.

Keywords

`Fade` `Realistic` `Old` `School` `Reaction`

Change State Weight

```
Characters » Animation » Change State Weight
```

Description

Changes the weight of the State over time at the specified layer

Parameters

Name	Description
Character	The character that plays the animation state
Layer	Slot number in which the animation state is allocated
Weight	The targeted opacity of the animation
Transition	The duration of the transition, in seconds

Keywords

`Characters` `Animation` `Blend` `State` `Opacity`

Change Twitching

```
Characters » Animation » Change Twitching
```

Description

Changes the magnitude of the subtle and random movement applied to each Character's bone

Parameters

Name	Description
Twitching	The target Twitching value between 0 and 1. Default is 1
Duration	How long it takes to perform the transition
Easing	The change rate of the parameter over time
Wait to Complete	Whether to wait until the transition is finished
Character	The game object with the Character target

Example 1

The Twitching value allows a Character to express subtle random movement found in life beings. Paired with the Breathing animation, it allows to have a consistent rhythm even when blending between other animations. It can also be useful to create idle animations using a static pose.

Keywords

`Tire` `Effort` `Struggle` `Sweat` `Exercise`

Enter State

```
Characters » Animation » Enter State
```

Description

Makes a Character start an animation State

Parameters

Name	Description
Character	The character that plays the animation state
State	The animation data necessary to play a state
Layer	Slot number in which the animation state is allocated
Blend Mode	Additively adds the new animation on top of the rest or overrides any lower layer animations
Delay	Amount of seconds to wait before the animation starts to play
Speed	Speed coefficient at which the animation plays
Weight	The opacity of the animation that plays. Between 0 and 1
Transition	The amount of seconds the animation takes to blend in

Keywords

`Characters` `Animation` `Animate` `State` `Play`

Play Gesture

```
Characters » Animation » Play Gesture
```

Description

Plays an Animation Clip on a Character once

Parameters

Name	Description
Character	The character that plays the animation
Animation Clip	The Animation Clip that is played
Avatar Mask	(Optional) Allows to play the animation on specific body parts of the Character
Blend Mode	Additively adds the new animation on top of the rest or overrides any lower layer animations
Delay	Amount of seconds to wait before the animation starts to play
Speed	Speed coefficient at which the animation plays. 1 means normal speed
Transition In	The amount of seconds the animation takes to blend in
Transition Out	The amount of seconds the animation takes to blend out
Wait To Complete	If true this Instruction waits until the animation is complete

Keywords

`Characters` `Animation` `Animate` `Gesture` `Play`

Stop Gestures

```
Characters » Animation » Stop Gesture
```

Description

Stops any animation Gestures playing on the Character

Parameters

Name	Description
Character	The character that plays animation Gestures
Delay	Amount of seconds to wait before the animation starts to blend out
Transition	The amount of seconds the animation takes to blend out

Keywords

`Characters` `Animation` `Animate` `Gesture` `Play`

Stop State

```
Characters » Animation » Stop State
```

Description

Stops an animation State from a Character

Parameters

Name	Description
Character	The character that stops its animation State
Layer	Slot number from which the state is removed
Delay	Amount of seconds to wait before the animation stops playing
Transition	The amount of seconds the animation takes to blend out

Keywords

`Characters` `Animation` `Animate` `State` `Exit` `Stop`

Footsteps

Footsteps Instructions

- Change Footstep Sounds
- Play Footstep

Change Footstep Sounds

```
Characters » Footsteps » Change Footstep Sounds
```

Description

Changes the sound table that links textures with footstep sounds

Parameters

Name	Description
Character	The character that plays animation Gestures
Footsteps	The sound table asset that contains information about how and when footstep sounds play

Keywords

`Character` `Foot` `Step` `Stomp` `Foliage` `Audio` `Run` `Walk` `Move`

Play Footstep

```
Characters » Footsteps » Play Footstep
```

Description

Plays a Footstep sound from a Material Sound asset

Parameters

Name	Description
Character	The character target
Material Sound	The material sound asset

Keywords

`Step` `Foot` `Impact` `Land` `Sound`

Interaction

Interaction Instructions

- Interact

Interact

```
Characters » Interaction » Interact
```

Description

Changes how the Player Character reacts to input commands

Parameters

Name	Description
Character	The Character that attempts to interact

Keywords

`Character` `Button` `Pick` `Do` `Use` `Pull` `Press` `Push` `Talk`

Navigation

Navigation Instructions

- Dash
- Jump
- Move Direction
- Move To
- Set Character Rotation
- Start Following
- Stop Following
- Stop Move
- Teleport

Dash

```
Characters » Navigation » Dash
```

Description

Moves the Character in the chosen direction for a brief period of time

Parameters

Name	Description
Direction	Vector oriented towards the desired direction
Speed	Velocity the Character moves throughout the whole movement
Damping	Defines the duration and gradually changes the rate of the movement over time
Wait to Finish	If true this Instruction waits until the dash is completed
Animation Forward	Animation played on the Character when dashing forward
Animation Backward	Animation played on the Character when dashing backwards
Animation Right	Animation played on the Character when dashing right
Animation Left	Animation played on the Character when dashing left
Character	The game object with the Character target

Example 1

The Damping value defines both the duration and the velocity rate at which the Character moves when performing the Dash. To change the duration of the dash open the animation curve window and move the last keyframe to the left to decrease the duration or to the right to increase it.

Example 2

The Damping value also defines the coefficient rate at which the Character moves while performing the Dash. By default the Character starts with a coefficient of 0. After 0.2 seconds it increases to 1 and goes back to 0 after 0.8 seconds. This curve is evaluated while performing a Dash and the coefficient is extracted from the curve and multiplied by the Speed to gradually change the rate at which the Character moves. For this reason, it is recommended that the Damping stay between 0 and 1.

Keywords

`Leap` `Blink` `Roll` `Flash` `Character` `Player`

Jump

```
Characters » Navigation » Jump
```

Description

Instructs the Character to jump

Parameters

Name	Description
Character	The game object with the Character target

Keywords

`Hop` `Leap` `Reach` `Character` `Player`

Move Direction

```
Characters » Navigation » Move Direction
```

Description

Attempts to move the Character towards the specified direction

Parameters

Name	Description
Direction	The the direction to move towards
Priority	Indicates the priority of this command against others
Character	The game object with the Character target

Keywords

`Constant` `Walk` `Run` `To` `Vector` `Character` `Player`

Move To

```
Characters » Navigation » Move To
```

Description

Instructs the Character to move to a new location

Parameters

Name	Description
Wait to Finish	If true this Instruction waits until the Character reaches its destination or it is canceled
Stop Distance	Distance to the destination that the Character considers it has reached the target
Character	The game object with the Character target

Example 1

The Stop Distance field is useful if you want [Character A] to approach another [Character B]. With a Stop Distance of 0, [Character A] tries to occupy the same space as the other one, bumping into it. Having a Stop Distance value of 2 allows [Character A] to stop 2 units away from [Character B]'s position

Keywords

`Walk` `Run` `Position` `Location` `Destination` `Character` `Player`

Set Character Rotation

```
Characters » Navigation » Set Character Rotation
```

Description

Changes the rotation behavior of the Character

Parameters

Name	Description
Character	The Character that changes its Rotation behavior
Rotation	The Rotation behavior that decides where the Character faces

Keywords

`Character` `Face` `Look` `Direction` `Pivot` `Lock`

Start Following

```
Characters » Navigation » Start Following
```

Description

Instructs a Character to follow another game object

Parameters

Name	Description
Target	The target game object to follow
Min Distance	Distance from the Target the Character aims to move when approaching the Target
Max Distance	Maximum distance to the Target the Character leaves before attempting to move closer
Character	The game object with the Character target

Keywords

`Lead` `Pursue` `Chase` `Walk` `Run` `Position` `Location` `Destination` `Character` `Player`

Stop Following

```
Characters » Navigation » Stop Following
```

Description

Instructs a Character to stop following a game object

Parameters

Name	Description
Character	The game object with the Character target

Keywords

`Cancel` `Lead` `Pursue` `Chase` `Character` `Player`

Stop Move

```
Characters » Navigation » Stop Move
```

Description

Attempts to stop the character from moving

Parameters

Name	Description
Priority	Indicates the priority of this command against others
Character	The game object with the Character target

Keywords

`Constant` `Walk` `Run` `To` `Vector` `Character` `Player`

Teleport

```
Characters » Navigation » Teleport
```

Description

Instantaneously moves a Character from its current position to a new one

Parameters

Name	Description
Location	The position and/or rotation where the Character is teleported
Character	The game object with the Character target

Keywords

`Change` `Position` `Location` `Respawn` `Spawn` `Character` `Player`

Player

Player Instructions

- Change Player
- Set Player Input

Change Player

```
Characters » Player » Change Player
```

Description

Changes the Character identified as the Player

Parameters

Name	Description
Character	The Character becomes the new Player character

Keywords

`Character` `Is` `Control`

Set Player Input

```
Characters » Player » Set Player Input
```

Description

Changes how the Player Character reacts to input commands

Parameters

Name	Description
Character	The Character that changes its Player Input behavior
Input	The new input method that the Character starts to listen

Keywords

`Character` `Button` `Control` `Keyboard` `Mouse` `Gamepad` `Joystick`

Properties

Properties Instructions

- Can Jump
- Change Angular Speed
- Change Gravity
- Change Height
- Change Jump Force
- Change Mass
- Change Movement Speed
- Change Radius
- Change Terminal Velocity
- Is Controllable
- Kill Character
- Revive Character

Can Jump

```
Characters » Properties » Can Jump
```

Description

Changes whether the Character is allowed to jump or not

Parameters

Name	Description
Character	The character target
Can Jump	Whether the character is allowed to jump or not

Keywords

`Hop` `Elevate`

Change Angular Speed

```
Characters » Properties » Change Angular Speed
```

Description

Changes the Character's angular speed over time

Parameters

Name	Description
Angular Speed	The target Angular Speed value for the Character, measured in degrees per second
Duration	How long it takes to perform the transition
Easing	The change rate of the parameter over time
Wait to Complete	Whether to wait until the transition is finished
Character	The game object with the Character target

Keywords

`Rotation` `Euler` `Direction` `Face` `Look`

Change Gravity

```
Characters » Properties » Change Gravity
```

Description

Changes the Character's gravity over time

Parameters

Name	Description
Gravity	The target Gravity value for the Character
Duration	How long it takes to perform the transition
Easing	The change rate of the parameter over time
Wait to Complete	Whether to wait until the transition is finished
Character	The game object with the Character target

Keywords

`Space`

Change Height

```
Characters » Properties » Change Height
```

Description

Changes the Character's height over time

Parameters

Name	Description
Height	The target Height value for the Character
Duration	How long it takes to perform the transition
Easing	The change rate of the parameter over time
Wait to Complete	Whether to wait until the transition is finished
Character	The game object with the Character target

Keywords

`Length`

Change Jump Force

```
Characters » Properties » Change Jump Force
```

Description

Changes the Character's jump force over time

Parameters

Name	Description
Jump Force	The target Jump Force value for the Character
Duration	How long it will take to perform the transition
Easing	The change rate of the parameter over time
Wait to Complete	Whether to wait until the transition is finished
Character	The game object with the Character target

Keywords

`Hop` `Build` `Wind` `Fly`

Change Mass

```
Characters » Properties » Change Mass
```

Description

Changes the Character's mass over time

Parameters

Name	Description
Mass	The target Mass value for the Character
Duration	How long it takes to perform the transition
Easing	The change rate of the parameter over time
Wait to Complete	Whether to wait until the transition is finished
Character	The game object with the Character target

Keywords

`Weight`

Change Movement Speed

```
Characters » Properties » Change Movement Speed
```

Description

Changes the Character's maximum speed over time

Parameters

Name	Description
Speed	The target movement Speed value for the Character
Duration	How long it takes to perform the transition
Easing	The change rate of the parameter over time
Wait to Complete	Whether to wait until the transition is finished
Character	The game object with the Character target

Keywords

`Linear` `Walk` `Run` `Jog` `Sprint` `Velocity` `Throttle`

Change Radius

`Characters » Properties » Change Radius`

Description

Changes the Character's radius over time

Parameters

Name	Description
Radius	The target Radius value for the Character
Duration	How long it takes to perform the transition
Easing	The change rate of the parameter over time
Wait to Complete	Whether to wait until the transition is finished
Character	The game object with the Character target

Keywords

`Diameter` `Space` `Fat` `Thin`

Change Terminal Velocity

```
Characters » Properties » Change Terminal Velocity
```

Description

Changes the Character's maximum fall-speed over time. Useful for gliding

Parameters

Name	Description
Terminal Velocity	The target Terminal Velocity value for the Character
Duration	How long it takes to perform the transition
Easing	The change rate of the parameter over time
Wait to Complete	Whether to wait until the transition is finished
Character	The game object with the Character target

Keywords

`Fall` `Glide` `Parachute` `Height`

Is Controllable

```
Characters » Properties » Is Controllable
```

Description

Changes whether the Character (Player) responds using input commands

Parameters

Name	Description
Character	The character target
Is Controllable	Whether the character responds to input commands

Kill Character

```
Characters » Properties » Kill Character
```

Description

Changes the state of the Character to dead

Parameters

Name	Description
Character	The character target

Keywords

`Dead` `Die` `Murder`

Revive Character

```
Characters » Properties » Revive Character
```

Description

Changes the state of the Character to alive

Parameters

Name	Description
Character	The character target

Keywords

`Respawn` `Alive` `Resurrect`

Ragdoll

Ragdoll Instructions

- Recover Ragdoll
- Start Ragdoll

Recover from Ragdoll

```
Characters » Ragdoll » Recover Ragdoll
```

Description

Recovers a Character from the Ragdoll state and stands up

Parameters

Name	Description
Character	The Character game object that recovers from the Ragdoll state

Keywords

`Characters` `Ragdoll` `Recover` `Stand`

Start Ragdoll

```
Characters » Ragdoll » Start Ragdoll
```

Description

Makes a Character enter a ragdoll state

Parameters

Name	Description
Character	The Character game object that changes to a Ragdoll state

Keywords

`Characters` `Ragdoll` `Dead` `Kill` `Die`

Visuals

Visuals Instructions

- Attach Prop
- Change Model
- Put On Skin Mesh
- Remove Prop
- Take Off Skin Mesh

Attach Prop

```
Characters » Visuals » Attach Prop
```

Description

Attaches a prefab Prop onto a Character's bone

Parameters

Name	Description
Character	The character target
Prop	The prefab object that is attached to the character
Bone	Which bone the prop is attached to
Position	Local offset from which the prop is distanced from the bone
Rotation	Local offset from which the prop is rotated from the bone

Keywords

`Characters` `Add` `Grab` `Draw` `Pull` `Take` `Object`

Change Model

```
Characters » Visuals » Change Model
```

Description

Changes the Character current model

Parameters

Name	Description
Character	The character target
Model	The prefab object that replaces the current Character model
Skeleton	Optional parameter that replaces the configuration of volumes
Footstep Sounds	Optional parameter that replaces the current Footstep sounds
Offset	A local offset from the center of the Character

Keywords

`Characters` `Model`

Put on Skin Mesh

```
Characters » Visuals » Put on Skin Mesh
```

Description

Creates a new instance of a skin mesh renderer and puts it on a Character

Parameters

Name	Description
Prefab	Game Object reference with a Skin Mesh Renderer that is instantiated
On Character	Target Character that uses its armature to wear the skin mesh

Keywords

`Renderer` `New` `Game Object` `Armature`

Remove Prop

```
Characters » Visuals » Remove Prop
```

Description

Removes a prefab Prop (if any) from a Character

Parameters

Name	Description
Character	The character target
Prop	The prefab object prop that is removed to the character

Keywords

`Characters` `Detach` `Let` `Sheathe` `Put` `Holster` `Object`

Take off Skin Mesh

```
Characters » Visuals » Take off Skin Mesh
```

Description

Removes an instance of a Skin Mesh from a Character

Parameters

Name	**Description**
Prefab	Game Object reference with a Skin Mesh Renderer that is removed
From Character	Target Character that uses its armature to wear the skin mesh

Keywords

`Renderer` `Game Object` `Armature`

DEBUG

Debug

Instructions

- Beep
- Comment
- Frame Step
- Log Number
- Log Text
- Pause Editor

Beep

```
Debug » Beep
```

Description

Plays the Operative System default 'beep' sound. This is intended for debugging purposes and doesn't do anything on a runtime application

Keywords

`Debug`

Comment

```
Debug » Comment
```

Description

Displays an explanation or annotation in the instructions list. It is intended to make instructions easier for humans to understand

Parameters

Name	Description
Text	The text of the comment

Keywords

`Debug` `Note` `Annotation` `Explanation`

Frame Step

```
Debug » Frame Step
```

Description

Performs a single frame step. It requires the Editor to be paused

Keywords

`Debug`

Debug Number

```
Debug » Log Number
```

Description

Prints a text from a numeric source to the Unity Console

Parameters

Name	Description
Number	The number to log

Keywords

`Debug` `Log` `Print` `Show` `Display` `Test` `Float` `Double` `Decimal` `Integer` `Message`

Debug Text

```
Debug » Log Text
```

Description

Prints a message to the Unity Console

Parameters

Name	Description
Message	The text message to log

Keywords

`Debug` `Log` `Print` `Show` `Display` `Name` `Test` `Message` `String`

Pause Editor

```
Debug » Pause Editor
```

Description

Pauses the Editor. This has no effect on standalone applications

Keywords

`Debug` `Break` `Pause` `Stop`

GAME OBJECTS

Game Objects

Instructions

- Add Component
- Change Layer
- Change Name
- Change Tag
- Destroy
- Disable Component
- Enable Component
- Instantiate
- Remove Component
- Set Active
- Set Game Object
- Toggle Active

Add Component

```
Game Objects » Add Component
```

Description

Adds a component class to the game object

Parameters

Name	Description
Game Object	Target game object

Keywords

`Add` `Append` `MonoBehaviour` `Behaviour` `Script`

Change Layer

```
Game Objects » Change Layer
```

Description

Changes the layer value of a game object

Parameters

Name	Description
Layer	The layer where the game object belongs to
Children Too	Whether to also change the layer of the game object's children or not
Game Object	Target game object

Keywords

`MonoBehaviour` `Behaviour` `Script`

Change Name

```
Game Objects » Change Name
```

Description

Changes the name of a game object

Parameters

Name	Description
Name	The new name assigned to the game object
Game Object	Target game object

Keywords

`MonoBehaviour` `Behaviour` `Script`

Change Tag

```
Game Objects » Change Tag
```

Description

Changes the Tag of a game object

Parameters

Name	Description
Tag	The tag value which the game object belongs to
Game Object	Target game object

Keywords

`MonoBehaviour` `Behaviour` `Script`

Destroy

```
Game Objects » Destroy
```

Description

Destroys a game object scene instance

Parameters

Name	Description
Game Object	Target game object

Keywords

`Remove` `Delete` `Flush` `MonoBehaviour` `Behaviour` `Script`

Disable Component

```
Game Objects » Disable Component
```

Description

Disables a component class from the game object

Parameters

Name	Description
Game Object	Target game object

Keywords

`Deactivate` `Turn` `Off` `MonoBehaviour` `Behaviour` `Script`

Enable Component

```
Game Objects » Enable Component
```

Description

Enables a component class from the game object

Parameters

Name	Description
Game Object	Target game object

Keywords

`Active` `Turn` `On` `MonoBehaviour` `Behaviour` `Script`

Instantiate

```
Game Objects » Instantiate
```

Description

Creates a new instance of a referenced game object

Parameters

Name	Description
Game Object	Game Object reference that is instantiated
Position	The position where the new game object is instantiated
Rotation	The rotation that the new game object has
Save	Optional value where the newly instantiated game object is stored

Keywords

`Create` `New` `Game Object`

Remove Component

```
Game Objects » Remove Component
```

Description

Removes an existing component from the game object

Parameters

Name	Description
Game Object	Target game object

Keywords

`Delete` `Destroy` `MonoBehaviour` `Behaviour` `Script`

Set Active

```
Game Objects » Set Active
```

Description

Changes the state of a game object to active or inactive

Parameters

Name	Description
Game Object	Target game object

Keywords

`Activate` `Deactivate` `Enable` `Disable` `MonoBehaviour` `Behaviour` `Script`

Set Game Object

```
Game Objects » Set Game Object
```

Description

Sets a game object value equal to another one

Parameters

Name	Description
Set	Where the value is set
From	The value that is set

Keywords

`Change` `Instance` `Variable` `Asset`

Toggle Active

```
Game Objects » Toggle Active
```

Description

Toggles the state of a game object to active or to inactive

Parameters

Name	Description
Game Object	Target game object

Keywords

`Activate` `Deactivate` `Enable` `Disable` `Switch` `Swap` `MonoBehaviour` `Behaviour` `Script`

LIGHTS

Lights

Instructions

- Change Color
- Change Intensity

Change Color

```
Lights » Change Color
```

Description

Smoothly changes the color of a Light component

Parameters

Name	Description
Color	The color the Light component starts emitting
Light	The game object with a Light component
Duration	How long it takes to perform the transition
Easing	The change rate of the parameter over time
Wait to Complete	Whether to wait until the transition is finished

Keywords

`Colour` `Hue` `Mood` `RGB` `Light` `Spot` `Sun` `Point` `Strength` `Burn` `Dark`

Change Intensity

```
Lights » Change Intensity
```

Description

Smoothly changes the intensity of a Light component

Parameters

Name	Description
Intensity	The intensity change that the Light component undergoes
Light	The game object with a Light component
Duration	How long it takes to perform the transition
Easing	The change rate of the parameter over time
Wait to Complete	Whether to wait until the transition is finished

Keywords

`Light` `Spot` `Sun` `Point` `Strength` `Burn` `Dark`

LOGIC

Logic

Instructions

- Broadcast Message
- Check Conditions
- Invoke Method
- Raise Signal
- Restart Instructions
- Run Actions
- Run Conditions
- Run Trigger
- Stop Actions
- Stop Conditions
- Stop Trigger

Broadcast Message

```
Logic » Broadcast Message
```

Description

Invokes any method on any component found on the target game object

Parameters

Name	Description
Game Object	The target game object that receives the broadcast message
Message	The name of the method or methods that are called
Send Upwards	If true the message travels from the game object towards the root

Example 1

By default all broadcast messages travel from the target game object and towards all its children. Setting the Send Upwards field to true makes the message travel from the game object towards the root parent

Keywords

`Execute` `Call` `Invoke` `Function`

Check Conditions

```
Logic » Check Conditions
```

Description

If any of the Conditions list is false it early exits and skips the execution of the rest of the Instructions below

Parameters

Name	Description
Conditions	List of Conditions that can evaluate to true or false

Keywords

`Execute` `Call` `Check` `Evaluate`

Invoke Method

```
Logic » Invoke Method
```

Description

Invokes a method from any script attached to a game object

Parameters

Name	Description
Method	The method/function that is called on a game object reference

Keywords

`Execute` `Call` `Invoke` `Function`

Raise Signal

```
Logic » Raise Signal
```

Description

Raises a specific signal, which is captured by other listeners

Parameters

Name	Description
Signal	The signal name risen

Keywords

`Event` `Command` `Fire` `Trigger` `Dispatch` `Execute`

Restart Instructions

```
Logic » Restart Instructions
```

Description

Stops executing the current list of Instructions and starts again from the top

Keywords

`Reset` `Call` `Again`

Run Actions

```
Logic » Run Actions
```

Description

Executes an Actions component object

Parameters

Name	Description
Actions	The Actions object that is executed
Wait Until Complete	If true this instruction waits until the Actions object finishes running

Keywords

`Execute` `Call` `Instruction`

Run Conditions

```
Logic » Run Conditions
```

Description

Executes a Conditions component object

Parameters

Name	Description
Conditions	The Conditions object that is executed
Wait Until Complete	If true this instruction waits until the Conditions object finishes running

Keywords

`Execute` `Call` `Check` `Evaluate`

Run Trigger

```
Logic » Run Trigger
```

Description

Executes a Trigger component object

Parameters

Name	Description
Trigger	The Trigger object that is executed
Wait Until Complete	If true this instruction waits until the Trigger object finishes running

Keywords

`Execute` `Call`

Stop Actions

```
Logic » Stop Actions
```

Description

Stops an Actions component object that is being executed

Parameters

Name	Description
Actions	The Actions object that is stopped

Keywords

`Cancel` `Pause`

Stop Conditions

```
Logic » Stop Conditions
```

Description

Stops a Conditions component object that is being executed

Parameters

Name	Description
Conditions	The Conditions object that is stopped

Keywords

`Cancel` `Pause`

Stop Trigger

```
Logic » Stop Trigger
```

Description

Stops a Trigger component object that is being executed

Parameters

Name	Description
Trigger	The Trigger object that is stopped

Keywords

`Cancel` `Pause`

MATH

Math

Sub Categories

- Arithmetic
- Boolean
- Geometry
- Text

Arithmetic

Arithmetic Instructions

- Absolute Number
- Add Numbers
- Clamp Number
- Cosine
- Divide Numbers
- Multiply Numbers
- Set Number
- Sign Of Number
- Sine
- Subtract Numbers
- Tangent

Absolute Number

```
Math » Arithmetic » Absolute Number
```

Description

Sets a value without its sign

Parameters

Name	Description
Set	Where the value is stored
Number	The input value

Keywords

`Change` `Float` `Integer` `Variable` `Sign` `Positive` `Modulus` `Magnitude`

Add Numbers

```
Math » Arithmetic » Add Numbers
```

Description

Add two values together

Parameters

Name	Description
Set	Where the resulting value is set
Value 1	The first operand of the arithmetic operation
Value 2	The second operand of the arithmetic operation

Keywords

`Sum` `Plus` `Float` `Integer` `Variable`

Clamp Number

```
Math » Arithmetic » Clamp Number
```

Description

Clamps a value between a range defined by two others (inclusive)

Parameters

Name	Description
Set	Where the resulting value is set
Value	The value that is clamped between two others
Minimum	The smallest possible value
Maximum	The largest possible value

Keywords

`Min` `Max` `Negative` `Minus` `Float` `Integer` `Variable`

Cosine

```
Math » Arithmetic » Cosine
```

Description

Sets a value equal the Cosine of a number

Parameters

Name	Description
Set	Where the value is stored
Cosine	The angle input in radians

Keywords

`Change` `Float` `Integer` `Variable`

Divide Numbers

```
Math » Arithmetic » Divide Numbers
```

Description

Performs a division between the first and the second values

Parameters

Name	Description
Set	Where the resulting value is set
Value 1	The first operand of the arithmetic operation
Value 2	The second operand of the arithmetic operation

Keywords

`Fraction` `Float` `Integer` `Variable`

Multiply Numbers

```
Math » Arithmetic » Multiply Numbers
```

Description

Multiplies two values together

Parameters

Name	Description
Set	Where the resulting value is set
Value 1	The first operand of the arithmetic operation
Value 2	The second operand of the arithmetic operation

Keywords

`Product` `Float` `Integer` `Variable`

Set Number

```
Math » Arithmetic » Set Number
```

Description

Sets a value equal to another value

Parameters

Name	Description
Set	Where the value is set
From	The value that is set

Keywords

`Change` `Float` `Integer` `Variable`

Sign of Number

```
Math » Arithmetic » Sign of Number
```

Description

Sets a value equal to -1 if the input number is negative. 1 otherwise

Parameters

Name	Description
Set	Where the value is stored
Number	The input value

Keywords

`Change` `Float` `Integer` `Variable` `Positive` `Negative`

Sine

```
Math » Arithmetic » Sine
```

Description

Sets a value equal the Sine of a number

Parameters

Name	Description
Set	Where the value is stored
Sine	The angle input in radians

Keywords

`Change` `Float` `Integer` `Variable`

Subtract Numbers

```
Math » Arithmetic » Subtract Numbers
```

Description

Subtracts the second value from the first one

Parameters

Name	Description
Set	Where the resulting value is set
Value 1	The first operand of the arithmetic operation
Value 2	The second operand of the arithmetic operation

Keywords

`Rest` `Negative` `Minus` `Float` `Integer` `Variable`

Tangent

```
Math » Arithmetic » Tangent
```

Description

Sets a value equal the Tangent of a number

Parameters

Name	Description
Set	Where the value is stored
Tangent	The angle input in radians

Keywords

`Change` `Float` `Integer` `Variable`

Boolean

Boolean Instructions

- And Bool
- Nand Bool
- Nor Bool
- Or Bool
- Set Bool
- Toggle Bool

AND Bool

```
Math » Boolean » AND Bool
```

Description

Executes an AND operation between to values and saves the result

Parameters

Name	Description
Set	Where the resulting value is set
Value 1	The first operand of the boolean operation
Value 2	The second operand of the boolean operation

Keywords

`Subtract` `Minus` `Variable`

NAND Bool

```
Math » Boolean » NAND Bool
```

Description

Executes a NAND operation between to values and saves the result

Parameters

Name	Description
Set	Where the resulting value is set
Value 1	The first operand of the boolean operation
Value 2	The second operand of the boolean operation

Keywords

`Not` `Negative` `Subtract` `Minus` `Variable`

NOR Bool

```
Math » Boolean » NOR Bool
```

Description

Executes a NOR operation between to values and saves the result

Parameters

Name	Description
Set	Where the resulting value is set
Value 1	The first operand of the boolean operation
Value 2	The second operand of the boolean operation

Keywords

`Not` `Negative` `Sum` `Plus` `Variable`

OR Bool

```
Math » Boolean » OR Bool
```

Description

Executes an OR operation between to values and saves the result

Parameters

Name	Description
Set	Where the resulting value is set
Value 1	The first operand of the boolean operation
Value 2	The second operand of the boolean operation

Keywords

`Sum` `Plus` `Variable`

Set Bool

```
Math » Boolean » Set Bool
```

Description

Sets a boolean value equal to another value

Parameters

Name	Description
Set	Where the value is set
From	The value that is set

Keywords

`Change` `Boolean` `Variable`

Toggle Bool

```
Math » Boolean » Toggle Bool
```

Description

Toggles the value of a Boolean value

Parameters

Name	Description
Set	The boolean value that stores the result
From	The boolean value that is toggled

Keywords

`Change` `Boolean` `Variable` `Not` `Flip` `Switch`

Geometry

Geometry Instructions

- Add Directions
- Add Points
- Clamp
- Cross Product
- Distance
- Dot Product
- Normalize
- Project On Plane
- Reflect On Plane
- Remap Coordinates
- Set Direction
- Set Point
- Subtract Directions
- Subtract Points
- Uniform Scale

Add Directions

```
Math » Geometry » Add Directions
```

Description

Adds two values that represent a direction in space and saves the result

Parameters

Name	Description
Set	Where the resulting value is set
Direction 1	The first operand of the geometric operation that represents a direction
Direction 2	The second operand of the geometric operation that represents a direction

Keywords

`Sum` `Plus` `Position` `Location` `Variable`

Add Points

```
Math » Geometry » Add Points
```

Description

Adds two values that represent a point in space and saves the result

Parameters

Name	Description
Set	Where the resulting value is set
Point 1	The first operand of the geometric operation that represents a point in space
Point 2	The second operand of the geometric operation that represents a point in space

Keywords

`Sum` `Plus` `Position` `Location` `Variable`

Clamp

```
Math » Geometry » Clamp
```

Description

Clamps all components of a Vector3 between two values

Parameters

Name	Description
Set	Dynamic variable where the resulting value is set
Value	The Vector3 value clamped between Minimum and Maximum
Minimum	The minimum value
Maximum	The maximum value

Keywords

`Limit` `Vector3` `Vector2` `Constraint` `Variable`

Cross Product

```
Math » Geometry » Cross Product
```

Description

Calculates the cross product of two direction values and saves the result

Parameters

Name	Description
Set	Where the resulting value is set
Direction 1	The first operand of the geometric operation that represents a direction
Direction 2	The second operand of the geometric operation that represents a direction

Keywords

`Multiply` `Orthogonal` `Perpendicular` `Normal` `Position` `Location` `Variable`

Distance

```
Math » Geometry » Distance
```

Description

Calculates the distance between two points in space and saves the result

Parameters

Name	Description
Set	Where the resulting value is set
Point 1	The first operand of the geometric operation that represents a point in space
Point 2	The second operand of the geometric operation that represents a point in space

Keywords

`Magnitude` `Position` `Location` `Variable`

Dot Product

```
Math » Geometry » Dot Product
```

Description

Calculates the dot product between two directions and saves the result

Parameters

Name	Description
Set	Where the resulting value is set
Direction 1	The first operand of the geometric operation that represents a direction
Direction 2	The second operand of the geometric operation that represents a direction

Keywords

`Direction` `Parallel` `Perpendicular`

Normalize

```
Math » Geometry » Normalize
```

Description

Makes the magnitude of a direction vector equal to 1

Parameters

Name	Description
Set	Dynamic variable where the resulting value is set
From	The direction vector that is normalized

Keywords

`Change` `Vector3` `Vector2` `Unit` `Magnitude` `Variable`

Project on Plane

```
Math » Geometry » Project on Plane
```

Description

projects a direction on a plane defined by a normal vector and saves the result

Parameters

Name	Description
Set	Where the resulting value is set
Direction	The direction vector that is projected on a plane
Plane Normal	The plane represented by the direction of its normal vector

Keywords

`Direction` `Surface` `Sway`

Reflect on Plane

```
Math » Geometry » Reflect on Plane
```

Description

Reflects a direction on a plane defined by a normal vector and saves the result

Parameters

Name	Description
Set	Where the resulting value is set
Direction	The direction vector that is reflected on a plane
Plane Normal	The plane represented by the direction of its normal vector

Keywords

`Direction` `Bounce` `Ricochet` `Snell`

Remap Coordinates

```
Math » Geometry » Remap Coordinates
```

Description

Changes each of the components of a Vector3 value

Parameters

Name	Description
Value	The Vector3 value affected by the operation
X	Where the X coordinate component is remapped
Y	Where the Y coordinate component is remapped
Z	Where the Z coordinate component is remapped

Keywords

`Change` `Vector3` `Vector2` `Component` `Towards` `Look` `Variable` `Axis`

Set Direction

```
Math » Geometry » Set Direction
```

Description

Changes the value of a Vector3 that represents a direction in space

Parameters

Name	Description
Set	Dynamic variable where the resulting value is set
From	The value that is set

Keywords

`Change` `Vector3` `Vector2` `Towards` `Look` `Variable`

Set Point

```
Math » Geometry » Set Point
```

Description

Changes the value of a Vector3 that represents a position in space

Parameters

Name	Description
Set	Dynamic variable where the resulting value is set
From	The value that is set

Keywords

`Change` `Vector3` `Vector2` `Position` `Location` `Variable`

Subtract Directions

```
Math » Geometry » Subtract Directions
```

Description

Subtracts two values that represent a direction in space and saves the result

Parameters

Name	Description
Set	Where the resulting value is set
Direction 1	The first operand of the geometric operation that represents a direction
Direction 2	The second operand of the geometric operation that represents a direction

Keywords

`Minus` `Rest` `Position` `Location` `Variable`

Subtract Points

```
Math » Geometry » Subtract Points
```

Description

Subtracts two values that represent a point in space and saves the result

Parameters

Name	Description
Set	Where the resulting value is set
Point 1	The first operand of the geometric operation that represents a point in space
Point 2	The second operand of the geometric operation that represents a point in space

Keywords

`Rest` `Minus` `Position` `Location` `Variable`

Uniform Scale

```
Math » Geometry » Uniform Scale
```

Description

Multiplies each component of a vector with a decimal

Parameters

Name	Description
Set	Where the resulting value is set
Vector	The first operand of the geometric operation that represents a direction
Value	The second operand of the geometric operation that represents a decimal number

Keywords

`Direction` `Homogeneous` `Multiply` `Product`

Text

Text Instructions

- Join
- Replace
- Set Text
- Substring

Join

```
Math » Text » Join
```

Description

Joins two string values and stores them

Parameters

Name	Description
Text 1	The source of the first text
Text 2	The source of the second text
Set	Where the resulting value is set

Keywords

`Concat` `Concatenate` `Together` `Mix` `String` `Text` `Character`

Replace

```
Math » Text » Replace
```

Description

Replaces all occurrences of a string with another string

Parameters

Name	Description
Text	The source of the text
Old Text	The text replaced
New Text	The text that replaces each occurrence
Set	Where the resulting value is set

Keywords

`Substitute` `Change` `String` `Text` `Character`

Set Text

```
Math » Text » Set Text
```

Description

Changes the value of a string

Parameters

Name	Description
Text	The source of the text
Set	Where the resulting value is set

Keywords

`String` `Text` `Character`

Substring

```
Math » Text » Substring
```

Description

Extracts a substring based on an index and length

Parameters

Name	Description
Text	The source of the text
Index	Starting index of the substring
Length	Amount of characters extracted
Set	Where the resulting value is set

Keywords

`String` `Text` `Character`

PHYSICS 2D

Physics 2D

Instructions

- Add Explosion Force 2D
- Add Force 2D
- Change Mass 2D
- Change Velocity 2D
- Gravity Scale 2D
- Is Kinematic 2D

Add Explosion Force 2D

```
Physics 2D » Add Explosion Force 2D
```

Description

Applies a force to a Rigidbody2D that simulates explosion effects

Parameters

Name	Description
Rigidbody	The game object with a Rigidbody2D component that receives the force
Origin	The position where the explosion originates
Radius	How far the blast reaches
Force	The force of the explosion, which its at its maximum at the origin
Force Mode	How the force is applied

Keywords

`Apply` `Velocity` `Impulse` `Propel` `Push` `Pull` `Boom` `Physics` `Rigidbody`

Add Force 2D

```
Physics 2D » Add Force 2D
```

Description

Adds a force to a game object with a Rigidbody2D

Parameters

Name	Description
Rigidbody	The game object that will receive the force. A Rigidbody2D attached is required
Direction	The direction in which the force will be applied
Force	The amount of force applied
Force Mode	The type of force applied

Keywords

`Apply` `Velocity` `Impulse` `Propel` `Push` `Pull` `Physics` `Rigidbody`

Change Mass 2D

```
Physics 2D » Change Mass 2D
```

Description

Changes the mass of a Rigidbody2D

Parameters

Name	Description
Rigidbody	The game object with a Rigidbody2D attached that will change its mass
Mass	The new mass the game object will be set to have

Keywords

`Weight` `Physics` `Rigidbody`

Change Velocity 2D

```
Physics 2D » Change Velocity 2D
```

Description

Changes the current velocity of a Rigidbody2D

Parameters

Name	Description
Rigidbody	The game object with a Rigidbody2D attached that will change its velocity
Velocity	The velocity the game object will change to

Keywords

`Speed` `Movement` `Physics` `Rigidbody`

Gravity Scale 2D

```
Physics 2D » Gravity Scale 2D
```

Description

Controls whether how gravity affects the Rigidbody2D

Parameters

Name	Description
Rigidbody	The game object with a Rigidbody2D attached that changes its gravity scale
Gravity Scale	The degree to which this object is affected by gravity

Keywords

`Physics` `Rigidbody`

Is Kinematic 2D

```
Physics 2D » Is Kinematic 2D
```

Description

Controls whether physics affects the Rigidbody2D

Parameters

Name	Description
Rigidbody	The game object with a Rigidbody2D attached that changes its kinematic usage
Is Kinematic	If enabled, forces, collisions or joints do not affect the rigidbody anymore

Keywords

`Physics` `Rigidbody`

PHYSICS 3D

Physics 3D

Instructions

- Add Explosion Force 3D
- Add Force 3D
- Change Mass 3D
- Change Velocity 3D
- Is Kinematic 3D
- Use Gravity 3D

Add Explosion Force 3D

```
Physics 3D » Add Explosion Force 3D
```

Description

Applies a force to a Rigidbody that simulates explosion effects

Parameters

Name	Description
Rigidbody	The game object with a Rigidbody component that receives the force
Origin	The position where the explosion originates
Radius	How far the blast reaches
Force	The force of the explosion, which its at its maximum at the origin
Force Mode	How the force is applied

Keywords

`Apply` `Velocity` `Impulse` `Propel` `Push` `Pull` `Boom` `Physics` `Rigidbody`

Add Force 3D

```
Physics 3D » Add Force 3D
```

Description

Adds a force to a game object with a Rigidbody

Parameters

Name	Description
Rigidbody	The game object with a Rigidbody component that receives the force
Direction	The direction in which the force is applied
Force	The amount of force applied
Force Mode	The type of force applied

Keywords

`Apply` `Velocity` `Impulse` `Propel` `Push` `Pull` `Physics` `Rigidbody`

Change Mass 3D

```
Physics 3D » Change Mass 3D
```

Description

Changes the mass of a Rigidbody

Parameters

Name	Description
Rigidbody	The game object with a Rigidbody attached that changes its mass
Mass	The new mass the game object

Keywords

`Weight` `Physics` `Rigidbody`

Change Velocity 3D

```
Physics 3D » Change Velocity 3D
```

Description

Changes the current velocity of a Rigidbody

Parameters

Name	Description
Rigidbody	The game object with a Rigidbody attached that changes its velocity
Velocity	The velocity the game object changes to

Keywords

`Speed` `Movement` `Physics` `Rigidbody`

Is Kinematic 3D

```
Physics 3D » Is Kinematic 3D
```

Description

Controls whether physics affects the Rigidbody

Parameters

Name	Description
Rigidbody	The game object with a Rigidbody attached that changes its kinematic usage
Is Kinematic	If enabled, forces, collisions or joints do not affect the rigidbody anymore

Keywords

`Physics` `Rigidbody`

Use Gravity 3D

```
Physics 3D » Use Gravity 3D
```

Description

Controls whether gravity affects the Rigidbody

Parameters

Name	Description
Rigidbody	The game object with a Rigidbody attached that changes its gravity usage
Use Gravity	If set to false the rigidbody behaves as in outer space

Keywords

`Physics` `Rigidbody`

RENDERER

Renderer

Instructions

- Change Material Color
- Change Material Float
- Change Material Texture
- Change Material

Change Material Color

```
Renderer » Change Material Color
```

Description

Changes over time the Color property of an instantiated material of a Renderer component

Parameters

Name	Description
Property	Name of the property to change
Color	Color target that the instantiated Material turns into
Duration	How long it takes to perform the transition
Easing	The change rate of the transition over time
Wait to Complete	Whether to wait until the transition is finished or not
Renderer	The game object with a Renderer component attached

Keywords

`Set` `Shader` `Hue` `Change`

Change Material Float

```
Renderer » Change Material Float
```

Description

Changes over time the Float property of an instantiated material of a Renderer component

Parameters

Name	Description
Property	Name of the property to change
Float	Decimal target that the instantiated Material's property turns into
Duration	How long it takes to perform the transition
Easing	The change rate of the transition over time
Wait to Complete	Whether to wait until the transition is finished or not
Renderer	The game object with a Renderer component attached

Keywords

`Set` `Shader` `Hue` `Change`

Change Material Texture

```
Renderer » Change Material Texture
```

Description

Changes the main texture of an instantiated material of a Renderer component

Parameters

Name	Description
Texture	Texture that replaces the Renderer's instantiated material
Renderer	The game object with a Renderer component attached

Keywords

`Set` `Shader` `Change`

Change Material

```
Renderer » Change Material
```

Description

Changes instantiated material of a Renderer component

Parameters

Name	Description
Material	Material that is set as the primary type of the Renderer
Renderer	The game object with a Renderer component attached

Keywords

`Set` `Shader` `Texture` `Change`

SCENES

Scenes

Instructions

- Load Scene
- Unload Scene

Load Scene

```
Scenes » Load Scene
```

Description

Loads a new Scene

Parameters

Name	Description
Scene	The scene to be loaded
Mode	Single mode replaces all other scenes. Additive mode loads the scene on top of the others
Async	Loads the scene in the background or freeze the game until its done
Scene Entries	Define the starting location of the player and other characters after loading the scene

Keywords

`Change`

Unload Scene

```
Scenes » Unload Scene
```

Description

Unloads an active scene

Parameters

Name	Description
Scene	The scene to be unloaded

Keywords

`Change` `Remove`

STORAGE

Storage

Instructions

- Delete Game
- Load Game
- Load Latest Game
- Save Game

Delete Game

```
Storage » Delete Game
```

Description

Deletes a previously saved game state

Parameters

Name	Description
Save Slot	Slot number that is erased. Default is 1

Keywords

`Load` `Save` `Delete` `Profile` `Slot` `Game` `Session`

Load Game

```
Storage » Load Game
```

Description

Loads a previously saved state of a game

Parameters

Name	Description
Save Slot	ID number to load the game from. It can range between 1 and 9999

Keywords

`Load` `Save` `Profile` `Slot` `Game` `Session`

Load Latest Game

```
Storage » Load Latest Game
```

Description

Loads the latest previously saved state of a game

Keywords

`Load` `Save` `Last` `Profile` `Game` `Session`

Save Game

```
Storage » Save Game
```

Description

Saves the current state of the game

Parameters

Name	Description
Save Slot	ID number to save the game. It can range between 1 and 9999

Keywords

`Load` `Save` `Profile` `Slot` `Game` `Session`

TESTING

Testing

Instructions

- Instruction Tester

Tester

```
Testing » Instruction Tester
```

Description

Appends a character to a static Chain field. For internal testing use only

Parameters

Name	Description
Character	A character that will be appended to InstructionTester.Chain

Example 1

Note that this Instruction is not accessible through the Inspector to avoid confusing new users. To run the test suit environment, create a new `InstructionList` object and append as many `InstructionTester` instances as your test requires.

```
InstructionList instructions = new InstructionList(
    new InstructionTester('a'),
    new InstructionTester('b'),
    new InstructionTester('c')
);

InstructionTester.Clear();
instructions.Run(null);

Debug.Log(InstructionTester.Chain);
// Prints: 'abc'
```

This instruction is for internal testing only.

TIME

Time

Instructions

- Time Scale
- Wait Frames
- Wait Seconds

Time Scale

```
Time » Time Scale
```

Description

Changes the Time Scale of the game

Parameters

Name	Description
Time Scale	The scale at which time passes. This can be used for slow motion effects
Blend Time	How long it takes to transition from the current time scale to the new one
Layer	Any time scale values using the same Layer is overwritten by this one.

Example 1

Setting a Time Scale of 0 will freeze the game. Useful for pausing the game

Example 2

The resulting Time Scale will be equal to the lowest time scale value between all Layers. For example, if the Time Scale with Layer = 0 has a value of 0.5 (which makes characters move in slow motion), and another Time Scale with Layer = 1 with a value of 0, the resulting Time Scale will be 0

Keywords

`Slow` `Motion` `Bullet` `Time` `Matrix`

Wait Frames

```
Time » Wait Frames
```

Description

Waits a certain amount of frames

Parameters

Name	Description
Frames	The amount of frames to wait

Example 1

This instruction is particularly useful in cases where you want to control the order of execution of two Actions. For example, imagine there are two Triggers executing at the same time, but you want to execute the instructions associated with one after the execution of the other one. You can use the 'Wait Frames' instruction to defer its execution 1 frame so the other one has had time to complete its own execution

Keywords

`Wait` `Time` `Frames` `Yield`

Wait Seconds

```
Time » Wait Seconds
```

Description

Waits a certain amount of seconds

Parameters

Name	Description
Seconds	The amount of seconds to wait

Keywords

`Wait` `Time` `Seconds` `Minutes` `Cooldown` `Timeout` `Yield`

TRANSFORMS

Transforms

Instructions

- Change Position
- Change Rotation
- Change Scale
- Clear Parent
- Set Parent

Change Position

```
Transforms » Change Position
```

Description

Changes the position of a game object over time

Parameters

Name	Description
Position	The desired position of the game object
Space	If the transformation occurs in local or world space
Duration	How long it takes to perform the transition
Easing	The change rate of the translation over time
Wait to Complete	Whether to wait until the translation is finished or not
Transform	The Transform of the game object

Keywords

`Location` `Translate` `Move` `Displace`

Change Rotation

```
Transforms » Change Rotation
```

Description

Changes the rotation of a game object over time

Parameters

Name	Description
Rotation	The desired rotation of the game object
Space	If the transformation occurs in local or world space
Duration	How long it takes to perform the transition
Easing	The change rate of the rotation over time
Wait to Complete	Whether to wait until the rotation is finished or not
Transform	The Transform of the game object

Keywords

`Rotate` `Angle` `Euler` `Tilt` `Pitch` `Yaw` `Roll`

Change Scale

```
Transforms » Change Scale
```

Description

Changes the local scale of a game object over time

Parameters

Name	Description
Scale	The desired scale of the game object
Duration	How long it takes to perform the transition
Easing	The change rate of the scaling over time
Wait to Complete	Whether to wait until the scaling is finished or not
Transform	The Transform of the game object

Keywords

`Size` `Resize` `Grow` `Reduce` `Small` `Big`

Clear Parent

```
Transforms » Clear Parent
```

Description

Clears the parent of a game object

Parameters

Name	Description
Transform	The Transform of the game object

Keywords

`Child` `Children` `Hierarchy` `Orphan`

Set Parent

```
Transforms » Set Parent
```

Description

Changes the parent of a game object

Parameters

Name	Description
Parent	The game object that becomes the parent
Transform	The Transform of the game object

Keywords

`Child` `Children` `Hierarchy` `Hang` `Inherit`

UI

Ui

Instructions

- Canvas Group Alpha
- Canvas Group Block Raycasts
- Canvas Group Interactable
- Change Color
- Change Dropdown
- Change Font Size
- Change Image
- Change Slider
- Change Text
- Change Toggle
- Focus On

Canvas Group Alpha

```
UI » Canvas Group Alpha
```

Description

Changes the opacity of the Canvas Group and affects all of its children

Parameters

Name	Description
Canvas Group	The Canvas Group component that changes its value
Alpha	The new opacity value transformation of the Canvas Group
Duration	How long it takes to perform the transition
Easing	The change rate of the parameter over time
Wait to Complete	Whether to wait until the transition is finished

Canvas Group Block Raycasts

```
UI » Canvas Group Block Raycasts
```

Description

Changes whether the Canvas Group blocks raycasts or not

Parameters

Name	Description
Canvas Group	The Canvas Group component that changes its value
Block Raycasts	If true, the canvas group and its children block raycasts

Canvas Group Interactable

```
UI » Canvas Group Interactable
```

Description

Changes the interactable value of a Canvas Group component

Parameters

Name	Description
Canvas Group	The Canvas Group component that changes its value
Interactable	The on/off state value

Change Color

```
UI » Change Color
```

Description

Changes the color of a Graphic component

Parameters

Name	Description
Graphic	The Graphic component that changes its tint color
Color	The new Color

Change Dropdown

```
UI » Change Dropdown
```

Description

Changes the value of a Dropdown or Text Mesh Pro Dropdown component

Parameters

Name	Description
Text	The Text or Text Mesh Pro component that changes its value
Index	The new index value of the Dropdown

Change Font Size

```
UI » Change Font Size
```

Description

Changes the size of the Text or Text Mesh Pro component content

Parameters

Name	Description
Text	The Text or Text Mesh Pro component that changes its font size
Size	The new text size, in pixels

Keywords

`Text`

Change Image

```
UI » Change Image
```

Description

Changes the Sprite of an Image component

Parameters

Name	Description
Override Sprite	If the Sprite replaced is the original or the overriden
Image	The Image component that changes its sprite value
Sprite	The new Sprite reference

Change Slider

```
UI » Change Slider
```

Description

Changes the value of a Slider component

Parameters

Name	Description
Slider	The Slider component that changes its value
Value	The new value set

Change Text

```
UI » Change Text
```

Description

Changes the value of a Text or Text Mesh Pro component

Parameters

Name	Description
Text	The Text or Text Mesh Pro component that changes its value
Value	The new value set

Change Toggle

```
UI » Change Toggle
```

Description

Changes the value of a Toggle component

Parameters

Name	Description
Toggle	The Toggle component that changes its value
Value	The new value set

Focus On

```
UI » Focus On
```

Description

Focuses on a specific UI component

Parameters

Name	Description
Focus On	The UI component that takes focus

Keywords

`Text`

VARIABLES

Variables

Instructions

- Clear List
- Collect Characters
- Collect Markers
- Filter List
- Loop List
- Remove From List
- Reverse List
- Shuffle List
- Sort List Alphabetically
- Sort List By Distance

Clear List

```
Variables » Clear List
```

Description

Removes all elements of a given Local or Global List Variables

Parameters

Name	Description
List Variable	Local List or Global List which elements are removed

Keywords

`Clean` `Remove` `Delete` `Destroy` `Size` `Array` `List` `Variables`

Collect Characters

```
Variables » Collect Characters
```

Description

Collects all Characters that within a certain radius of a position

Parameters

Name	Description
Origin	The position where the rest of the game objects are collected
Max Radius	How far from the Origin the game objects are collected
Min Radius	How far from the Origin game objects start to be collected
Store In	List where the collected game objects are saved
Filter	Checks a set of Conditions with each collected game object

Example 1

Note that in most cases it is not desirable to set the Min Radius to 0. Doing so will also collect game objects at a distance of 0 from the Origin. For example, if we want to collect all enemies around the Player and we set a Min Radius of 0, the Player will also be collected because it's a Character at a distance 0 from himself

Keywords

`Gather` `Get` `Set` `Array` `List` `Variables`

Collect Markers

```
Variables » Collect Markers
```

Description

Collects all Markers that within a certain radius of a position

Parameters

Name	Description
Origin	The position where the rest of the game objects are collected
Max Radius	How far from the Origin the game objects are collected
Min Radius	How far from the Origin game objects start to be collected
Store In	List where the collected game objects are saved
Filter	Checks a set of Conditions with each collected game object

Example 1

Note that in most cases it is not desirable to set the Min Radius to 0. Doing so will also collect game objects at a distance of 0 from the Origin. For example, if we want to collect all enemies around the Player and we set a Min Radius of 0, the Player will also be collected because it's a Character at a distance 0 from himself

Keywords

`Gather` `Get` `Set` `Array` `List` `Variables`

Filter List

```
Variables » Filter List
```

Description

Checks Conditions against each element of a list and removes it if the Condition is not true

Parameters

Name	Description
List Variable	Local List or Global List which elements are filtered
Filter	Checks a set of Conditions with each collected game object and removes the element if the Condition is not true

Example 1

The Filter field runs the Conditions list for each element in a Local List Variables or Global List Variables. It sets as the 'Target' value the currently examined game object. For example, filtering by the tag name 'Enemy' can be done using the 'Tag' Condition and comparing the field 'Target' with the string 'Enemy'. All game objects that are not tagged as 'Enemy' are removed

Keywords

`Remove` `Pick` `Select` `Array` `List` `Variables`

Loop List

```
Variables » Loop List
```

Description

Loops a Game Object List Variables and executes an Actions component for each value

Parameters

Name	Description
List Variable	Local List or Global List which elements are iterated
Actions	The Actions component executed for each element in the list. The Target argument of any Instruction contains the object inspected

Keywords

`Iterate` `Cycle` `Every` `All` `Stack`

Remove from List

```
Variables » Remove from List
```

Description

Deletes an elements from a given Local or Global List Variables

Parameters

Name	Description
List Variable	Local List or Global List which elements are removed

Keywords

`Delete` `Destroy` `Size` `Array` `List` `Variables`

Reverse List

```
Variables » Reverse List
```

Description

Reorders the elements of a list so the first ones become the last ones

Parameters

Name	Description
List Variable	Local List or Global List which elements are reversed

Keywords

`Invert` `Order` `Sort` `Array` `List` `Variables`

Shuffle List

```
Variables » Shuffle List
```

Description

Randomly shuffles the position of each element on a List Variable

Parameters

Name	Description
List Variable	Local List or Global List which elements are shuffled

Keywords

`Randomize` `Sort` `Array` `List` `Variables`

Sort List Alphabetically

```
Variables » Sort List Alphabetically
```

Description

Sorts the List Variable elements based on their alphabet distance

Parameters

Name	Description
List Variable	Local List or Global List which elements are sorted
Order	Sort alphabetically ascending or descending
Ignore Case	Whether the string comparison should ignore upper/lower case

Keywords

`Order` `Organize` `Array` `List` `Variables`

Sort List by Distance

```
Variables » Sort List by Distance
```

Description

Sorts the List Variable elements based on their distance to a given position

Parameters

Name	Description
List Variable	Local List or Global List which elements are sorted
Position	The reference position that is used to measure the sorting distance
Order	From Closest to Farthest puts the closest elements to the Position first

Keywords

`Order` `Organize` `Array` `List` `Variables`

Custom Instructions

Game Creator allows to very easily create custom **Instructions** and use them along with the rest.

Programming Knowledge Required

This section assumes you have some programming knowledge. If you don't know how to code you might be interested in checking out the Game Creator Hub page. Programmers altrusitically create custom **Instructions** for others to download and use in their project.

CREATING AN INSTRUCTION

The easiest way to create an **Instruction** C# script is to right click on your *Project* panel and select *Create* → *Game Creator* → *Developer* → *C# Instruction*. This will create a template script with the boilerplate structure of an Instruction:

```
using System;
using System.Threading.Tasks;
using GameCreator.Runtime.Common;
using GameCreator.Runtime.VisualScripting;

[Serializable]
public class MyInstruction : Instruction
{
    protected override Task Run(Args args)
    {
        // Your code here...
        return DefaultResult;
    }
}
```

Anatomy of an Instruction

An **Instruction** is a class that inherits from the `Instruction` super class. The abstract `Run(...)` method is the entry point of an **Instruction**'s execution, which is automatically called when it's this instruction's time to be executed.

The `Run(...)` method has a single parameter of type `Args`, which is a helper class that contains a reference to the game object that initiated the call (`args.Self`) and the targeted game object (`args.Target`), if any.

Yielding in Time

Most instruction will be executed in a single frame. However, some instructions might require to put the execution on hold for a certain amount of time, before resuming the execition. The most simple example is with the "Wait for Seconds" instruction, which pauses the execution for a few seconds before resuming.

The `Instruction` super class contains a collection of methods that helps with time management.

Async/Await

Instructions use the async/await methodology to manage the flow of an instruction over the course of time. Using the `await` symbol requires the `Run()` method to have the `async` symbol on its method definition:

```
protected override async Task Run(Args args)
{ }
```

NextFrame

The `NextFrame()` methods pauses the execution of the Instruction for a single frame, then resumes.

```
protected override async Task Run(Args args)
{
    await this.NextFrame();
}
```

Time

The `Time(float time)` method pauses the execution of an Instruction for a certain amount of time. The `time` parameter is in seconds.

```
protected override async Task Run(Args args)
{
    await this.Time(5f);
}
```

While

The `While(Func<bool> function)` method pauses the execution of an Instruction for as long as the result of the method passed as a parameter returns true. This method is executed every frame and the execution will resume as soon as it returns `false`.

```
protected override async Task Run(Args args)
{
    await this.While(() => this.IsPlayerMoving());
}
```

Until

The `Until(Func<bool> function)` method pauses the execution of an Instruction for as long as the result of the method passed as a parameter returns true. This method is executed every frame and the execution will resume as soon as it returns `true`.

```
protected override async Task Run(Args args)
{
    await this.Until(() => this.PlayerHasReachedDestination());
}
```

Decoration & Documentation

It is highly recommended to document and decorate the **Instruction** so it's easier to find and use. It is done using class-type attributes that inform **Game Creator** of the quirks of this particular instruction.

For example, to set the title of an instruction to "Hello World", use the `[Title(string name)]` attribute right above the class definition:

```
using System;
using System.Threading.Tasks;
using GameCreator.Runtime.Common;
using GameCreator.Runtime.VisualScripting;

[Title("Hello World")]
[Serializable]
public class MyInstruction : Instruction
{
    protected override Task Run(Args args)
    {
        // ...
    }
}
```

Title

The title of the Instruction. If this attribute is not provided, the title will be a beautified version of the class name.

```
[Title("Title of Instruction")]
```

Description

A description of what the Instruction does. This is both used in the floating window documentation, as well as the description text when uploading an Instruction to the Game Creator Hub.

```
[Description("Lorem Ipsum dolor etiam porta sem magna mollis")]
```

Image

The `[Image(...)]` attribute changes the default icon of the Instruction for one of the default ones. It consists of 2 parameters:

- **Icon [`Type`]:** a Type class of an `IIcon` derived class. **Game Creator** comes packed with a lot of icons although you can also create your own.
- **Color [`Color`]:** The color of the icon. Uses Unity's `Color` class.

For example, one of the icons included is the "Solid Cube" icon. To display a red solid cube as the icon of the instruction, use the following attribute:

```
[Image(typeof(IconCubeSolid), Color.red)]
```

Category

A sequence of sub-categories organized using the slash (`/`) character. This attribute helps keep the Instructions organized when the Instructions list dropdown is displayed.

```
[Category("Category/Sub Category/Name")]
```

The example above will display the Instruction under the sub directory *Category* *Sub Category* *.Name*

Version

A semantic version to keep track of the development of this Instruction. It's important to note that when updating an Instruction to the Game Creator Hub, the version number must always be higher than the one on the server.

The semantic version follows the standard *Major Version*, *Minor Version*, *Patch Version*. To know more about how semantic versioning works, read the following page: https://semver.org.

```
[Version(1, 5, 3)]
```

Parameters

When an Instruction has exposed fields in the Inspector, it's a good idea to document what these do. You can add as many `[Parameter(name, description)]` attributes as exposed fields has the Instruction.

For example, if the Instruction has these two fields:

```
public bool waitForTime = true;
public float duration = 5f;
```

You can document those fields adding:

```
[Parameter("Wait For Time", "Whether to wait or not")]
[Parameter("Duration", "The amount of seconds to wait")]
```

Keywords

Keywords are strings that help the fuzzy finder more easily search for an instruction. For example, the "Change Position" instruction doesn't reference the word "move" or "translate" anywhere in its documentation. However, these words are very likely to reference this instruction when the user types them in the search box.

```
[Keywords("Move", "Translate")]
```

Example

The Example attribute allows to display a text as an example of use of this Instruction. There can be more than one `[Example(...)]` attribute per instruction. This is particularly useful when uploading instructions on the Game Creator Hub.

Markdown

It is recommended to use Markdown notation when writing examples

```
[Example("Sed posuere consectetur est at lobortis)]
```

Multiple Lines

You can use the `@` character in front of a string to break the example text in multiple lines. To create a new paragraph, simply add two new lines. For example:

```
[Example(@"
    This is the first paragraph.
    This is also in the first paragraph, right after the previous sentence

    This line is part of a new paragraph.
)]
```

Dependency

This attribute is optional and only used in the Game Creator Hub. If this Instruction uses some particular feature of a specific module, it will first check if the user downloading this instruction has that module installed. If it does not, it will display an error message and forbid downloading it. This is useful to avoid throwing programming errors.

The `[Dependency(...)]` attribute consists of 4 parameters:

- **Module ID:** For example, the ID of the Inventory module is `gamecreator.inventory`.
- **Major Version:** The minimum major version of the dependency module.
- **Minor Version:** The minimum minor version of the dependency module.
- **Patch Version:** The minimum patch version of the dependency module.

```
[Dependency("gamecreator.inventory", 1, 5, 2)]
```

1.5.3 Triggers

Triggers

Triggers are components attached to game objects that listen to events that happen on the scene and react by executing a sequence of instructions.

Triggers

Example

In the image above, the **Trigger** is listening for the *Space* keyboard key to be pressed down. As soon as that happens, it calls the instructions list from below, which prints the message "Space key pressed!"

CREATING A TRIGGER

Right click on the *Hierarchy* panel and select *Game Creator* *Visual Scripting* *Trigger*. A game object named 'Trigger' will appear in the scene with a component of the same name.

Alternatively you can also add the **Trigger** component to any game object clicking on the Inspector's Add Component button and searching for Trigger.

Deleting Triggers

To delete a Trigger component, simply click on the component's little cog button and select "Remove Component" from the dropdown menu.

CHANGING THE EVENT

Triggers listen to very specific events, chosen by the user. To change the type of **Event** a Trigger listens, click on the event name and a dropdown menu will appear. Navigate it using the mouse or searching for a specific event in the seach box field.

Change Trigger Event

INSTRUCTIONS

The **Instructions** list that appear below work exactly the same was the **Actions** component. For more information about this component, visit the Actions page.

Events

EVENTS

Sub Categories

- Audio
- Characters
- Input
- Interactive
- Lifecycle
- Logic
- Physics
- Storage
- Ui
- Variables

AUDIO

Audio

Events

- On Change Ambient Volume
- On Change Master Volume
- On Change Sound Effects Volume
- On Change Speech Volume
- On Change Ui Volume

On Change Ambient Volume

```
Audio » On Change Ambient Volume
```

Description

Executed when the Ambient Volume is changed

Keywords

`Audio` `Sound` `Level`

On Change Master Volume

```
Audio » On Change Master Volume
```

Description

Executed when the Master Volume is changed

Keywords

`Audio` `Sound` `Level`

On Change Sound Effects Volume

```
Audio » On Change Sound Effects Volume
```

Description

Executed when the Sound Effects Volume is changed

Keywords

`Audio` `Sound` `Level`

On Change Speech Volume

```
Audio » On Change Speech Volume
```

Description

Executed when the Speech Volume is changed

Keywords

`Audio` `Sound` `Level`

On Change UI Volume

```
Audio » On Change UI Volume
```

Description

Executed when the UI Volume is changed

Keywords

`Audio` `Sound` `Level`

CHARACTERS

Characters

Sub Categories

- Navigation
- Ragdoll

Events

- On Become Npc
- On Become Player
- On Change Model
- On Die
- On Revive

On Become NPC

```
Characters » On Become NPC
```

Description

Executed when a character that is a Player becomes an NPC

On Become Player

```
Characters » On Become Player
```

Description

Executed when a character becomes the Player

On Change Model

```
Characters » On Change Model
```

Description

Executed when a character changes its model

On Die

```
Characters » On Die
```

Description

Executed when the character dies

On Revive

```
Characters » On Revive
```

Description

Executed when a dead character revives

Keywords

`Resurrect` `Respawn`

Navigation

Navigation Events

- On Jump
- On Land
- On Navlink Enter
- On Navlink Exit
- On Step

On Jump

```
Characters » Navigation » On Jump
```

Description

Executed every time the character performs a jump

On Land

```
Characters » Navigation » On Land
```

Description

Executed every time the character lands on the ground

On NavLink Enter

```
Characters » Navigation » On NavLink Enter
```

Description

Executed when a character enters a navigation mesh Off Mesh Link

On NavLink Exit

```
Characters » Navigation » On NavLink Exit
```

Description

Executed when a character exists a navigation mesh Off Mesh Link

On Step

```
Characters » Navigation » On Step
```

Description

Executed every time the character takes a step

Keywords

`Footstep` `Foot` `Feet` `Ground`

Ragdoll

Ragdoll Events

- On Recover Ragdoll
- On Start Ragdoll

On Recover Ragdoll

```
Characters » Ragdoll » On Recover Ragdoll
```

Description

Executed when the character recovers from the ragdoll mode

On Start Ragdoll

```
Characters » Ragdoll » On Start Ragdoll
```

Description

Executed when the character enters the ragdoll mode

INPUT

Input

Events

- On Cursor Click
- On Input

On Cursor Click

```
Input » On Cursor Click
```

Description

Detects when the cursor clicks this game object

Parameters

Name	Description
Button	The mouse button to detect
Min Distance	If set to None, the mouse input acts globally. If set to Game Object, the event only fires if the target object is within a certain radius

Keywords

`Down` `Mouse` `Button` `Hover` `Left` `Middle` `Right`

On Input

```
Input » On Input
```

Description

Detects when a button is interacted with

Parameters

Name	Description
Button	The button that triggers the event
Min Distance	If set to None, the input acts globally. If set to Game Object, the event only fires if the target object is within the specified radius

Keywords

`Down` `Up` `Press` `Release` `Keyboard` `Mouse` `Button` `Gamepad` `Controller` `Joystick`

INTERACTIVE

Interactive

Events

- On Blur
- On Focus
- On Interact

On Blur

```
Interactive » On Blur
```

Description

Executed when the Character loses focus on this Interactive object

On Focus

```
Interactive » On Focus
```

Description

Executed when the Character focuses on this Interactive object

On Interact

```
Interactive » On Interact
```

Description

Executed when a Character interacts with this Trigger

Parameters

Name	Description
Use Raycast	Checks if there is something between the character and the Trigger

Example 1

The 'Use Raycast' option checks if there is no other collider between the Character and the Trigger

LIFECYCLE

Lifecycle

Events

- On Become Invisible
- On Become Visible
- On Disable
- On Enable
- On Interval
- On Invoke
- On Late Update
- On Start
- On Update

On Become Invisible

```
Lifecycle » On Become Invisible
```

Description

Executed when the game object it is attached to is no longer visible by any camera

Keywords

`Hide` `Disappear`

On Become Visible

```
Lifecycle » On Become Visible
```

Description

Executed when the game object it is attached to becomes visible to any camera

Keywords

`Show` `Render` `Appear`

On Disable

```
Lifecycle » On Disable
```

Description

Executed when the game object it is attached to becomes disabled or inactive

Keywords

`Inactive` `Active` `Enable`

On Enable

```
Lifecycle » On Enable
```

Description

Executed when the game object it is attached to becomes enabled and active

Keywords

`Active` `Disable` `Inactive`

On Interval

```
Lifecycle » On Interval
```

Description

Executes after an amount of seconds have passed between each call

Parameters

Name	Description
Time Mode	The time scale in which the interval is calculated
Interval	Amount of seconds between each iteration

Keywords

`Loop` `Tick` `Continuous` `FPS`

On Invoke

```
Lifecycle » On Invoke
```

Description

Executed only when calling its Invoke() method

Keywords

`Script` `Manual`

On Late Update

```
Lifecycle » On Late Update
```

Description

Executed every frame after all On Update events are fired, as long as the game object is enabled

Keywords

`Loop` `Tick` `Continuous`

On Start

```
Lifecycle » On Start
```

Description

Executed on the frame when the game object is enabled for the first time

Keywords

`Initialize`

On Update

```
Lifecycle » On Update
```

Description

Executed every frame as long as the game object is enabled

Keywords

`Loop` `Tick` `Continuous`

LOGIC

Logic

Events

- On Hotspot Activate
- On Hotspot Deactivate
- On Receive Signal

On Hotspot Activate

```
Logic » On Hotspot Activate
```

Description

Executed when its associated Hotspot is activated

Keywords

`Spot`

On Hotspot Deactivate

```
Logic » On Hotspot Deactivate
```

Description

Executed when its associated Hotspot is deactivated

Keywords

`Spot`

On Receive Signal

```
Logic » On Receive Signal
```

Description

Executed when receiving a specific signal name from the dispatcher

Keywords

`Event` `Command` `Fire` `Trigger` `Dispatch` `Execute`

PHYSICS

Physics

Events

- On Collide
- On Trigger Enter Tag
- On Trigger Enter
- On Trigger Exit Tag
- On Trigger Exit
- On Trigger Stay

On Collide

```
Physics » On Collide
```

Description

Executed when the Trigger collides with a game object

Keywords

`Crash` `Touch` `Bump` `Collision`

On Trigger Enter Tag

```
Physics » On Trigger Enter Tag
```

Description

Executed when a game object with a Tag enters the Trigger collider

Parameters

Name	Description
Tag	A string that represents a group of game objects

Keywords

`Pass` `Through` `Touch` `Collision` `Collide`

On Trigger Enter

```
Physics » On Trigger Enter
```

Description

Executed when a game object enters the Trigger collider

Keywords

`Pass` `Through` `Touch` `Collision` `Collide`

On Trigger Exit Tag

```
Physics » On Trigger Exit Tag
```

Description

Executed when a game object with a Tag exists the Trigger collider

Parameters

Name	Description
Tag	A string that represents a group of game objects

Keywords

`Pass` `Through` `Touch` `Collision` `Collide`

On Trigger Exit

```
Physics » On Trigger Exit
```

Description

Executed when a game object leaves the Trigger collider

Keywords

`Leave` `Through` `Touch` `Collision` `Collide`

On Trigger Stay

```
Physics » On Trigger Stay
```

Description

Executed while a game object stays inside the Trigger collider

Keywords

`Pass` `Through` `Touch` `Collision` `Collide`

STORAGE

Storage

Events

- On Save

On Save

```
Storage » On Save
```

Description

Executed when the game is saved

Keywords

`Show` `Render` `Appear`

UI

Ui

Events

- On Deselect
- On Hover Enter
- On Hover Exit
- On Select

On Deselect

```
UI » On Deselect
```

Description

Executed when the UI element is deselected

Keywords

`Mouse` `Choose` `Focus` `Pick` `Pointer`

On Hover Enter

```
UI » On Hover Enter
```

Description

Executed when the pointer hovers the UI element

Keywords

`Mouse` `Over` `Pointer`

On Hover Exit

```
UI » On Hover Exit
```

Description

Executed when the pointer exits the hovered UI element

Keywords

`Mouse` `Over` `Pointer`

On Select

```
UI » On Select
```

Description

Executed when the UI element is selected

Keywords

`Mouse` `Choose` `Focus` `Pick` `Pointer`

VARIABLES

Variables

Events

- On Global List Variable Change
- On Global Name Variable Change
- On Local List Variable Change
- On Local Name Variable Change

On Global List Variable Change

```
Variables » On Global List Variable Change
```

Description

Executed when the Global List Variable is modified

On Global Name Variable Change

```
Variables » On Global Name Variable Change
```

Description

Executed when the Global Name Variable is modified

On Local List Variable Change

```
Variables » On Local List Variable Change
```

Description

Executed when the Local List Variable is modified

On Local Name Variable Change

```
Variables » On Local Name Variable Change
```

Description

Executed when the Local Name Variable is modified

Custom Events

Game Creator allows to create custom **Events** that listen to events and react accordingly. Note that it's up to the programmer to determine the most performant way to detect an event.

Programming Knowledge Required

This section assumes you have some programming knowledge. If you don't know how to code you might be interested in checking out the Game Creator Hub page. Programmers altrusitically create custom **Events** for others to download and use in their project.

CREATING AN EVENT

The easiest way to create an **Event** C# script is to right click on your *Project* panel and select _Create Game Creator Developer C# Event. This will create a template script with the boilerplate structure:

```
using System;
using GameCreator.Runtime.VisualScripting;

[Serializable]
public class MyEvent : Event
{
    protected override void OnStart(Trigger trigger)
    {
        base.OnStart(trigger);
        _ = trigger.Execute(this.Self);
    }
}
```

Anatomy of an Event

An **Event** is a class that inherits from the `Event` super class. It contains a large collection of virtual methods to inherit from, which are very similar to `MonoBeheaviour` methods.

Example

For example, to detect when the **Trigger** component is initialized, you can override the `OnAwake()` or the `OnStart()` methods. For a full list of all available methods to override, check the *Event.cs* script file.

All methods come with a `trigger` parameter, which references the **Trigger** component that owns this **Event**.

Fire an Event

Once you have setup the necessary code to detect an event, it's time to tell the **Trigger** to exeecute the specified reaction. This is done using the `Execute(target)` method from the Trigger component:

```
trigger.Execute(this.Self);
```

Async/Await

Note that the `Execute(...)` method returns an async task so the code can wait until the reaction completes before resuming the execution. Most of the times however, you will prefer to fire and forget about the reaction. In those cases you can use the discard (`_`) modifier:

```
_ = trigger.Execute(this.Self);
```

On the other hand, if you want to wait until the instruction sequence has completed, you can await for the resolution of these:

```
await trigger.Execute(this.Self);
```

The `Execute(target)` method allows to pass a game object parameter, which is the *Target* game object of the instructions list. For example, if the **Event** you are programming is trying to detect the collision between 2 colliders, the `target` should reference the other collider game object.

Decoration & Documentation

It is highly recommended to document and decorate the **Event** so it's easier to find and use. It is done using class-type attributes that inform **Game Creator** of the quirks of this particular event.

For example, to set the title of an Event to "Hello World", use the `[Title(string name)]` attribute right above the class definition:

```
using System;
using GameCreator.Runtime.VisualScripting;

[Title("Hello World")]
[Serializable]
public class MyEvent : Event
{
    protected override void OnStart(Trigger trigger)
    {
        base.OnStart(trigger);
        _ = trigger.Execute(this.Self);
    }
}
```

Title

The title of the Event. If this attribute is not provided, the title will be a beautified version of the class name.

```
[Title("Title of Event")]
```

Description

A description of what the Event does. This is used as the description text when uploading an Event to the Game Creator Hub.

```
[Description("Lorem Ipsum dolor etiam porta sem magna mollis")]
```

Image

The `[Image(...)]` attribute changes the default icon of the Event for one of the default ones. It consists of 2 parameters:

- **Icon [`Type`]:** a Type class of an `IIcon` derived class. **Game Creator** comes packed with a lot of icons although you can also create your own.
- **Color [`Color`]:** The color of the icon. Uses Unity's `Color` class.

For example, one of the icons included is the "Solid Cube" icon. To display a red solid cube as the icon of the event, use the following attribute:

```
[Image(typeof(IconCubeSolid), Color.red)]
```

Category

A sequence of sub-categories organized using the slash (`/`) character. This attribute helps keep the Events organized when the dropdown list is displayed.

```
[Category("Category/Sub Category/Name")]
```

The example above will display the Event under the sub directory *Category* *Sub Category* *.Name*

Version

A semantic version to keep track of the development of this Event. It's important to note that when updating an Event to the Game Creator Hub, the version number must always be higher than the one on the server.

The semmantic version follows the standard *Major Version*, *Minor Version*, *Patch Version*. To know more about how semantic versioning works, read the following page: https://semver.org.

```
[Version(1, 5, 3)]
```

Parameters

When an Event has exposed fields in the Inspector, it's a good idea to document what these do. You can add as many `[Parameter(name, description)]` attributes as exposed fields has the Event.

For example, if the Event has these two fields:

```
public bool checkDistance = true;
public float distance = 5f;
```

You can document those fields adding:

```
[Parameter("Check Distance", "Whether to check the distance or not")]
[Parameter("Distance", "The maximum distance between targets")]
```

Keywords

Keywords are strings that help the fuzzy finder more easily search for an Event. For example, the "On Become Visible" event doesn't reference the word "hide" anywhere in its documentation. However, these words are very likely to reference this event when the user types them in the search box.

```
[Keywords("Hide")]
```

Example

The Example attribute allows to display a text as an example of use of this Event. There can be more than one `[Example(...)]` attribute per event. This is particularly useful when uploading events on the Game Creator Hub.

Markdown

It is recommended to use Markdown notation when writing examples

```
[Example("Sed posuere consectetur est at lobortis)]
```

Multiple Lines

You can use the `@` character in front of a string to break the example text in multiple lines. To create a new paragraph, simply add two new lines. For example:

```
[Example(@"
    This is the first paragraph.
    This is also in the first paragraph, right after the previous sentence

    This line is part of a new paragraph.
)]
```

Dependency

This attribute is optional and only used in the Game Creator Hub. If this Event uses some particular feature of a specific module, it will first check if the user downloading this event has that module installed. If it does not, it will display an error message and forbid downloading it. This is useful to avoid throwing programming errors.

The `[Dependency(...)]` attribute consists of 4 parameters:

- **Module ID:** For example, the ID of the Inventory module is `gamecreator.inventory`.
- **Major Version:** The minimum major version of the dependency module.
- **Minor Version:** The minimum minor version of the dependency module.
- **Patch Version:** The minimum patch version of the dependency module.

```
[Dependency("gamecreator.inventory", 1, 5, 2)]
```

1.5.4 Conditions

Conditions

Conditions are components attached to game objects that, when executed, start checking the conditions in each **Branch**, from top to bottom. If all the **Conditions** of a branch return success, then the **Instructions** associated to that branch are executed, and stops checking any further.

If any of the **Conditions** of a **Branch** returns `false`, it skips to the next branch.

Conditions

Example

In the image above, the **Conditions** component has just one **Branch**. This branch checks whether the player is moving or not. If it happens to move moving while this Conditions component is executed, it will print the "Player is moving" message on the console.

CREATING CONDITIONS

Right click on the *Hierarchy* panel and select *Game Creator* *Visual Scripting* *Conditions* A game object named 'Conditions' will appear in the scene with a component of the same name.

Alternatively you can also add the **Conditions** component to any game object clicking on the Inspector's Add Component button and searching for Conditions.

Deleting Conditions

To delete a Conditions component, simply click on the component's little cog button and select "Remove Component" from the dropdown menu.

ADDING BRANCHES

To add a new **Branch** simply click on the *Add Branch* button. This will create a new branch at the bottom of the **Conditions** component. You can then click and drag the `=` symbol on the right and reorder the branch list.

Branch Order

Remember that top branches have higher priority than lower ones when executed.

All **Branches** have a *Description* field, which can be used to more easily identify what that branch does. It has no gameplay effect.

CONDITIONS AND INSTRUCTIONS

A **Branch** is composed of a list of **Conditions** and a list of **Instructions**. Adding them is as easy as clicking on the *Add Condition* and *Add Instruction* respectively and choose the desired element.

Negate Condition

It is important to note that a specific **Condition** can be negated. For example, if the condition "Is Player Moving" returns success when the player is moving, but false when it's not, you can check for the opposite effect clicking on the small green toggle. It will now return true of the player is not moving, and true otherwise.

Toggle Condition

Empty Conditions List

An empty conditions list will always return success.

Conditions

CONDITIONS

Sub Categories

- Cameras
- Characters
- Game Objects
- Input
- Math
- Physics
- Scenes
- Storage
- Text
- Transforms

CAMERAS

Cameras

Conditions

- Is Shot Active

Is Shot Active

```
Cameras » Is Shot Active
```

Description

Returns true if the Camera Shot is assigned to the Main Camera

Parameters

Name	Description
Shot	The camera shot

Keywords

`Camera` `Enabled` `Assigned` `Running`

CHARACTERS

Characters

Sub Categories

- Animation
- Busy
- Interaction
- Navigation
- Properties
- Visuals

Animation

Animation Conditions

- Has State In Layer

Has State in Layer

```
Characters » Animation » Has State in Layer
```

Description

Returns true if the Character has a State running at the specified layer index

Parameters

Name	Description
Layer	The layer in which the Character may have a State running
Character	The Character instance referenced in the condition

Keywords

`Characters` `Animation` `Animate` `State` `Play` `Character` `Player`

Busy

Busy Conditions

- Are Legs Available
- Is Available
- Is Busy
- Is Left Arm Available
- Is Left Leg Available
- Is Right Arm Available
- Is Right Leg Available

Are Legs Available

```
Characters » Busy » Are Legs Available
```

Description

Returns true if the Character's legs are available to start a new action

Parameters

Name	Description
Character	The Character instance referenced in the condition

Keywords

`Occupied` `Available` `Free` `Doing` `Foot` `Feet` `Character` `Player`

Is Available

```
Characters » Busy » Is Available
```

Description

Returns true if the Character is not doing any action and is free to start one

Parameters

Name	Description
Character	The Character instance referenced in the condition

Keywords

`Occupied` `Available` `Free` `Doing` `Character` `Player`

Is Busy

```
Characters » Busy » Is Busy
```

Description

Returns true if the Character doing an action that prevents from starting another one

Parameters

Name	Description
Character	The Character instance referenced in the condition

Keywords

`Occupied` `Available` `Free` `Doing` `Character` `Player`

Is Left Arm Available

```
Characters » Busy » Is Left Arm Available
```

Description

Returns true if the Character's left arm is available to start a new action

Parameters

Name	Description
Character	The Character instance referenced in the condition

Keywords

`Occupied` `Available` `Free` `Doing` `Hand` `Finger` `Character` `Player`

Is Left Leg Available

```
Characters » Busy » Is Left Leg Available
```

Description

Returns true if the Character's left leg is available to start a new action

Parameters

Name	Description
Character	The Character instance referenced in the condition

Keywords

`Occupied` `Available` `Free` `Doing` `Foot` `Feet` `Character` `Player`

Is Right Arm Available

```
Characters » Busy » Is Right Arm Available
```

Description

Returns true if the Character's right arm is available to start a new action

Parameters

Name	Description
Character	The Character instance referenced in the condition

Keywords

`Occupied` `Available` `Free` `Doing` `Hand` `Finger` `Character` `Player`

Is Right Leg Available

```
Characters » Busy » Is Right Leg Available
```

Description

Returns true if the Character's right leg is available to start a new action

Parameters

Name	Description
Character	The Character instance referenced in the condition

Keywords

`Occupied` `Available` `Free` `Doing` `Foot` `Feet` `Character` `Player`

Interaction

Interaction Conditions

- Can Interact

Can Interact

```
Characters » Interaction » Can Interact
```

Description

Returns true if the Character has any interactive element available

Parameters

Name	Description
Character	The Character instance referenced in the condition

Keywords

`Character` `Button` `Pick` `Do` `Use` `Pull` `Press` `Push` `Talk` `Character` `Player`

Navigation

Navigation Conditions

- Is Airborne
- Is Grounded
- Is Idle
- Is Moving

Is Airborne

```
Characters » Navigation » Is Airborne
```

Description

Returns true if the Character not touching the ground

Parameters

Name	Description
Character	The Character instance referenced in the condition

Keywords

`Fly` `Fall` `Flail` `Jump` `Float` `Suspend` `Character` `Player`

Is Grounded

```
Characters » Navigation » Is Grounded
```

Description

Returns true if the Character touching the floor

Parameters

Name	Description
Character	The Character instance referenced in the condition

Keywords

`Floor` `Stand` `Land` `Character` `Player`

Is Idle

```
Characters » Navigation » Is Idle
```

Description

Returns true if the Character is not moving

Parameters

Name	Description
Character	The Character instance referenced in the condition

Keywords

`Stay` `Quiet` `Still` `Character` `Player`

Is Moving

```
Characters » Navigation » Is Moving
```

Description

Returns true if the Character is currently in an active moving phase

Parameters

Name	Description
Character	The Character instance referenced in the condition

Keywords

`Translate` `Towards` `Destination` `Target` `Follow` `Walk` `Run` `Character` `Player`

Properties

Properties Conditions

- Can Jump
- Compare Gravity
- Compare Height
- Compare Mass
- Compare Radius
- Compare Speed
- Is Dead
- Is Player
- Jump Force
- Terminal Velocity

Compare Mass

```
Characters » Properties » Can Jump
```

Description

Returns true if the character has the Can Jump property set to true

Parameters

Name	Description
Character	The Character instance referenced in the condition

Keywords

`Active` `Enabled` `Leap` `Hop` `Character` `Player`

Compare Gravity

```
Characters » Properties » Compare Gravity
```

Description

Returns true if the comparison between a number and the Character's gravity is satisfied

Parameters

Name	Description
Character	The Character instance referenced in the condition

Keywords

`Force` `Vertical` `Character` `Player`

Compare Height

```
Characters » Properties » Compare Height
```

Description

Returns true if the comparison between a number and the Character's height is satisfied

Parameters

Name	Description
Character	The Character instance referenced in the condition

Keywords

`Length` `Long` `Character` `Player`

Compare Mass

```
Characters » Properties » Compare Mass
```

Description

Returns true if the comparison between a number and the Character's mass is satisfied

Parameters

Name	Description
Character	The Character instance referenced in the condition

Keywords

`Weight` `Character` `Player`

Compare Radius

```
Characters » Properties » Compare Radius
```

Description

Returns true if the comparison between a number and the Character's radius is satisfied

Parameters

Name	Description
Character	The Character instance referenced in the condition

Keywords

`Diameter` `Width` `Fat` `Skin` `Space` `Character` `Player`

Compare Speed

```
Characters » Properties » Compare Speed
```

Description

Returns true if the comparison between a number and the Character's speed is satisfied

Parameters

Name	Description
Character	The Character instance referenced in the condition

Keywords

`Velocity` `Travel` `Movement` `Walk` `Run` `Step` `Character` `Player`

Is Dead

`Characters » Properties » Is Dead`

Description

Returns true if the character has been killed

Parameters

Name	Description
Character	The Character instance referenced in the condition

Keywords

`Kill` `Kaput` `Character` `Player`

Is Player

```
Characters » Properties » Is Player
```

Description

Returns true if the Character is marked as a Player

Parameters

Name	Description
Character	The Character instance referenced in the condition

Keywords

`Control` `Character` `Character` `Player`

Compare Jump Force

```
Characters » Properties » Jump Force
```

Description

Returns true if the comparison between a number and the Character's jump force is satisfied

Parameters

Name	Description
Character	The Character instance referenced in the condition

Keywords

`Hop` `Leap` `Character` `Player`

Compare Terminal Velocity

```
Characters » Properties » Terminal Velocity
```

Description

Returns true if the comparison between a number and the Character's terminal velocity is satisfied

Parameters

Name	Description
Character	The Character instance referenced in the condition

Keywords

`Max` `Fall` `Vertical` `Down` `Character` `Player`

Visuals

Visuals Conditions

- Has Prop Attached

Has Prop Attached

```
Characters » Visuals » Has Prop Attached
```

Description

Returns true if the Character has a Prop attached to the specified bone

Parameters

Name	Description
Bone	The bone that has the prop attached to
Character	The Character instance referenced in the condition

Keywords

`Characters` `Holds` `Grab` `Draw` `Pull` `Take` `Object` `Character` `Player`

GAME OBJECTS

Game Objects

Conditions

- Compare Game Objects
- Compare Layer
- Compare Tag
- Does Component Exist
- Does Game Object Exist
- Is Component Enabled
- Is Game Object Active

Compare Game Objects

```
Game Objects » Compare Game Objects
```

Description

Returns true if the game object is the same as another one

Parameters

Name	Description
Game Object	The game object instance used in the comparison
Compare To	The game object instance that is compared against

Keywords

`Same` `Equal` `Exact` `Instance`

Compare Layer

```
Game Objects » Compare Layer
```

Description

Returns true if the game object belongs to any of the layer mask values

Parameters

Name	Description
Game Object	The game object instance used in the condition
Layer Mask	A bitmask of Layer values

Keywords

`Mask` `Physics` `Belong` `Has`

Compare Tag

```
Game Objects » Compare Tag
```

Description

Returns true if the game object is tagged with a concrete name

Parameters

Name	Description
Game Object	The game object instance used in the condition
Tag	The Tag name checked against the game object

Keywords

`Belong` `Has` `Is`

Does Component Exist

```
Game Objects » Does Component Exist
```

Description

Returns true if the game object has the component attached

Parameters

Name	Description
Game Object	The game object instance used in the condition
Component	The component type that is searched

Keywords

`Null` `Scene` `Lives`

Does Game Object Exist

```
Game Objects » Does Game Object Exist
```

Description

Returns true if the game object reference is not null

Parameters

Name	Description
Game Object	The game object instance used in the condition

Keywords

`Null` `Scene` `Lives`

Is Component Enabled

```
Game Objects » Is Component Enabled
```

Description

Returns true if the game object has the component enabled

Parameters

Name	Description
Game Object	The game object instance used in the condition
Component	The component type checked

Keywords

`Null` `Active`

Is Game Object Active

```
Game Objects » Is Game Object Active
```

Description

Returns true if the game object reference exists and is active

Parameters

Name	Description
Game Object	The game object instance used in the condition

Keywords

`Null` `Scene` `Enabled`

INPUT

Input

Conditions

- Is Key Held Down
- Is Key Pressed
- Is Key Released
- Is Mouse Held Down
- Is Mouse Pressed
- Is Mouse Released

Is Key Held Down

```
Input » Is Key Held Down
```

Description

Returns true if the keyboard key is being held down this frame

Parameters

Name	Description
Key	The Keyboard key that is checked

Keywords

`Button` `Active` `Down` `Press`

Is Key Pressed

```
Input » Is Key Pressed
```

Description

Returns true if the keyboard key is pressed during this frame

Parameters

Name	Description
Key	The Keyboard key that is checked

Keywords

`Button` `Down`

Is Key Released

```
Input » Is Key Released
```

Description

Returns true if the keyboard key is released during this frame

Parameters

Name	Description
Key	The Keyboard key that is checked

Keywords

`Button` `Up`

Is Mouse Held Down

```
Input » Is Mouse Held Down
```

Description

Returns true if the mouse button is being held down

Parameters

Name	Description
Button	The Mouse button that is checked

Keywords

`Key` `Up` `Click` `Cursor`

Is Mouse Pressed

```
Input » Is Mouse Pressed
```

Description

Returns true if the mouse button is pressed during this frame

Parameters

Name	Description
Button	The Mouse button that is checked

Keywords

`Key` `Down` `Cursor`

Is Mouse Released

```
Input » Is Mouse Released
```

Description

Returns true if the mouse button is released during this frame

Parameters

Name	Description
Button	The Mouse button that is checked

Keywords

`Key` `Up` `Click` `Cursor`

MATH

Math

Sub Categories

- Arithmetic
- Boolean
- Geometry

Arithmetic

Arithmetic Conditions

- Compare Decimal
- Compare Integer

Compare Decimal

```
Math » Arithmetic » Compare Decimal
```

Description

Returns true if a comparison between two decimal values is satisfied

Parameters

Name	Description
Value	The decimal value that is being compared
Comparison	The comparison operation performed between both values
Compare To	The decimal value that is compared against

Keywords

`Number` `Float` `Comma` `Equals` `Different` `Bigger` `Greater` `Larger` `Smaller`

Compare Integer

```
Math » Arithmetic » Compare Integer
```

Description

Returns true if a comparison between two integer values is satisfied

Parameters

Name	Description
Value	The integer value that is being compared
Comparison	The comparison operation performed between both values
Compare To	The integer value that is compared against

Keywords

`Number` `Whole` `Equals` `Different` `Bigger` `Greater` `Larger` `Smaller`

Boolean

Boolean Conditions

- Compare Boolean

Compare Boolean

```
Math » Boolean » Compare Boolean
```

Description

Returns true if a comparison between two boolean values is satisfied

Parameters

Name	Description
Value	The boolean value that is being compared
Comparison	The comparison operation performed between both values
Compare To	The boolean value that is compared against

Geometry

Geometry Conditions

- Compare Direction
- Compare Distance
- Compare Point

Compare Direction

```
Math » Geometry » Compare Direction
```

Description

Returns true if a comparison between two direction values is satisfied

Parameters

Name	Description
Value	The direction value that is being compared
Comparison	The comparison operation performed between both values
Compare To	The direction value that is compared against

Keywords

`Towards` `Vector` `Magnitude` `Length` `Equals` `Different` `Greater` `Larger` `Smaller`

Compare Distance

```
Math » Geometry » Compare Distance
```

Description

Returns true if a comparison of the distance between two points is satisfied

Parameters

Name	Description
Point A	The first operand that represents a point in space
Point B	The second operand that represents a point in space
Comparison	The comparison operation performed between both values
Distance	The distance value compared against

Keywords

`Position` `Vector` `Magnitude` `Length` `Equals` `Different` `Greater` `Larger` `Smaller`

Compare Point

```
Math » Geometry » Compare Point
```

Description

Returns true if a comparison between two points in space is satisfied

Parameters

Name	Description
Value	The point in space that is being compared
Comparison	The comparison operation performed between both values
Compare To	The point in space that is compared against

Keywords

`Position` `Vector` `Magnitude` `Length` `Equals` `Different` `Greater` `Larger` `Smaller`

PHYSICS

Physics

Conditions

- Check Box 2D
- Check Box 3D
- Check Circle
- Check Sphere
- Is Kinematic
- Is Sleeping

Check Box 2D

```
Physics » Check Box 2D
```

Description

Returns true if casting a 2D box at a position doesn't collide with anything

Parameters

Name	Description
Position	The scene position where the box's center is cast. Z axis is ignored
Size	Size of each side's extension along its local axis
Angle	Cloc-wise rotation measured in degrees
Layer Mask	A bitmask that skips any objects that don't belong to the list

Example 1

Note that this Instruction uses Unity's 2D physics engine. It won't collide with any 3D objects

Keywords

`Check` `Collide` `Touch` `Suit` `Square` `Cube` `2D`

Check Box 3D

```
Physics » Check Box 3D
```

Description

Returns true if casting a 3D box at a position doesn't collide with anything

Parameters

Name	Description
Position	The scene position where the box's center is cast
Rotation	The rotation of the cube cast in world space
Half Extents	Half size of the cube that extents along its local axis
Layer Mask	A bitmask that skips any objects that don't belong to the list

Example 1

Note that this Instruction uses Unity's 3D physics engine. It won't collide with any 2D objects

Keywords

`Check` `Collide` `Touch` `Suit` `Square` `Cube` `3D`

Check Circle

```
Physics » Check Circle
```

Description

Returns true if casting a circle at a position doesn't collide with anything

Parameters

Name	Description
Position	The scene position where the circle's center is cast. Z axis is ignored
Radius	The radius of the circle in Unity units
Layer Mask	A bitmask that skips any objects that don't belong to the list

Example 1

Note that this Instruction uses Unity's 2D physics engine. It won't collide with any 3D objects

Keywords

`Check` `Collide` `Touch` `Suit` `Sphere` `Circumference` `Round` `2D`

Check Sphere

```
Physics » Check Sphere
```

Description

Returns true if casting a sphere at a position doesn't collide with anything

Parameters

Name	Description
Position	The scene position where the sphere's center is cast
Radius	The radius of the sphere in Unity units
Layer Mask	A bitmask that skips any objects that don't belong to the list

Example 1

Note that this Instruction uses Unity's 3D physics engine. It won't collide with any 2D objects

Keywords

`Check` `Collide` `Touch` `Suit` `Circle` `Circumference` `Round` `3D`

Is Kinematic

```
Physics » Is Kinematic
```

Description

Returns true if the game object's Rigidbody or Rigidbody2D is marked as Kinematic

Parameters

Name	Description
Game Object	The game object instance with a Rigidbody or Rigidbody2D

Keywords

`Affect` `Physics` `Force` `Rigidbody`

Is Sleeping

```
Physics » Is Sleeping
```

Description

Returns true if the game object's Rigidbody or Rigidbody2D is sleeping

Parameters

Name	Description
Game Object	The game object instance with a Rigidbody or Rigidbody2D

Keywords

`Affect` `Physics` `Force` `Rigidbody` `Awake`

SCENES

Scenes

Conditions

- Is Scene Loaded

Is Scene Loaded

```
Scenes » Is Scene Loaded
```

Description

Returns true if the scene has been loaded

Parameters

Name	Description
Scene	The Unity Scene reference used in the condition

STORAGE

Storage

Conditions

- Has Save

Has Save

```
Storage » Has Save
```

Description

Returns true if there is at least one saved game

Keywords

`Game` `Load` `Continue` `Resume` `Can` `Is`

TEXT

Text

Conditions

- Text Contains
- Text Equals

Text Contains

```
Text » Text Contains
```

Description

Returns true if the second text string occurs in the first one

Parameters

Name	Description
Text	The text string
Substring	The text string contained in Text

Keywords

`String` `Char` `Sub`

Text Equals

```
Text » Text Equals
```

Description

Returns true if two text Strings are equal

Parameters

Name	Description
Text 1	The first text string to compare
Text 2	The second text string to compare

Keywords

`String` `Char`

TRANSFORMS

Transforms

Conditions

- Child Count
- Is Child Of
- Is Sibling Of

Child Count

```
Transforms » Child Count
```

Description

Compares the amount of direct children of a game object

Parameters

Name	Description
Target	The children amount of this game object instance
Comparison	The comparison operation between the child count and a value
Compare To	The second value compared

Keywords

`Transform` `Hierarchy` `Descendant` `Ancestor` `Parent` `Father` `Amount`

Is Child Of

```
Transforms » Is Child Of
```

Description

Returns true if the game object is the parent of the other one

Parameters

Name	Description
Child	The game object instance further down in the hierarchy of the parent
Parent	The game object instance that is higher in the hierarchy

Keywords

`Transform` `Hierarchy` `Descendant` `Ancestor` `Parent` `Father` `Mother`

Is Sibling Of

```
Transforms » Is Sibling Of
```

Description

Returns true if the game object shares the same parent as the other one

Parameters

Name	Description
Sibling A	The game object instance compared
Sibling B	Another game object instance compared

Keywords

`Transform` `Hierarchy` `Ancestor` `Brother` `Sister`

Custom Conditions

Game Creator allows to very easily create custom **Conditions**.

Programming Knowledge Required

This section assumes you have some programming knowledge. If you don't know how to code you might be interested in checking out the Game Creator Hub page. Programmers altrusitically create custom **Conditions** for others to download and use in their project.

CREATING A CONDITION

The easiest way to create an **Condition** C# script is to right click on your *Project* panel and select *Create* *Game Creator* *Developer* *C# Condition*. This will create a template script with the boilerplate structure:

```
using System;
using GameCreator.Runtime.Common;
using GameCreator.Runtime.VisualScripting;

[Serializable]
public class MyCondition : Condition
{
    protected override bool Run(Args args)
    {
        return true;
    }
}
```

Anatomy of an Instruction

A **Condition** is a class that inherits from the `Condition` super class. The abstract `Run(...)` method is the entry point of a **Condition**'s execution, which is automatically called. This method must always return `true` if it's successful, or `false` otherwise.

The `Run(...)` method has a single parameter of type `Args`, which is a helper class that contains a reference to the game object that initiated the call (`args.Self`) and the targeted game object (`args.Target`), if any.

Decoration & Documentation

It is highly recommended to document and decorate the **Condition** so it's easier to find and use. It is done using class-type attributes that inform **Game Creator** of the quirks of this particular condition.

For example, to set the title of a condition to "Hello World", use the `[Title(string name)]` attribute right above the class definition:

```
using System;
using GameCreator.Runtime.Common;
using GameCreator.Runtime.VisualScripting;

[Title("Hello World")]
[Serializable]
public class MyCondition : Condition
{
    protected override bool Run(Args args)
    {
        return true;
    }
}
```

Title

The title of the Condition. If this attribute is not provided, the title will be a beautified version of the class name.

```
[Title("Title of Condition")]
```

Description

A description of what the Condition does. This is both used in the floating window documentation, as well as the description text when uploading a Condition to the Game Creator Hub.

```
[Description("Lorem Ipsum dolor etiam porta sem magna mollis")]
```

Image

The `[Image(...)]` attribute changes the default icon of the Condition for one of the default ones. It consists of 2 parameters:

- **Icon [`Type`]:** a Type class of an `IIcon` derived class. **Game Creator** comes packed with a lot of icons although you can also create your own.
- **Color [`Color`]:** The color of the icon. Uses Unity's `Color` class.

For example, one of the icons included is the "Solid Cube" icon. To display a red solid cube as the icon of the condition, use the following attribute:

```
[Image(typeof(IconCubeSolid), Color.red)]
```

Category

A sequence of sub-categories organized using the slash (`/`) character. This attribute helps keep the Conditions organized when the dropdown list is displayed.

```
[Category("Category/Sub Category/Name")]
```

The example above will display the Condition under the sub directory *Category* *Sub Category* *.Name*

Version

A semantic version to keep track of the development of this Condition. It's important to note that when updating a Condition to the Game Creator Hub, the version number must always be higher than the one on the server.

The semmantic version follows the standard *Major Version*, *Minor Version*, *Patch Version*. To know more about how semantic versioning works, read the following page: https://semver.org.

```
[Version(1, 5, 3)]
```

Parameters

When a Condition has exposed fields in the Inspector, it's a good idea to document what these do. You can add as many `[Parameter(name, description)]` attributes as exposed fields has.

For example, if the Condition has these two fields:

```
public bool condition1 = true;
public bool condition2 = false;
```

You can document those fields adding:

```
[Parameter("Condition 1", "First condition value to check")]
[Parameter("Condition 2", "Second condition value to check")]
```

Keywords

Keywords are strings that help the fuzzy finder more easily search for a condition. For example, the "Is Character Moving" condition doesn't reference the word "idle" or "walk" anywhere in its documentation. However, these words are very likely to reference this condition when the user types them in the search box.

```
[Keywords("Idle", "Walk", "Run")]
```

Example

The Example attribute allows to display a text as an example of use of this Condition. There can be more than one `[Example(...)]` attribute per condition. This is particularly useful when uploading conditions on the Game Creator Hub.

Markdown

It is recommended to use Markdown notation when writing examples

```
[Example("Sed posuere consectetur est at lobortis)]
```

Multiple Lines

You can use the `@` character in front of a string to break the example text in multiple lines. To create a new paragraph, simply add two new lines. For example:

```
[Example(@"
    This is the first paragraph.
    This is also in the first paragraph, right after the previous sentence

    This line is part of a new paragraph.
")]
```

Dependency

This attribute is optional and only used in the Game Creator Hub. If this Condition uses some particular feature of a specific module, it will first check if the user downloading this condition has that module installed. If it does not, it will display an error message and forbid downloading it. This is useful to avoid throwing programming errors.

The `[Dependency(...)]` attribute consists of 4 parameters:

- **Module ID:** For example, the ID of the Inventory module is `gamecreator.inventory`.
- **Major Version:** The minimum major version of the dependency module.
- **Minor Version:** The minimum minor version of the dependency module.
- **Patch Version:** The minimum patch version of the dependency module.

```
[Dependency("gamecreator.inventory", 1, 5, 2)]
```

1.5.5 Hotspots

Hotspots

Hotspots are components attached to game objects that don't have any direct impact on gameplay. Instead, they help the user understand what's interactive and what is not. For example, highlighting a specific object when the player character is nearby, making the head turn towards an important object and so on.

Hotspots

Trigger + Hotspot

Triggers are usually placed along side with **Hotspot** components. One deals with the interaction itself, while the other hints the player about the **Trigger** being an interactive object.

HOW IT WORKS

A **Hotspot** consists of a *Target* field and a *Radius*, which are the position and distance relative to the Hotspot at which it's activated. When a hotspot is activated, it signals its **Spot** list the targeted game object is nearby. When the targeted object gets further away than the *Radius* field, the hotspot gets deactivated.

Hotspot Gizmo

Selecting a game object with a **Hotspot** component will display in the scene a visual representation of the distance at which the target is considered close enough to activate it.

Debugging

On playmode, the red gizmo appears in a much lighter color. If the targeted object activates the Hotspot, the Hotspot's gizmo will change to green, to indicate the Hotspot is active.

No Phyics Engine

The **Hotspot** distance check doesn't use Unity's Phyics engine because it would force both the Hotspot and the targeted object to have a *Collider* component attached to them. Instead it simply checks the distance between the center of the hotspot and the targeted game object.

CREATING HOTSPOTS

There are two ways to create a Hotspot object. One is to create an object that contains a Hotspot component, by right clicking on the *Hierarchy* panel and selecting *Game Creator* *Visual Scripting* *Hotspot*. This creates a scene object with the component attached to it.

However, an Actions component can also be added to any game object. Simply click on any game object's *Add Component* button and type Actions.

Deleting Actions

To delete an Actions component, simply click on the component's little cog button and select "Remove Component" from the dropdown menu.

ADDING SPOTS

Spots are individual elements that highlight something specific and are evaluated from top to bottom.

Add new Spot

To add a new **Spot** click on the *Add Spot* button and choose the desired one from the dropdown list. Note that **Spots** are evaluated from top to bottom. There can be two spots of the same type, but if they both overlap, the last one will override the effect.

Spots

SPOTS

Spots

- Cursor
- Look At
- Look On Focus
- Object On Focus
- Object
- Text On Focus
- Text

CURSOR

```
Cursor
```

Description

Changes the cursor image when hovering the Hotspot

LOOK AT

```
Look At
```

Description

Makes the Character look at the center of the Hotspot when it's activatedand smoothly look away when it's deactivated

LOOK ON FOCUS

```
Look on Focus
```

Description

Makes the Character look at the center of the Hotspot when it's an interactive and is focused

OBJECT ON FOCUS

```
Object on Focus
```

Description

Creates or Activates a prefab game object when the Interactive object is focused and deactivates it when its unfocused

OBJECT

```
Object
```

Description

Creates or Activates a prefab game object when the Hotspot is enabled and deactivates it when the Hotspot is disabled

TEXT ON FOCUS

```
Text on Focus
```

Description

Displays a text in a world-space canvas when the Hotspot is focused by the target and hides it when it is not. If no Prefab is provided, a default UI is displayed

TEXT

```
Text
```

Description

Displays a text in a world-space canvas when the Hotspot is enabled and hides it when is disabled. If no Prefab is provided, a default UI is displayed

1.6 Variables

1.6.1 Variables

Variables are data containers that allow to dynamically change their value and let the game keep track of the player's progress.

Example

A very simple use case of **Variables** is keeping track of the player's score. Let's say we have a named variable called *score* and has an initial value of 0. Every time the player picks up a star, the *score* variable is incremented and its value is displayed.

Types of Variables

Game Creator has two types of variables:

NAME VARIABLES

Are identified by their unique name. For example, the name *score* can reference a numeric variable that keeps track of the player's score value.

Name Variables

LIST VARIABLES

Are identified by their 0-based index. Think of them as a collection of values, placed one after another. For example, to access the first value, use the index *0*. To access the second position, use the index *1*, etc...

Note all values of a **List Variable** are of a particular type.

List Variables

Name or List?

As a rule of thumb, it is recommended the use of **Name Variables**. **List Variables** are useful when you have an unknown number of objects to choose from. For example, when locking on an enemy from a group that surrounds the player.

Scope of Variables

Variables can either be local or global.

LOCAL VARIABLES

Local Variables are bound to a particular scene and can't be used outside of it.

GLOBAL VARIABLES

On the other hand, **Global Variables** can be queried and modified from any scene.

Types

Both **Global Variables** and **Local Variables** can be *List* or *Name* based.

Value Types

All **Variables** have an initial value assigned to them that can be modifed at runtime. By default, **Game Creator** comes with a limited number of types to choose from, but other modules might increment the amount available.

- **Number:** Stores numeric values. Both decimal and integers.
- **String:** Stores text-based characters.
- **Boolean:** Can only store two values: *true* or *false*.
- **Vector 3:** Stores an (x,y,z) vector value
- **Color:** Stores an RGBA color value. Can also contain HDR information.
- **Texture:** Stores a reference to a Texture asset.
- **Sprite:** Stores a reference to a Sprite asset.
- **Game Object:** Stores a reference to a game object.

Saving Values

It is important to note that not all data types can be saved between play-sessions. **Textures**, **Sprites** and **Game Objects** and not primitive types and thus, they can't be serialized at runtime.

1.6.2 Global Name Variables

Global Name Variables are variables identified by a unique string of characters that live outside the scene and can be accessed and modified from anywhere.

Creating a Global Name Variable

To create a **Global Name Variable**, right click on the *Project Panel* and select *Create* *Game Creator* *Variables* *Name Variables*. A new asset will appear in the project panel, which can be used to define each of the variables contained within.

Global Name Variables

Conflicting ID

Note that two Global Variables can't have the same unique ID. Otherwise they'll override each other's values. To generate a new unique ID, expand the *ID* field and click the "Regenerate" button.

Adding new entries

To add a new variable entry, type the name of the variable on the creation field and press enter (or click on the little [+] button).

The name of a variable can be modified, as well as its value type. The *Value* field also contains the starting value of this particular variable entry.

Save & Load

Vales can be saved between play sessions to later be restored when loading a game. Disabling the *save* option will make all variables keep the initial value as their starting value, even after loading a previously saved game.

1.6.3 Global List Variables

Global List Variables are variables identified by their numberic index value and can be accessed from anywhere.

Creating a Global List Variable

To create a **Global List Variable**, right click on the *Project Panel* and select *Create* *Game Creator* *Variables* *List Variables*. A new asset will appear in the project panel, which can be used to define the collection of variables.

Global List Variables

Conflicting ID

Note that two Global Variables can't have the same unique ID. Otherwise they'll override each other's values. To generate a new unique ID, expand the *ID* field and click the "Regenerate" button.

Save & Load

Vales can be saved between play sessions to later be restored when loading a game. Disabling the *save* option will make all variables keep the initial value as their starting value, even after loading a previously saved game.

1.6.4 Local Name Variables

Local Name Variables are variables identified by a unique string of characters that live inside a scene and can only reference objects that are contained inside this scene.

Creating a Local Name Variable

To create a **Local Name Variable**, right click on the Hierarchy Panel_ and select *Game Creator* *Variables* *Name Variables* A new game object will appear with the **Local Name Variables** component. Alternatively you can also add this component to any existing game object.

Global Name Variables

Conflicting ID

Note that two Local Variables can't have the same unique ID. Otherwise they'll override each other's values. To generate a new unique ID, expand the *ID* field and click the "Regenerate" button.

Adding new entries

To add a new variable entry, type the name of the variable on the creation field and press enter (or click on the little [+] button).

The name of a variable can be modified, as well as its value type. The *Value* field als contains the starting value of this particular variable entry.

Save & Load

Vales can be saved between play sessions to later be restored when loading a game. Disabling the *save* option will make all variables keep the initial value as their starting value, even after loading a previously saved game.

1.6.5 Local List Variables

Local List Variables are variables identified by their numberic index value and can only be accessed from the scene they are part of.

Creating a Local List Variable

To create a **Local List Variable**, right click on the Hierarchy Panel_ and select *Create* *Game Creator* *Variables* *List Variables*. A new game object with the component will appear in the scene and hierarchy. Alternatively, you can also add the *Local List Variables* component to any existing game object.

Local List Variables

Conflicting ID

Note that two Local Variables can't have the same unique ID. Otherwise they'll override each other's values. To generate a new unique ID, expand the *ID* field and click the "Regenerate" button.

Save & Load

Vales can be saved between play sessions to later be restored when loading a game. Disabling the *save* option will make all variables keep the initial value as their starting value, even after loading a previously saved game.

1.7 Advanced

1.7.1 Advanced

Game Creator includes a collection of tools used throughout the entire ecosystem. This section briefly goes over all of them and provides a link to each tool's page, where they are explained in-depth, with use cases and examples.

Advanced Level

This section of the Documentation assumes you are familiar with Unity and Game Creator. Some sections may require you to also have some coding knowledge.

Audio

Game Creator has a 4 channel audio system that makes it very easy to change volume settings and play both diegetic and non-diegetic sound effects.

Learn about **Audio**

Signals

Communication between game objects is handled using the visual scripting tools, such as **Triggers** and **Actions**. However, there may be cases where the developer needs to respond to more tailored events that don't exist in Game Creator.

Signals

The **Raise Signal** instruction broadcasts a message with a specific identifier and any **Trigger**(s) listening to that specific id will be executed. To receive a signal message, use the **On Receive Signal** and specify the identifier.

Mark as Favorite

To avoid misspelling mistakes you can mark a **Signal** name as *favorite*, which can be used selecting them from the dropdown button on the right side. To unfavorite a name, simply click again on the *star* button.

Data Structures

Advanced Data Structures (also known as *ADS*) are generic data structures that help better perform certain tasks.

- **Unique ID**: Uniquely identifies an object with a serializable Guid.
- **Singleton**: It ensures there's zero or one instance of a class at any given moment and its value is globally accessible.
- **Dictionary**: A serializable dictionary.
- **Hash Set**: A serializable Hash Set.
- **Link List**: A serializable Linked List.
- **Matrix 2D**: A serializable 2D matrix.
- **Tree**: Generic structure that allows to have acyclic parent-child depenedencies between multiple class instances.
- **Ring Buffer**: This structure is similar to a generic list, but sequentially accessing its elements yields in an infinite circular loop, where the last element connects with the first one.
- **State Machine**: A data structure that allows to dynamically manipulate a state machine and define logic on each of its nodes independently.
- **Spatial Hash**: An advanced data structure that allows to detect collisions of any radial size inside an infinite spatial domain with an O complexity of *log(n)*.

Variables API

Local Variables and **Global Variables** can be modified at runtime using the exposed API. Note that **Local** variables are accessed via their component and **Global** variables require to be accessed through a singleton manager that contain their runtime values.

Learn how to use the **Variables API**

Properties

Properties are a core feature that allows to dynamically access a value. They are usually displayed as a drop-down menu and allow to retrieve them depending on the option selected.

For example, a `PropertyGetPosition` allows to get a `Vector3` that represents a position, from different sources; A constant value, the Player's position, the main camera's position, from a Local Variable, etc...

Learn more about **Properties**

Saving & Loading

Game Creator comes with a fully extensible save and load system that allows to easily keep track of the game progress and restore its state at any time. All that needs to be done is to implement an interface called `IGameSave` and subscribe/unsubscribe inside the `OnEnable()` and `OnDisable()` methods respectively.

- **Saving and Loading**
- **Saving custom data**
- **Saving on custom databases**

There is a special component called **Remember** that allows to cherry-pick the bits of data you want to save when saving a game.

Tweening

Game Creator comes packged with a powerful **Tweening** (or automatic frame interpolation, from in-between-ing) system. It allows to *fire & forget* a command that creates a tween between a starting value and end value. The transition can be linear or an easing function can be specified.

Learn more about **Tweening**

Examples and Templates

Game Creator and all modules come with a collection of examples and templates ready to be used on your games and applications. Other developers can leverage this feature in order to create reusable examples that can be installed/uninstalled across multiple projects or share them if you are a module developer using the **Example Manager** window.

Learn more about **Creating custom Examples**

1.7.2 Audio

Game Creator comes with an audio manager that automatically manages and optimizes the creation and decomission of audio sources. There are 4 different types of audio channels, each with its own volume slider and properties.

Ambient

Ambient sounds are what one could also call background music or ambience. It's a looped tracked played in the background, and can be diegetic or non-diegetic. For example, a battle music track, the chirping of birds in a forest, or the sound of a waterfall.

Play Ambient Instruction

Use the **Play Ambient** Instruction to play an audio clip as an Ambient sound. It will keep playing until a **Stop Ambient** Instruction is executed.

Sound Effects

Sound Effects (also known as SFX) are one-time clips played at a very specific time. The majority of sounds on a game will be sound effects, for example: Punching a character, footstep sounds, or a slash of a sword. Most sound effects are diegetic and thus, by default expect a spatial position.

Sound Variation

To avoid the jarring effect where the same sound effect is played over and over again in a small time window, sound effects can automatically randomly alter the *speed* and *pitch* of sounds. This allows to, for example, play a machine gun sound effect, where each shot is slightly different than the previous one.

Play Sound Effect Instruction

Use the **Play Sound Effect** Instruction to play an audio clip as a Sound Effect. It will automatically decommision the audio source once the clip finishes playing.

UI

UI sound effects are non-diegetic clips played when the player interacts with the user interface. For example, hovering over a button, clicking it or crafting an item after the user waits a timeout.

Play UI Instruction

Use the **Play UI** Instruction to play an audio clip as a UI sound effect.

Speech

Speech clips are very similar to **Sound Effects** with the difference that they are bound to a Character, so that a specific character can only play one speech clip at a time.

Play Sound Effect Instruction

Use the **Play Speech** Instruction to play an audio clip as a Speech sound effect. If another clip is was being played on the same target, it will stop the previous speech and play the new one. This is useful when the user skips conversations.

1.7.3 Data Structures

Index

DATA STRUCTURES

Advanced Data Structures (also known as *ADS*) are generic data structures that help better perform certain tasks.

- **Unique ID**: Uniquely identifies an object with a serializable Guid.
- **Singleton**: It ensures there's zero or one instance of a class at any given moment and its value is globally accessible.
- **Dictionary**: A serializable dictionary.
- **Hash Set**: A serializable Hash Set.
- **Link List**: A serializable Linked List.
- **Matrix 2D**: A serializable 2D matrix.
- **Tree**: Generic structure that allows to have acyclic parent-child depenedencies between multiple class instances.
- **Ring Buffer**: This structure is similar to a generic list, but sequentially accessing its elements yields in an infinite circular loop, where the last element connects with the first one.
- **State Machine**: A data structure that allows to dynamically manipulate a state machine and define logic on each of its nodes independently.
- **Spatial Hash**: An advanced data structure that allows to detect collisions of any radial size inside an infinite spatial domain with an O complexity of *log(n)*.

Unique ID

To generate unique identifiers, it is usually used the `System.Guid` class, because it provides a fast and reliable mechanism to generate long enough IDs that the collision chance is almost zero.

However, this class is not serializable. That's why **Game Creator** comes with the `UniqueID` class, which serves two purposes:

- **Serializable:** This means that any changes made to this ID will be kept between editor sessions.
- **Custom UI:** When showing this ID in a Unity Window, it automatically displays a nice and handy box with buttons that allow to easily modify this ID or even regenerate it, in case that's necessary.

INITIALIZATION

To initialize a class instance of `UniqueID` is as easy as calling the constructor class. For example, let's say we want to add a unique ID to a `MonoBehaviour` class:

```
public class MyComponent : MonoBehaviour
{
    public UniqueID myID = new UniqueID();
}
```

This will automagically assign a unique ID to the `myID` field. If we drag and drop this component onto a scene game object, we'll see this field with its associated ID.

ACCESSING ID

Accessing the ID value can be performed getting the `IdString` struct, which contains a string based ID and its hash value. This last one is recommended when comparing to ids:

To get the hash value:

```
int hash = this.myID.Get.Hash;
```

To get the `string` value:

```
string id = this.myID.Get.String;
```

Best Practices

Accessing the `string` value of the UniqueID should only be done if you plan on serializing this value somewhere. For comparing two IDs, it is best if you simply compare their `hash` value, as the probablity that two strings have the same hash value its very, very very low. On the other hand, comparing two `int` values is extremely fast and performant.

Singleton

The **Singleton** pattern ensures there's, at most, one instance of a class at any given time. Because of that, it can be globally accessed from its class name. To make a singleton class, inherit from the `Singleton<T>` type:

```
public MyClass : Singleton<MyClass>
{ }
```

To access this class, use `MyClass.Instance` which returns an instance of the MyClass. If none was present, it creates one and then it returns it, so you don't have to worry about keeping track whether it has been created or not.

MonoBehaviour

This Singleton pattern is specifically designed to work with Unity and thus, it requires the `MyClass` to inherit from `MonoBehaviour`. However, this is defined automatically when inheriting from the `Singleton<T>` class.

If you need to perform some setup when creating a new class instance, override the the `OnCreate()` method. Likewise, you can also override the `OnDestroy()` method to execute some logic when the instance is destroyed.

```
public MyClass : Singleton<MyClass>
{
    protected override void OnCreate()
    {
        base.OnCreate();
        // This is executed only once when created
    }

    protected override void OnDestroy()
    {
        base.OnDestroy();
        // This is executed only once when destroyed
    }
}
```

Singleton instances can survive or be destroyed every time their scene is unloaded. By default all singleton classes survivde scene reloading. But if you want to destroy them when changing between scenes, override the `SurviveSceneLoads` and set it to `false`:

```
public MyClass : Singleton<MyClass>
{
    protected override bool SurviveSceneLoads => false;
}
```

Dictionary

The serializable dictionary allows to have the whole fully fledged functionality of `System.Collections.Dictionary` but also allows to automatically serialize its values.

To create a serializable dictionary, simply inherit from `TSerializableDictionary<TKey, TValue>`. For example, to create a dictionary that uses `string` as their key and `GameObject` as their value:

```
public MyDictionary : TSerializableDictionary<string, GameObject>
{ }
```

You can now create a dictionary that automatically serializes its values and use it as any normal dictionary:

```
public MyComponent : MonoBehaviour
{
    public MyDictionary dictionary = new MyDictionary();

    private void Awake()
    {
        // Add element to dictionary:
        this.dictionary.Add("Hello World", this.gameObject);

        // Print element added
        Debug.Log(this.dictionary["Hello World"].name);
    }
}
```

Hash Set

The serializable hash set allows to have the functionality of `System.Collections.HashSet` but also allows to automatically serialize its values.

To create a serializable hash set, simply inherit from `TSerializableHashSet<T>`. For example, to create a hash set that uses `string` types:

```
public MyHashSet : TSerializableHash<string>
{ }
```

You can now create a hash set that automatically serializes its values and use it as:

```
public MyComponent : MonoBehaviour
{
    public MyHashSet hashSet = new MyHashSet();

    private void Awake()
    {
        // Add element:
        this.hashSet.Add("Hello World");

        // Print if it can find the elements
        Debug.Log(this.hashSet.Contains("Hello World"));
        Debug.Log(this.hashSet.Contains("Foo"));
    }
}
```

Link List

The serializable linked list allows to have the functionality of `System.Collections.LinkedList` but also allows to automatically serialize its values.

To create a serializable linked list, simply inherit from `TSerializableLinkList<T>`. For example, to create a hash set that uses `GameObject` types:

```
public MyLinkedList : TSerializableLinkList<GameObject>
{ }
```

You can now create a list that automatically serializes its values and use it as:

```
public MyComponent : MonoBehaviour
{
    public MyLinkedList list = new MyLinkedList();

    public GameObject objectA;
    public GameObject objectB;
    public GameObject objectC;

    private void Awake()
    {
        // Add element:
        this.list.Add(this.objectA);
        this.list.AddLast(this.objectB);
        this.list.AddFirst(this.objectC);

        // Print the first element:
        Debug.Log(this.list.First().name);
    }
}
```

Matrix 2D

The serializable 2D matrix allows to have an array of arrays (where all rows and columns have the same size) and the structure can be serialized in order to persist in the Inspector or saving the game.

To create a serializable matrix, simply inherit from `TSerializableMatrix2D<T>`. For example, to create a matrix that uses `GameObject`:

```
public MyMatrix : TSerializableMatrix2D<GameObject>
{ }
```

You can now create a matrix that automatically serializes its values:

```
public MyComponent : MonoBehaviour
{
    public MyMatrix matrix = new MyMatrix(10, 5);

    private void Awake()
    {
        // Add element:
        this.matrix[2, 3] = this.gameObject;

        // Print element added
        Debug.Log(this.matrix[2, 3].name);
    }
}
```

Tree

The Tree class allows to create acyclic dependency graphs that start from a root node and end with leaf nodes. A single node can have an unlimited number of branches.

To create a Tree, inherit from the `Tree<T>` class, where T is the value type of the node. For example, to create a tree of game objects:

```
public MyTree : Tree<GameObject>
{ }
```

```
public MyComponent : MonoBehaviour
{
    public MyTree tree = new MyTree();

    private void Awake()
    {
        // Add element:
        this.tree.AddChild(this.gameObject);

        foreach (var child in this.tree)
        {
            // Print child id:
            Debug.Log(child.Value.id);

            // Print child game object:
            Debug.Log(child.Value.Data.name);
        }
    }
}
```

A `Tree<T>` class is both the tree and the node class. So any child of a tree returns a tree object too. A tree can return its parent:

```
MyTree parent = this.tree.Parent
```

And it's children, which is a dictionary indexed by its Ids:

```
KeyValuePair<string, GameObject> = this.tree.Children;
```

Ring Buffer

The **Ring Buffer** is a very interesting data structure that works very similar to an array, except that its capacity is capped and iterating over its elements will automatically jump from its tail to its head when reaching the end of the list. Think of it as an array with a limited capacity where the tail joins the head, thus shaping it a ring.

To create a ring buffer, create a class that inherits from the `Ring<T>` class or directly use the `Ring<T>` type. For example, to create a ring buffer with 5 elements:

```
Ring<string> myRing = new Ring<string>(
    "string 1",
    "string 2",
    "string 3",
    "string 4",
    "string 5",
);
```

The ring buffer starts with its index pointing to the first element. Calling `Next()`, `Current()` and `Previous()` will change the pointer and return the new value. For example:

```
// Set the index to 0:
myRing.Index = 0;

// Iterate 100 times:
for (int i = 0; i < 100; ++i)
{
    // Print the next value:
    Debug.Log(myRing.Next());
}
```

The previous code snippet will iterate the previous ring 20 times (100 / 5) and print the name of each entry.

An interesting method of the ring buffer is the `Update(callback)`. This method accepts a method as its parameter and executes it for every element of the ring. For example:

```
myRing.Update(Debug.Log);
```

The previous method will print each of the entries of the ring buffer, as the `Debug.Log()` method is applied to each one of them.

State Machine

A **State Machine** is a commonly used pattern that allows to isolated the complexity of multiple tasks in different nodes, in a way that each node is not aware of what others do.

About State Machines

For a full description of what a finite state machine is check this Wikipedia article.

CREATING STATES

Let's start seeing how to create states before creating a state machine. A **State** is a single node unit from the state machine. To create one, create a class that inherits from the `StateMachine.State` abstract class:

```
public class MyState1 : StateMachine.State
{ }
```

A State has 3 virtual methods that can be overriden in order to execute its custom logic:

```
// Executed when the machine changes to this state
void WhenEnter(StateMachine machine)
{ }

// Executed when the machine exists from this state
protected virtual void WhenExit(StateMachine machine)
{ }

// Executed every frame while this state is active
protected virtual void WhenUpdate(StateMachine machine)
{ }
```

A state has an `IsActive` property that can be queried to check if this state is currently the active one.

If you need to hook events to a State in order to make it work with other scripts, you can also subscribe to its event system.

```
// Executed when the machine changes to this state
event Action<StateMachine, State> EventOnEnter;

// Executed when the machine exists from this state
event Action<StateMachine, State> EventOnExit;

// Executed every frame while this state is active, before the WhenUpdate(...)
event Action<StateMachine, State> EventOnBeforeUpdate;
```

For example, let's first create an instance of `MyState1`:

```
MyState1 state1 = new MyState();
```

Now let's hook an external method that prints a message when the state is entered:

```
state1.EventOnEnter += this.OnEnterState;
```

The `OnEnterState(...)` method must have the following signature:

```
public void OnEnterState(StateMachine machine, State state)
{
    Debug.Log("Hello World!");
}
```

CREATING A STATE MACHINE

To create a state machine, create a class that inherits from `StateMachine`:

```
public class MyStateMachine : StateMachine
{
    public MyStateMachine(State state) : base(state)
    { }
}
```

First State

Note that a State Machine requires at least one state to be passed to the constructor. This is the first starting state that the machine will begin with.

The developer is responsible for calling its `Update()` method. We recommend calling it in a *MonoBehaviour*'s `Update()`.

To instruct the machine to change from one state to another, use the `Change(State)` method:

```
MyState1 state1 = new MyState1();
MyState2 state2 = new MyState2();

// Initialize with state1
MyStateMachine machine = new MyStateMachine(state1);

// Change to state2
machine.Change(state2);
```

A State Machine also has 2 events that allow methods to be subscribed, which are launched as soon as there is a change in the currently active state:

```
event Action<State> EventStateEnter;
event Action<State> EventStateExit;
```

Spatial Hash

The **Spatial Hash** algorithm is a performant non-physics based query system that returns a list of objects contained in a position and a certain radius.

Performance

This algorithm scales with the amount of objects tracked. Its performance shines the most when there are multiple queries launched in a single frame. For more information about how this algorithm works check this Twitter post:

https://twitter.com/catsoftstudios/status/1201520331724333058

CREATING A DOMAIN

The first thing needed is to create a world domain from where to track all objects and organize the space partitioning. We recommend setting up a static class that will handle registering all the changes that happen in the scene. For example:

```
public static class MySpatialHash
{
    public static SpatialHash Value { get; private set; } = new SpatialHash();

    [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.SubsystemRegistration)]
    private static void OnSubsystemsInit()
    {
        Value = new SpatialHash();
    }
}
```

The previous code snippet initializes the `Value` field with the default `SpatialHash` constructor. the `OnSubsystemInit()` is a method that gets called at the very beginning of starting the game, before any scene is loaded, thanks to its attribute.

TRACKING CHANGES

Each object instance is responsible for updating the domain value when it changes. To do so, the object must implement the `ISpatialHash` interface, as well as call the `Insert()`, `Remove()` and `Update()` methods to start, stop and update the spatial hash's domain. For example:

```
public class MyComponent : MonoBehaviour, ISpatialHash
{
    void OnEnable()
    {
        // Start tracking this object
        MySpatialHash.Value.Insert(this);
    }

    void OnDisable()
    {
        // Stop tracking this object
        MySpatialHash.Value.Remove(this);
    }

    void Update()
    {
        // Update tracking position
        MySpatialHash.Value.Update(this);
    }

    // ISpatialHash interface. Position in space:
    Vector3 ISpatialHash.Position => this.transform.position;

    // ISpatialHash interface. Identifies this class:
    int ISpatialHash.UniqueCode => this.gameObject.GetInstanceID();
}
```

Boost Performance

This code is meant for demonstration purposes and might not be optimal on every case. If you want to squeeze every drop of performance, you may want to cache the last tracked position and only call the `Update(this)` method when its position has changed.

REQUESTING COLLECTIONS

To request all the objects around a point and within a specific radius, use the `Query(Vector3 point, float radius)` method, which returns a list of game objects contained in the specified region.

```
// Define a point and radius in the 3D space:
Vector3 point = new Vector3(0,0,0);
float radius = 10f;

// request for all tracked game object within:
List<ISpatialHash> list = MySpatialHash.Value.Query(point, radius);
```

The list contains all components that implement the `ISpatialHash` interface tracked in this domain that are within the spherical region defined.

1.7.4 Variables API

Local Variables

Local Name Variables and **Global Name Variables** are components attached to game objects and their value is bound to the scene they are. To access their runtime values you reference the component and call one of their public methods.

LOCAL NAME VARIABLES

Local Name Variables are components attached to game objects and can be referenced like any other script. To access any of its values you can use the following methods:

Getting values

```
bool Exists(string name)
```

Returns true if the variable exists. False otherwise

```
object Get(string name)
```

Returns the value of the variable. Requires to be casted to the correct value

Setting values

```
void Set(string name, object value)
```

Sets the value of a variable

Listening to events

You can also register when a **Local Name Variable** changes using the following methods:

```
void Register(Action<string> callback)
```

Executes the callback every time a variable changes its value

```
void Unregister(Action<string> callback)
```

Stops executing the callback when the variable changes

LOCAL LIST VARIABLES

A **Local List Variables** component has the following methods for getting and manipulating its values:

Getting values

```
object Get(IListGetPick pick)
```

Returns the value indexed by the pick parameter

```
int Count
```

Property that returns the number of elements of the list

Setting values

```
void Set(IListSetPick pick, object value)
```

Sets a value indexed by the pick parameter

```
void Insert(IListGetPick pick, object content)
```

Inserts a value at the indexed position

```
void Push(object value)
```

Adds a new value at the end of the list

```
void Remove(IListGetPick pick)
```

Removes the value indexed by the pick parameter

```
void Clear()
```

Removes all values from the list

```
void Move(IListGetPick pickA, IListGetPick pickB)
```

Moves the value indexed at a position to a new index

Listening to events

You can also register when a **Local List Variable** changes any of its items using the following methods:

```
void Register(Action<ListVariableRuntime.Change, int> callback)
```

Executes the callback method whenever there's a change

```
void Unregister(Action<ListVariableRuntime.Change, int> callback)
```

Stops executing the callback when the list changes

Global Variables

Global Name Variables and **Global List Variables** are scriptable objects and their runtime value is stored in a separate singleton manager called `GlobalNameVariablesManager` and `GlobalListVariablesManager`.

GLOBAL NAME VARIABLES

The `GlobalNameVariablesManager` has the following methods available:

Getting values

```
bool Exists(GlobalNameVariables asset, string name)
```

Returns true if the variable exists. False otherwise

```
object Get(GlobalNameVariables asset, string name)
```

Returns the value of the variable. Requires to be casted to the correct value

Setting values

```
void Set(GlobalNameVariables asset, string name, object value)
```

Sets the value of a variable

Listening to events

You can also register when a **Global Name Variable** changes using the following methods:

```
void Register(GlobalNameVariables asset, Action<string> callback)
```

Executes the callback every time the variable changes its value

```
void Unregister(GlobalNameVariables asset, Action<string> callback)
```

Stops executing the callback when the variable changes

GLOBAL LIST VARIABLES

The `GlobalListVariablesManager` has the following methods:

Gettings values

```
int Count(GlobalListVariables asset)
```

Returns the number of elements of the list

```
object Get(GlobalListVariables asset, IListGetPick pick)
```

Setting values

Returns the value indexed by the pick parameter

```
void Set(GlobalListVariables asset, IListSetPick pick, object value)
```

Sets a value indexed by the pick parameter

```
void Insert(GlobalListVariables asset, IListGetPick pick, TValue content)
```

Inserts a value at the indexed position

```
void Push(GlobalListVariables asset, TValue value)
```

Adds a new value at the end of the list

```
void Remove(GlobalListVariables asset, IListGetPick pick)
```

Removes the value indexed by the pick parameter

```
void Clear(GlobalListVariables asset)
```

Removes all values from the list

```
void Move(GlobalListVariables asset, IListGetPick pickA, IListGetPick pickB)
```

Moves the value indexed at a position to a new index

Listening to events

You can also register when a **Global List Variable** changes any of its items using the following methods:

```
void Register(GlobalListVariables asset, Action<ListVariableRuntime.Change, int> callback)
```

Executes the callback method whenever there's a change

```
void Unregister(GlobalListVariables asset, Action<ListVariableRuntime.Change, int> callback)
```

Stops executing the callback when the list changes

1.7.5 Properties

Game Creator properties are a special type of class that allows to dynamically specify the source of a field value using a dropdown menu. The menu's options are dynamic and can be added without the need of overwriting **Game Creator** core code, allowing to write maintainable and decoupled code.

Polymorphic Serialization

Properties take advantage of Unity's polymorphic serialzation, which means that the dropdown menu options are decoupled from the core code. Anyone can plug in their own menu options without overwriting any scripts.

There are different types of **Properties**, each with its own set of options. All of them have in common that, when retrieving them, an instance of `Args` parameter is passed, which contains two fields:

- **Target:** A reference to the `Game Object` responsible for calling the property
- **Self:** A reference to the `Game Object` containing the property reference.

Args Parameter

There are some cases where the `Target` and `Self` fields will reference the same game object.

Property *Get* types allow to retrieve a value and Property *Set* types allow to set a value. **Game Creator** comes with a collection of both types, but each module increases the amount available. You can even create your own property types to extend the existing ones.

Property Get Types

There are a few default property types available:

- `PropertyGetBool` : A boolean value type
- `PropertyGetColor` : A representation of a Color
- `PropertyGetDecimal` : A decimal value
- `PropertyGetDirection` : A Vector3 representing a direction
- `PropertyGetGameObject` : References a game object
- `PropertyGetInstantiate` : Allows to reference an instance
- `PropertyGetInteger` : An integer value
- `PropertyGetLocation` : A position and/or rotation
- `PropertyGetOffset` : A Vector3 that offsets from a position
- `PropertyGetPosition` : A Vector3 representing a point in space
- `PropertyGetRotation` : A Quaternion representing a rotation
- `PropertyGetScale` : A Vector3 representing a scalar value
- `PropertyGetScene` : Allows to select scene objects
- `PropertyGetSprite` : Returns Sprite assets
- `PropertyGetString` : Returns texts
- `PropertyGetTexture` : To retrieve Texture assets

Property Set Types

- `PropertySetBool` : Sets a boolean value
- `PropertySetColor` : Sets a Color type value

- `PropertySetGameObject`: Sets a game object reference
- `PropertySetNumber`: Sets a numeric value
- `PropertySetSprite`: Sets a Sprite reference
- `PropertySetString`: Sets a text-based value
- `PropertySetTexture`: Sets a Texture asset reference
- `PropertySetVector3`: Sets a Vector3 type value

Using Properties

UI Toolkit

Using properties requires the Editor scripts to be written using Unity's UI Toolkit. IMGUI is not supported.

To use a property it's very simple. You just need to declare them as you would with a primitive type, but instead of getting the value directly, call the `Get(args)` method to retrieve its value.

For example, let's say that in a component, you want to get a `string` value. Instead of declaring a value like this:

```
public string myValue = "This is my string";
```

You could use a property so the source of that string value isn't hard-coded, but set from the Inspector. Like this:

```
public PropertyGetString myValue = new PropertyGetString();
```

This will display a dropdown menu on the Inspector with the current option selected. By default it's a constant string, but the value can be chosen to come from the name of a game object, a local or global variable, etc.

To get the value you simply call the `Get(args)` method:

```
string value = this.myValue.Get(args);
```

Args

The `Args` (arguments) class is a two-field struct that contains the game object considered as the *source* of the call as well as the *targeted* game object. This class is necessary in order to use properties that reference the "Self" or "Target" values. If you are not sure what the self and target objects are, simply pass in the current MonoBehaviour's game object:

```
Args args = new Args(this.gameObject);
```

1.7.6 Save & Load

Saving and Loading

Game Creator comes with a flexible mechanism to keep track of changes made at runtime and store these by calling a simple `Save()` method. Likewise, restoring any previously saved game can be done executing a `Load()` method from the class responsible for managing this functionality.

WHAT CAN BE SAVED AND LOADED

The Save/Load system can save any primitive serializable field: integers, booleans, strings, positions, rotations or any managed instance type marked with the `[System.Serializable]` attribute.

However, it does not serialize objects inheriting or fields referencing objects that inherit from `Unity.Object`. For example: Game Objects, Transforms, MonoBehaviours, ...

SAVE SLOTS

Most games allow to store multiple saves and allow the user to choose which one to restore when loading a previous saved play. With **Game Creator**, each one of these save spaces are called **slots** and they are represented by an integer number ranging from 1 up to 999.

Note

Notice that you can have up to 998 slots. The number 0 is reserved for *shared settings*.

SAVING

To save a game, it's as easy as calling the `Save(slot: integer)` method through the `SaveLoadManager` singleton class. This class is responsible for tracking all objects in the scene and silently collects their state in a background process. Saving a game can be done using the following line, passing a constant save slot number 1 as a parameter:

```
SaveLoadManager.Instance.Save(1);
```

By default, the saving system uses Unity's *PlayerPrefs* system, which blocks the main thread until al data is written. However, **Game Creator** provides tools that allow to customize how data is saved. You could even have an online database where you dump the player's save files.

Because we can't assume the saving will be done synchronously, the `Save(slot: int)` method returns a `Task` that can be awaited. This is very useful if you plan on synchronizing the game save with an external database, such as Steam, Firebase or any other online data warehouse service.

To handle these cases, all that needs to be done is use an async method and await the result. Like so:

```
public async Task MySaveFunction()
{
    Debug.Log("Start saving game");
    await SaveLoadManager.Instance.Save(1);
    Debug.Log("Game has been saved");
}
```

However, if you are using the default *PlayerPrefs* save system or your custom one does block the main thread when saving, you can either await the task or use a discard operator:

```
public void MySaveFunction()
{
    Debug.Log("Start saving game");
    _ = SaveLoadManager.Instance.Save(1);
    Debug.Log("Game has been saved");
}
```

LOADING

Loading a previously saved game is very similar to saving one.

It is important to highlight that loading a game forces to unload the current scene and loads the saved one afterwards. Even if they are the same.

```
SaveLoadmManager.Instance.Load(1);
```

The `Load(slot: int)` method returns a `Task` object, just like the `Save(slot: int)`. You can choose to either await the load or, in most cases, use the discard operator:

```
public void MyLoadFunction()
{
    _ = SaveLoadManager.Instance.Load(1);
}
```

DELETING

A user may want to delete all the information associated to a save *slot*. This can be done using the following line:

```
SaveLoadManager.Instance.Delete(1);
```

The `Delete(slot: int)` method also returns a `Task` object. However, in this case, it may be more interesting knowing when a delete operation has finished.

EVENTS

The saving and loading system contains 6 events that programmers can hook onto to detect when a saving and a loading process has started.

- `public event Action<int> EventBeforeSave;`
- `public event Action<int> EventAfterSave;`
- `public event Action<int> EventBeforeLoad;`
- `public event Action<int> EventAfterLoad;`
- `public event Action<int> EventBeforeDelete;`
- `public event Action<int> EventAfterDelete;`

For example, doing something when a *save* operation is about to start can be achieved subscribing to the `EventBeforeSave` event:

```
void Start()
{
    SaveLoadManager.Instance.EventBeforeStart += this.OnBeforeSave;
}

public void OnBeforeSave(int slot)
{
    Debug.Log("About to save game in slot " + slot);
}
```

You can subscribe to as many methods as you need in each event. However, make sure to remove the subscription when the class that is doing subscribing is destroyed. For example, following the excerpt from above, it would also be optimal to do:

```
void OnDestroy()
{
    SaveLoadManager.Instance.EventBeforeStart -= this.OnBeforeSave;
}
```

CUSTOMIZE

As mentioned before, **Game Creator** doesn't assume a specific save or load procedure. In fact, it provides with tools to customize how data is collected and stored in order for the developer to customize it and tailor it to its needs.

In the following sections we'll see how to:

- Create a custom class that can be saved
- Create a custom database communication service

Custom Data

The `SaveLoadManager` class keeps track of all savable objects in the scene and collects their state in a background process so when the `Save()` method is invoked, it contains all the information required to successfully perfom the oepration.

In order to let the `SaveLoadManager` know what objects it needs to keep track of the developers need to implement the `IGameSave` interface on each object that contains data to save.

As soon as the object is available, it must call the `Subscribe(reference: IGameSave, priority: int)` method. Likewise, when the object is destroyed it should call `Unsubscribe(reference: IGameSave)`.

THE IGAMESAVE INTERFACE

The `IGameSave` interface requires to fill the following methods and properties:

- `string SaveID`: Gives an id that uniquely identifies this data
- `bool IsShared`: Tells whether this data is shared across all save games
- `Type SaveType`: Returns the type of the object to be serialized and stored
- `object SaveData`: Returns the instance of the object that's going to be saved
- `LoadMode LoadMode`: Define whether loading happens following a `Greedy` or a `Lazy` format
- `void OnLoad(object value)`: Callback for when the game is loaded

In order to understand better how this works, it's better to demonstrate this with an example.

Let's say that in our game we have one single chest in a scene that the player can only open once.

```
public class MyChest: MonoBehaviour
{
    public bool hasBeenOpened = false;

    public void OnOpen()
    {
        Debug.Log("Do something, like giving a potion to player");
        this.hasBeenOpened = true;
    }
}
```

In order to keep track of whether the chest has been opened or not, we implement the `IGameSave` interface on the component that defines the behavior of the chest:

```
public class MyChest: MonoBehaviour, IGameSave
{
    public bool hasBeenOpened = false;

    public void OnOpen()
    {
        if (this.hasBeenOpened) return;

        Debug.Log("Do something, like giving a potion to player");
        this.hasBeenOpened = true;
    }

    // The id for this save game is 'my-chest'
    public string SaveID => "my-chest";

    // This save should not be shared across multiple slots
    public bool IsShared => false;

    // The object type we're going to be saving
    public Type SaveType => typeof(bool);

    // The value we're going to store
    public object SaveData => this.hasBeenOpened;

    // The loading mode should be set as lazy
    public LoadMode LoadMode => LoadMode.Lazy;

    // When loading the game, restore the state
    public void OnLoad(object value)
    {
        this.hasBeenOpened = (bool)value;
    }
}
```

Most fields should be self explanatory. It is importnat to highlight though, that it's up to the developer to implement how the state is restored. The `OnLoad(object value)` is called when a game is loaded, and the `value` parameter is the value from a previously saved game. It's the developer's responsability to cast the object value to a valid type and assign the values to whichever fields are necessary.

The **Load Mode** is a tricky concept. It's an enum that allows to choose between two options:

- **Lazy**: This should be the default option for 90% of the cases. When this option is selected, the save and load system will restore the state of an object when this object is created. Not before.
- **Greedy**: This requires a persistent object that survives cross-scene transitions (set as `DontDestroyOnLoad()` method). Most commonly used with singleton patterns, this mode forces the load as soon as the event is triggered.

SUBSCRIPTION

Now, all that's left to do is tell the `SaveLoadManager` to keep track of this component as soon as it's initialized, and unsubscribe from it when the component is destroyed. Following the previous example, we implement the `OnEnable()` and `OnDisable()` Unity methods to subscribe and unsubscribe respectively:

```
public class MyChest: MonoBehaviour, IGameSave
{
    public bool hasBeenOpened = false;

    void OnEnable()
    {
        _ = SaveLoadManager.Subscribe(this);
    }

    void OnDisable()
    {
        _ = SaveLoadManager.Unsubscribe(this);
    }

    // IGameSave implementation below
    // ...
}
```

This gives all the necessary information to the save and load system about the life-cycle of this object so it can keep track of its state progress. If your object is never destroyed and survives scene transitions, you can skip the unsubscription.

To wrap things up, here's the full script of the example:

```
public class MyChest: MonoBehaviour, IGameSave
{
    public bool hasBeenOpened = false;

    public void OnOpen()
    {
        if (this.hasBeenOpened) return;

        Debug.Log("Do something, like giving a potion to player");
        this.hasBeenOpened = true;
    }

    void OnEnable()
    {
        _ = SaveLoadManager.Subscribe(this);
    }

    void OnDisable()
    {
        _ = SaveLoadManager.Unsubscribe(this);
    }

    public string SaveID => "my-chest";
    public bool IsShared => false;

    public Type SaveType => typeof(bool);
    public object SaveData => this.hasBeenOpened;

    public LoadMode LoadMode => LoadMode.Lazy;

    public void OnLoad(object value)
    {
        this.hasBeenOpened = (bool)value;
    }
}
```

The `hasBeenOpened` property will always return `false` if the `OnOpen()` method has never been executed, but will return `true` if it has at some point. If the user saves and loads back the game, its value will be kept.

Custom Save Location

By default, **Game Creator** saves games using the *PlayerPrefs* built-in system. However, although this solution is cross-platform and will work for most users, some might prefer to sync their saves with an online database or use a different system than Unity's *PlayerPrefs*.

Here we will explore how easy it is to extend the save location.

IDATASTORAGE INTERFACE

To create a custom save location, one must create a class that implements the `IDataStorage` interface, which contains all the necesary methods to store game information.

To make things easier, we're going to create a very simple system that communicates with an online database and stores the game saves there using http requests.

Note

Notice that there aren't any error handling mechanism for sake of simplicity. A production-ready product should also check and inform of the necessary errors that may ocurr.

Let's create our storage location class called `MyOnlineDatabase.cs`:

```
[Serializable]
public class MyOnlineDatabase: IDataStorage
{
    private const string URL_DB_SET = "https://database.mywebsite.com/set";
    private const string URL_DB_GET = "https://database.mywebsite.com/get";
    private const string URL_DB_DEL = "https://database.mywebsite.com/del";

    string IDataStorage.Title => "My Online Database";
    string IDataStorage.Description => "Store data in online database";

    async Task IDataStorage.DeleteAll()
    {
        // Create a web request to delete the content
        UnityWebRequest request = UnityWebRequest.Post(URL_DB_DEL, "");
        UnityWebRequestAsyncOperation handle = request.SendWebRequest();
        while (!handle.isDone) await Task.Yield();
    }

    async Task IDataStorage.DeleteKey(string key)
    {
        // Create a web request to delete a key
        UnityWebRequest request = UnityWebRequest.Post(URL_DB_DEL, key);
        UnityWebRequestAsyncOperation handle = request.SendWebRequest();
        while (!handle.isDone) await Task.Yield();
    }

    async Task<bool> IDataStorage.HasKey(string key)
    {
        // Checks whether a key exists in the database (code 200)
        UnityWebRequest request = UnityWebRequest.Post(URL_DB_GET, key);
        UnityWebRequestAsyncOperation handle = request.SendWebRequest();
        while (!handle.isDone) await Task.Yield();
        return handle.webRequest.responseCode == 200;
    }

    async  Task<object> GetBlob(string key, Type type, object value)
    {
        // Create a request to get the value identified by a key
        UnityWebRequest request = UnityWebRequest.Post(URL_DB_GET, key);
        UnityWebRequestAsyncOperation handle = request.SendWebRequest();
        while (!handle.isDone) await Task.Yield();
        return JsonUtility.FromJson(
            handle.webRequest.downloadHandler.text,
            type
        );
    }

    async Task<string> IDataStorage.GetString(string key, string value)
    { /* ... */ }

    async Task<float> IDataStorage.GetFloat(string key, float value)
    { /* ... */ }

    async Task<int> IDataStorage.GetInt(string key, int value)
    { /* ... */ }

    async Task SetBlob(string key, object value)
    {
        // Requests the creation or update of a value onto the database
        UnityWebRequest request = UnityWebRequest.Post(URL_DB_SET, new Data(){
            id =  key,
            data = JsonUtility.ToJson(value)
        });
        UnityWebRequestAsyncOperation handle = request.SendWebRequest();
        while (!handle.isDone) await Task.Yield();
    }

    async Task IDataStorage.SetString(string key, string value)
    { /* ... */ }

    async Task IDataStorage.SetFloat(string key, float value)
    { /* ... */ }

    async Task IDataStorage.SetInt(string key, int value)
    { /* ... */ }
}
```

The first properties `Title` and `Description` allow to give a name to this system, which later can be selected from a dropdown menu in the Preferences window.

The following methods define how data is manipulated: retrieving data, setting data and deleting data. There are 3 URL we're using to exemplify how we can create an http request to send the information to our server, which can delete, create or retrieve the information depending on the endpoint used.

Some methods have been skipped because their implementation was very similar to other ones.

It is important to note though that all methods have the `async` prefix and either return a `Task` object or a `Task` associated with an object. This is because there's a certain amount of time elapsed between the http request and the answer from the server. Being able to await requests let's you tailor how to safely chain commands and make sure each request is successfully fulfilled.

Remember

The **Remember** component allows to cherry-pick the data that is stored when saving the game. By default, it stores the position, rotation and scale.

Remember

To add a new element to be saved, click on the *Add Memory* button and select the type of data to save.

CREATING A MEMORY

Game Creator comes with a set of default memories, but you can create custom ones that extend the data stored. To create a new **Memory** create a new class that inherits from the `Memory` class. For this example, we'll create a *memory* that saves name of the game object attached to this memory.

```
[Serializable]
public class MemoryName : Memory
{
    public override string Title => "Name of Game Object";

    public override Token GetToken(GameObject target)
    {
        return new TokenName(target);
    }

    public override void OnRemember(GameObject target, Token token)
    {
        if (token is TokenName tokenName)
        {
            target.name = tokenName.text;
        }
    }
}
```

The `Title` property determines the name of this memory. This has no effect on the data stored but it displays this value on the Inspector.

The `GetToken(...)` method returns the `Token` instance of this memory and is called when the game data is scheduled to be saved. A `Token` is a data container that contains the data to be stored. In this case, we'll need to create a new class called `TokenName` that inherits from `Token` and has a serializable field to save the name of the object.

```
[Serializable]
public class TokenName : Token
{
    public string text;

    public TokenName(GameObject target) : base()
    {
        this.text = target.name;
    }
}
```

The `OnRemember(...)` method is called when loading a previously saved game and is used to restore its state. In this case, it changes the name of the game object to the one it tries to *remember*.

Decorations

The custom `Memory` class instance can be decorated using any of the attributes found in the `Instruction`, `Condition` and `Event` classes.

1.7.7 Tween

Tweening is the process to define a starting positon and an end position, and let it transition from one to the other over the course of a specifed duration.

For exmaple, opening a door can be easily achieved defining it's starting position as its current position and its end point as the same as its starting one, plus 2 units up in the Y axis. Once you specify the duration, the door will slide upwards when the tweening is activated.

The Tweening library has been created with Game Creator in mind, but can also be leveraged to be used in other scripts. Use the `Tween.To(...)` static method to create a new transition.

The `To(gameObject, input)` has two parameters: The *Game Object* that recieves the tweening, and an instance of a `TweenInput` class, which configures the animation.

Following the example from above, let's say we want to slide a "door" object 2 units up in the air. We can define the `TweenInput` class instance like this:

```
Vector3 valueSource = door.position;
Vector3 valueTarget = door.position + Vector3(0,2,0);
float duration = 5f;

ITweenInput tween = new TweenInput<Vector3>(
    valueSource,
    valueTarget,
    duration,
    (a, b, t) => door.position = Vector3.Lerp(a, b, t),
    Tween.GetHash(typeof(Transform), "transform"),
    Easing.Type.QuadInOut
);
```

Transition Type

In this example we use a Vector3 transition, but it accepts any value type, like numbers, colors, quaternions, ... It's up to the *updateCall* to interpolate between the initial and final value.

Let's break down each of these parameters in order:

```
TweenInput<Vector3>(
    Vector3 start,
    Vector3 end,
    float duration
    Update updateCall,
    int hash,
    Easing.Type easing
);
```

- **start:** A value indicating the starting position
- **end:** A value indicating the end position
- **duration:** The amount of time it takes to complete the transition
- **updateCall:** A method called every frame while the transition occurs. Contains 3 parameters: The starting value, the end value and the completion ratio between 0 and 1.
- **hash:** An integer that uniquely identifies this transition. If another transition with the same id starts, it cancels the previous one.
- **easing:** An optional easing function. If none is provided, it will use a linear function.

1.7.8 Custom Installs

Game Creator comes with the **Install** window, which allows a user to install and uninstall examples and templates from all modules. This is something available to all module developers and here you'll learn how to create, step by step, a template for a module called "My Module".

Installer

The **installer** directory is where the compressed file with the information about it is located. This folder is usually found under the custom Module's path but can be anywhere on the project folder. It must contain two files:

- An **Installer** configuration file, which contains all the information related to the example, including its name, the module it belongs to, a description and the version of this package.
- A **Package.unitypackage** file, which contains the compressed assets that will be unpacked upon installing.

Installation Location

The installed location is the directory where the example is decompressed after installing an example in order to be used by the user. This folder is always located at the following route:

```
Assets/Plugins/Game Creator/Installs/
```

An installed extension will always have a folder parent called after the name of the module, followed by a dot, followed by the name of the example, followed by an @ symbol and the semmantic version of the example. For example, if the example is called "My Example" and it's from a module called "My Module", the installation location of the example will be:

```
Assets/Plugins/Game Creator/Installs/MyModule.MyExample@1.0.0/
```

Creating a custom Installer

The example installer can be placed anywhere in the project. For simplicity it should be created where you have the rest of the module's assets. For example, if you are creating a module called "My Module" and an example of that called "My Example", at the root of the Unity project, you may want to place the installer inside the *MyModule* folder:

```
Assets/
    MyModule/
        Examples/
            MyExample/
        Scripts/
        Textures/
        ...
```

THE INSTALLER ASSET

Now that there is a folder where we can drop in the installation files, we'll create an **Installer** asset inside the **MyExample** folder. To do so, right click on the aforementioned folder and select `Create -> Game Creator -> Developer -> Installer`. If the option doesn't appear, you can also duplicate any existing Installer asset. Once you have the Installer asset you can rename it so it makes sense for your project.

Name Convention

We recommend sticking to Game Creator's naming convention and name the asset following "[ModuleName].[ExampleName]". This makes it easier to identify the asset and avoids conflicting names with other examples from other modules.

With the **Installer** in place, click on the *Configuration* button to expand the properties available and fill in the fields:

- **Name:** Name of the Example. Following the example from above, this would be "My Example.
- **Module:** Name of the module. It is important to note that this determines the category of the example. In the use case from above, the name would be "My Module".
- **Description:** A thorough description of this example. Make sure to indicate any quirks the example may have or how to get started once the example is installed.
- **Author:** Name of the creator of this example. This has no implication other than giving credit to the creator.
- **Version:** The semmantic version of this example. Make sure to increase the value every time you create a new version of the example.
- **Complexity:** How difficult it is for users to understand this example. This is for informational purposes only.
- **Dependencies:** A collection of ID (module name + example name) that this example depends on.

Dealing with Dependencies

The **Install** window will automatically install any dependencies that an example may depend on, without prompting the user to do so. This allows to quickly resolve any conflicts between this example and others that are required to be installed.

For example, if the example *Example A* has *Example B* as a dependency, and this last one is not yet installed, attempting to install *Example A* will install both *Example A* and *Example B*.

If *Example B* cannot be found, it won't be possible to install *Example A* from the Install window and will prompt the user an error message telling which module could not be found.

MAKING THE SKELETON

Now that we have the installer in place it's time to create the skeleton from which to build our example. To do so, select the previously created **Installer** and in the Inspector, right click on the name of the installer. This will make a dropdown menu appear with a bunch of options:

- **Install Package:** Forces the installation of this example. However, it is recommended to use the Install window to perform any installation instructions.
- **Delete Package:** Deletes the installed example, if there's any.
- **Build Package:** Changes the name of the installation path to fit the version number and creates a *Package.unitypackage* file at the installation location.
- **Create Package:** Creates the bare bones structure that allows to develop a new example.

In our case, we want to click on the "Create Package" option. This will create a new folder at:

```
Assets/Plugins/Game Creator/Installs/MyModule.MyExample@1.0.0/
```

Inside this folder you can place all prefabs, materials, scenes or any content that the example must have. To generate (or compress) this folder so it can be shared, select the option "Build Package" from the previous dropdown menu. This will export all assets inside the aforementioned folder and create a file called **Package.unitypackage** at the same directory as the **Installer**.

Sharing your example

Once you have the example built, it is ready to be distributed. To share this example installer, you just need to export the folder with the installer and the Package.unitypackage file generated.

If you (or the user) opens the Install window, the module will be displayed as a sub category of the specificed module with the option to install it, update it and/or delete it, depending on whether there is an installed version or not.

1.8 Releases

1.8.1 Releases

Unity version

Game Creator 2 is in Beta and requires **Unity 2021.2** or higher in order to work.

2.3.15 (Next Release)

Release Pending

NEW

- Game Creator Toolbar
- Signal dispatching
- Instruction: Character move to direction
- Instruction: Character stop movement
- Instructions: Camera Shots
- Instruction: Raise Signal
- Event: Receive Signal
- Event: On Change Audio Volume
- Property: Audio Mixer Parameter

ENHANCED

- Run Visual Scripting components from Unity events
- Easier to navigate dropdown menus
- Footsteps textures mimic character rotation
- A Camera Shot can be assigned as the main one
- Improved performance of Editor UI elements

CHANGED

- Tree class renamed to Trie

FIXED

- Variables now accept integers, floats and doubles
- Some events were invoked when the Trigger was disabled
- Error thrown with inactive Local Variables
- Dead characters don't twitch or breathe anymore
- Async Manager exception throw exiting Play-Mode
- Settings window compressing overflowing elements
- Event: On Click does not execute over UI elements
- UI Controls have the UI layer as default
- Locations allow to specify the rotation
- Procedural animations take into account Time Scale

2.2.14

Release December 27, 2021

NEW

- Shot: New Anchor Peek camera shot
- Marker: New Inwards type
- Instruction: Play Footstep
- Spot: Look on Focus
- Shots: Can use easing functions
- New deep clone utility to duplicate instances
- Properties: Get and Set audio volumes

ENHANCED

- Spots: Disabled while interacting
- Spots: Offset option for World and Self space
- Event: Lifecycle events have better description
- Play button is contextually hidden
- Colors have HDR and non-HDR option
- Name Variables display non-available options
- String Variables can get values from other types

CHANGED

- Instruction: Toggle Bool uses one single property

FIXED

- Shots interpolate based on its duration
- Event: Characters not registering changes
- Prefab Variables error at runtime
- Tweening UI elements uses unscaled time
- Actions and Triggers catch exceptions
- Spatial Hash queries on Markers

2.1.13

Released December 1, 2021

NEW

- Interaction system
- Condition: Can Interact
- Instruction: Interact
- Event: On Focus
- Event: On Blur
- Event: On change NPC to Player
- Event: On change Player to NPC
- Spot: Text on Focus
- Spot: Object on Focus

ENHANCED

- Leaning IK default values
- Character inspector UX
- Conditions have more friendly names

CHANGED

- Name of Point and Click button
- Motion unit is more compartmentalized
- Hide Character gizmos collapsing each unit

FIXED

- Spatial Hash returning farther values

2.0.12

Released November 24, 2021

NEW

- Driver: Skin width exposed in Inspector
- Ragdoll animations

CHANGED

- Save/Load format does not use special characters

FIXED

- Shot: Lock On ignores Anchor and Target clipping
- Event: On Enter NavLink not detected
- Event: On Exit NavLink not detected
- NavMesh: Agents move between Off-Mesh Links
- Scene asset were null in standalone builds
- Character: Sinking in ground when using Feet IK

2.0.11

Released November 16, 2021

NEW

- Paste button for Visual Scripting
- Instruction: Sort List alphabetically

ENHANCED

- Conditions redesign
- Focus on search fields automatically
- Event: Input distance has offset option
- Shot: Lock on includes better default values

CHANGED

- States: Weight uses a range control

FIXED

- Airborne animations did not loop correctly
- Null check for characters Bone Rack

2.0.10

Released November 2, 2021

NEW

- Event: On Hover Enter
- Event: On Hover Exit
- Event: On Select
- Event: On Deselect
- Event: On Late Update
- Event: On Trigger Stay
- Condition: Has Prop Attached
- Property: Transform Offset
- Property: Character Bone
- Property: Spherical random point
- Property: Rotation of Camera
- Input: Interaction
- Point and Click examples

ENHANCED

- Right click on Dropdown options to go back
- Virtualized `TPolymorphicListTool` methods

FIXED

- Missing scroll in Game Creator Hub
- Regression: Point & Click on Player unit
- Characters Props out of range access

2.0.9

Released October 20, 2021

NEW

- Input: Mobile virtual joystick support
- IK: Lean towards motion direction
- Event: On Hotspot Activate
- Event: On Hotspot Deactivate
- Property: Light Intensity and Range

ENHANCED

- Rendering Pipeline in documentation
- Visual Scripting search engine precision

CHANGED

- Renamed Example Manager to Install window
- Renamed execution events to lifecycle path

FIXED

- Character radius out of sync with driver unit
- Crouch and Walk string input codes
- Memory leak in Camera Shot preview window
- Skeleton valid prefab type
- Conversion between float and double values
- Test Runner using float values

2.0.8

Released October 6, 2021

NEW

- New getters for each Vector3 component
- Instruction: Clamp Vector3

ENHANCED

- Examples with higher contrasting textures

FIXED

- Null check for gamepads and keyboards
- Null check for material _MainTex
- Input for walking using crouch settings
- Character footstep bones incorrect instance

2.0.7

Released September 27, 2021

NEW

- Start State to Character component
- Latest documentation PDF file
- Option to run camera in Fixed Update
- Dust FX on examples when character lands
- Rigidbody character Driver
- Instruction: Set Text
- Instruction: Text Join
- Instruction: Text Replace
- Instruction: Text Substring

ENHANCED

- Handling on Character units
- Performance on Reflective properties

FIXED

- Examples and improved their visuals
- Physics engine methods being called every frame

2.0.6

Released September 22, 2021

NEW

- Tank Controls to characters
- Copy & Paste to all lists
- Duplicate button to all lists
- Faster method to get managed reference values
- Get and Set values from Input devices
- Get and Set fields using C# Reflection
- Get and Set properties using C# Reflection
- Event: On Navigation Link Enter
- Event: On Navigation Link Exit
- Condition: Compare Child Count
- Instruction: Remap Coordinates
- Instruction: Uniform Scale a Vector3 value
- Instruction: Loop List

FIXED

- Animator null when changing model in Editor
- Audio not taking into account time scale
- Incorrect description on some Input methods
- Changing kernel units while in play-mode
- Global variable access in standalone builds

2.0.5

Released September 17, 2021

NEW

- IsRunning property to Actions and Conditions
- Property to search an object by name
- Memory: Name
- Memory: Tag
- Memory: Layers
- Memory: Is Active
- Memory: Light Color
- Memory: Light Intensity
- Instruction: Change name of Game Object

FIXED

- Point & Click incorrect raycast order
- Point & Click ignore over UI game objects
- Memories not drawing some properties
- Date not parsing using system culture

2.0.4

Released September 16, 2021

NEW

- NavMeshAgent avoidance quality
- NavMeshAgent avoidance priority

FIXED

- Material Sound error when texture is null
- Player not moving without a Main Camera
- Description of Usage Input buttons

2.0.3

Released September 14, 2021

NEW

- Mouse button modifier to Delta Mouse input
- Youtube cover image to welcome screen

FIXED

- Game Creator Hub paths on Windows
- Game Creator Hub package install hierarchy
- Examples Manager installer version check

2.0.2

Released September 13, 2021

NEW

- Option to create impacts for Material Sounds
- Model position offset to Character animation
- Complete & Basic Locomotion States
- Instruction: Toggle Active
- Bool Property: Does not Exist
- Bool Property: Is not Active
- Input: Usage/Crouch
- Input: Usage/Walk

FIXED

- Invalid Hub URL on Windows machines
- Invalid Documentation URL
- Skeleton asset error when using 3D models
- Stop State instruction layer index
- Primary motion input with joystick dead-zone
- Foot IK disabled during gestures with root-motion
- Look IK alignment with target's line of sight
- Animation time scale on characters

2.0.1

Released September 10, 2021

- First release

2. Inventory

2.1 Inventory

Inventory

Using items, combining them, crafting new ones or trading them with other characters is at the heart of many games.

The **Inventory** module has been meticulously crafted to support a wide variety of situations that involve the use and management of items.

Get Inventory

Requirements

The **Inventory** module is an extension of **Game Creator 2** and won't work without it

2.2 Setup

Welcome to getting started with the **Inventory** module. In this section you'll learn how to install this module and get started with the examples which it comes with.

2.2.1 Prepare your Project

Before installing the **Inventory** module, you'll need to either create a new Unity project or open an existing one.

Game Creator

It is important to note that **Game Creator** should be present before attempting to install any module.

2.2.2 Install the Inventory module

If you haven't purchased the **Inventory** module, head to the Asset Store product page and follow the steps to get a copy of this module.

Once you have purchased it, click on Window Package Manager to reveal a window with all your available assets.

Type in the little search field the name of this package and it will prompt you to download and install the latest stable version. Follow the steps and wait till Unity finishes compiling your project.

2.2.3 Examples

We highly recommend checking the examples that come with the **Inventory** module. To install them, click on the *Game Creator* dropdown from the top toolbar and then the *Install* option.

The **Installer** window will appear and you'll be able to manage all examples and template assets you have in your project.

- **Items**: Template items ready to be used in your games
- **UI**: Samples for creating loot user interfaces, inventories, merchants and crafting windows
- **Examples**: A collection of scenes that will help you understand each and every option of the Inventory module, in an organized and tidy way.

Installer Inventory

The **Examples** requires both the **Items** and **UI** extensions in order to work.

There is also an extra *skin* for adventure games that allows to swap the default inventory for a typical old-school point and click inventory.

Dependencies

Clicking on the **Examples** install button will install all dependencies automatically.

Once you have the examples installed, click on the *Select* button or navigate to `Plugins/GameCreator/Installs/Inventory.Examples/`.

Inventory Examples

2.3 Items

2.3.1 Items

Items are in-game objects that can be added to a Bag, and represent the name and description, properties, visual representation, and other information that allows to craft, trade, use and equip them.

Creating an Item

Items are scriptable objects and to create one, you'll need to right click on the *Project Panel* and navigate to *Create* *Game Creator* *Inventory*. *Item*

Item

An **Item** asset will appear, with a list of sections that can be expanded or collapsed so it is easy for the user to modify and organize your items.

The **ID** value is a unique text that represents an item. When creating a new asset, it will be completely unique. However, duplicating an existing item will also duplicate the ID and a red message will appear above stating that there are two items with the same **ID**.

To solve that, expand the field and click on the *Regenerate* button to create a new unique ID. You can also type in a name if you follow a naming convention that ensures that all item IDs are unique.

The **Prefab** field is used to drop/instantiate an item onto the scene. If no prefab is provided, the item will not be instantiated.

INHERITANCE

The **Parent** field allows an item to inherit values from another item, such as Properties and Sockets.

Item A equals Item B?

Comparing two items takes into account their parent-child relationship. For example, if Item *A* inherits from Item *B* and a Condition is trying to determine if an object is equal to another one:

Item A inherits from Item B

- *A* will always return success when comparing if *A* equals *B* or equals *A*.
- *B* will always return success when comparing if *B* equals *B* but not to *A*, because *A* is further down in the inheritance chain.

An Item will always return success if asked whether it is equal to itself or any of its parent items.

INFORMATION

This section allows to define the **Name**. **Description**, **Sprite** representation and **Color** of the **Item**.

Item Information

Localization

All these fields use dynamic properties so their values can be localized.

SHAPE

The shape of an **Item** determines the **Width** and **Height** the item occupies in the inventory bag, if it's a grid-based inventory.

It also determines the **Weight** of the item, in case the bag has a max weight limit.

The **Max Stack** field determines how many of the exact same item can be stacked one on top of another.

Item Information

Stacking restrictions

If an **Item** has one or more Sockets, the **Max Stack** will be automatically restricted to 1, due to technical constraints.

PRICE

An **Item**'s trading value is determined by a Currency asset and a numeric value. This value is the total *pure* one, without any discounts or modifiers applied.

Item Price

One Currency

Note that an item can only be traded using a single currency.

Sockets

The price of an Item that can have other Items attached is the result of the sum of the price of all Items attached, plus the price of the Item itself.

For example, if the item *Sword* has a price of 45 gold and a *Magic Rune* costs 20 gold pieces, the value of the *Sword* with the rune attached will be 65 (45 + 20).

PROPERTIES

Properties define mutable values that an item defines. A Property is a data block that is identified by a name and contains a value and a text that can be used to display information about this item and use it in-game.

Item Properties

Use case of Properties

The most common use-case of a property is definining the attack power of a weapon. One could easily use an item that represents a *Sword* and add a property called `attack` and has a value of 35.

Item Attack Property

See more information about this in the **Properties** page.

SOCKETS

Sockets allow to attach items onto other items. The type of item that can be attached is determined using item *inheritance*.

Item Sockets

Attaching Runes

For example, a socket accepts the item *Rune*, then all items that inherit from the *Rune* item will be accepted.

See more information about this in the **Sockets** page.

EQUIPPING

Some items can be equipped by the wearer (usually the Character with the *Bag* component).

Item Equipment

See more information about this in the **Equipping** page.

USAGE

This section allows to define the behavior of an utility **Item** which can be used at any given time.

Item Use

A usable item can have a finite or infinite amount of usages. The **Consume on Use** toggle defines whether an item is consumed upon use or not.

Finite vs Infinite usages

For example, a *Health Potion* is consumed when used. However a *Whistle* can be used many times.

The **Can Use** conditions are executed every time a runtime item is attempted to be used. If the result is successful, the item is used.

When an **Item** is used, the **On Use** instructions are executed, where **Self** refers to the game object with the *Bag* component the item belongs to, and the **Target** is the references the wearer of the *Bag*.

Execute From Parent

Both the **Can Use** conditions and the **On Use** instructions can optionally execute the parent Item's *Can Use* and *On Use* instructions before executing itself.

This is very useful to avoid repeating the same logic over multiple items. For example, if drinking any potion results in the character executing a particular animation and playing a sound effect, these instructions can be placed in a parent Item called *Potions* so each child Item (Health Potion, Mana Potion, ...) does not have to.

CRAFTING

The **Crafting** section allows to define recipes to create new **Items** as well as dismantle them into multiple ingredients.

Item Crafting

See more information about this in the **Crafting** page.

2.3.2 Properties

Properties are mutable values that compose a runtime item. For example, an **Item**'s attack power, its durability or whether they apply a special effect, such as *Burn*.

Item Properties

Creating a new Property

To create a new **Property** all that needs to be done is to click on the *Add Property* button.

Item Attack Property

The **Property ID** field determines the unique ID of this Property. It is used to identify it, so make sure it's a name that's easy to remember and type.

Is Hidden determines if a Property is hidden in the UI. For more information, see the Hiding Properties section.

The rest of fields are all optional.

- **Icon**: Provides the Property with a Sprite to be used in user interfaces.
- **Color**: Assigns a color to the Property. Useful to differentiate items in user interfaces.
- **Number**: A mutable value that can be used in-game, such as increasing stats.
- **Text** A dynamic value that is usually used to represent the in-game name of the Property.

Mutable vs Immutable

Mutable is a programming concept which means that the value is dynamic and can be changed at runtime. Immutable, in contrast, means that its value can't be changed once a value is assigned.

Inheriting Properties

Checking the **Inherit Properties** toggle found at the top will automatically inherit all properties from its parent(s).

Item Inherit Properties

The value of an inherited **Property** can be overridden by checking its left toggle and changing the field value.

Taking advantage of inheritance

It is very common to have a type of item that shares the same properties with all its child items. Setting a base value for the parent item type will make it much easier to define what each sub-item does.

For example, let's say all *shield* items have a `defense` value. We could add this property on the base item "Shield" and propagate this property to all other shields that inherit from this item, and just change the final value, so a "Wooden Shield" has a lower `defense` value than a `Steel Shield`.

Hiding Properties

When displaying properties in the UI, these can be sequentially displayed, without having to manually set them one by one. If the **Is Hidden** checkbox is ticked, these properties will not be displayed in the user interface.

Item UI Properties

Stuff behind the scenes

This is specially useful when a property represents something that the user should not be aware of.

For example, some items could have the `is-metal` property that determines if an item is a metallic one or not.

2.3.3 Sockets

Sockets allow to attach items onto other items. For example, a Sword can have a socket that allows to attach a *Rune* so it increases its properties.

Item Sockets

Inherit Parent Sockets

Ticking the **Inherit from Parent** checkbox will instruct the **Item** to inherit all **Sockets** from its parent(s).

The socket section is divided in two parts: The part that defines the object attached to the socket, and the part that accepts attachments.

Objects attached to Sockets

The **Socket Prefab** field accepts a prefab game object, which is instantiated when attaching this **Item** onto another Item's **Socket**.

Item Sockets Prefab instance

To configure where the prefab is instantiated, the scene prefab object must have a **Prop** component. This component automatically updates and correctly instantiates the attachment prefabs in the right places, defined in the component's Editor.

Item Sockets Prop component

In this case, the *Metal Shield* has a **Prop** component that inserts the instance of a prefab of any attached rune at the center of the socket.

Configuration of Sockets

To add a **Socket** to an item, simply click on the *Add Socket* button.

Item Sockets new Socket

A **Socket** is defined by a **Base** Item that determines which types of objects can be attached to, and a **Socket ID**, which is used by the *Prop* component.

Base Item

It is important to note that the **Base** item determines the type of item that the Sockets accepts, not the specific item. In the example above, it accepts a *Rune* item, but will also accept any item that has a *Rune* item parent, such as the *Rune of Attack* and *Rune of Defense* included in the examples.

How Properties affect Sockets

When attaching an **Item** onto another one's **Socket**, only their shared **Properties** are added.

Sword with a Rune of Attack

Let's imagine we have a **Sword** with a single *Property*

- `attack` = 10

And a **Rune** with the following *Properties*:

- `attack` = 5
- `defense` = 5

Attaching the Rune to the Sword results in the latter have an `attack` value of 15 (10 + 5), but will ignore the `defense` Property because it is not present in the Sword.

2.3.4 Equipping

To define an equippable Item, the **Is Equippable** checkbox must be ticked, which enables the rest of the options.

Item Equipment

When attempting to equip an **Item**, the Conditions **Can Equip** will first be checked.

If it succeeds, it will instantiate the prefab and execute the **On Equip** instruction list. The **Prefab** field is the game object prefab instantated when equipping this particular Item.

Equipping an item Unequips others

Attempting to equip an **Item** on a slot that is already filled by another **Item** will automatically unequip the current one so the new **Item** can be equipped.

When unequipping an **Item** it will execute the **On Unequip** instruction list.

Equipment

To know more about how to define which **Equipment** slots are available for a character, see Equipment in the Bag section.

When executing the **Can Equip** conditions and the **On Equip** and **On Unequip** instructions:

- The **Self** property references the game object that contains the Item being equipped/unequipped.
- The **Target** references the wearer of the **Bag** (which usually is the same as the Bag object itself).

It is important to note that when a currently equipped item changes the value of one of its Sockets, it will first unequip it, change the **Socket** value and equip it again.

Execute From Parent

If the **Execute From Parent** checkbox is marked, the instructions and conditions from the item's parent item will be executed first (and its parent too, if the parent has *Execute From Parent* marked).

This is very useful to avoid repeating the same logic over multiple items. For example, if the parent type *Swords* contains a **Property** called `attack` and all sub-items from Swords have different `attack` values, there is no need for all sword sub-items to add a **Stat Modifier** with that property.

Instead, the *Swords* item can execute the common logic between all swords, and each sub-item just needs to have the *Execute From Parent* checkbox enabled.

2.3.5 Crafting

The **Crafting** section both defines a way to craft the **Item** being examined, as well as tear it apart and dismantle it into multiple **Items**.

Item Crafting

There are 3 distinct sections inside the **Crafting** tab.

Ingredients

Ingredients are **Items** that can be used to craft the current one, or dismantle it into these ingredients.

To create a new **Ingredient** click on the *Add Ingredient*.

Item Crafting Ingredients

This will create a new ingredient entry with an **Item** field and the amount of those necessary.

Infinite ingredients

There is no limit to the amount of **Ingredients** you can create.

Craft

When attempting to craft an **Item** it will first check if the **Conditions** are sufficient. If so, it will then require a certain amount of **Ingredients** defined.

If there are enough ingredients, these will be subtracted from the Bag.

Empty Conditions

Leaving the **Conditions** field empty will always return success and means there are no conditions to craft it, outside from the **Igredients**.

Once the **Conditions** and **Ingredients** requirements are fulfilled, it will create a new instance of the **Item** and add it to the Bag.

Afterwards, it will call the **Instructions**, in case the designer wants to do something afterwards, such as increasing the proficiency of the Player in crafting.

Dismantle

Dismantling an **Item** is the inverse process of **Crafting**: Instead of creating the current **Item** from a collection of **Ingredients**, it destroys the **Item** and reclaim the **Ingredients**.

Reclaim Probability

When **Dismantling** an **Item** there is a *Reclaim Chance* value that determines the chance to recover each of the **Ingredients**. A value of 1 will always recover all ingredients, while a value of 0.5 will only have a chance to recover around 50% of them.

2.4 Bags

2.4.1 Bags

A **Bag** is a component that can be attached to any game object, and contains **Items** and **Currencies**.

Bag

The **Inventory** module comes with 2 types of **Bags**:

- **List**: Sequentially displays the items one after the other and all occupy the same amount of space.
- **Grid**: Each item occupies a certain amount of cells and these can be manually arranged inside the inventory grid-view.

Recommendation

We recommend sticking with the **List** type, as it is easier to understand and manage. **Grid** inventory systems should be only used by experienced users.

To change the type of **Bag** click on the right-side arrow button and choose the type from the dropdown menu.

Bag Options

A **Bag** can define a **Maximum Weight** and a **Maximum Height**.

- If a maximum height is defined, there is a maximum amount of **Items** it can hold.
- If a maximum weight is defined, if the sum of all **Item**'s weight exceeds the maximum value, the **Bag** is considered overloaded.

Too much weight

It is important to note that a **Bag** can't exceed a maximum amount of height (if any is defined). However, a **Bag** will still accept new **Items** even if its content weight exceeds the maximum weight defined.

Equipment

The **Equipment** field is an optional value that accepts an **Equipment Asset**. If provided, it allows the wearer of the **Bag** to equip **Items**.

To know more about how to configure it, see the Equipment section.

Stock and Wealth

Some **Bags** may contain a certain amount of **Items** and **Currency** by default. For example, a Merchant may have some default stock available.

Bag Stock and Wealth

- Clicking on the *Add Stock* button creates a new **Stock** option that accepts an **Item** and a certain amount of it.
- Clicking on the *Add Wealth* button creates a new **Wealth** option that accepts a **Currency** and its value.

Random Loot

A **Bag** can also be used as a Chest where the player loots its contents. To generate random loot, we recommend using **Loot Tables**, instead of **Stock** options.

Skin UI

The **Skin UI** field is a UI skin asset that displays a different type of user interface that depends on what the purpose of the Bag is. For example, a **Bag** attached to the Player character could display an Inventory UI, while a Chest displays a UI with its content and a button to transfer all of them to the Player's bag.

Custom Skins

To know more about designing custom skins, see the User Interface section.

Wearer

The **Wearer** selector refers to the targeted game object that wears the **Bag**'s equipment. By default it is set to *Self* because the **Bag** is usually attached along the **Character** component. However, if for some reason that is not the case, you can choose which character should be targeted as the equipment wearer.

2.4.2 Equipment

The **Equipment** asset is a scriptable object that lives in the *Project Panel* which contains information about the amount of equippable slots and what bone matches each one of them.

The Equipment Asset

To create an **Equipment** asset, right click on the *Project Panel* and select Create   Game Creator   Inventory   Equipment.

Equipment

An **Equipment** initially has no equipment. Click on the *Add Equipment Slot* button to add a new slot.

Equipment Slot

An equipment slot has a **Base Item** and a **Bone** reference.

- The **Base Item** is the type of **Item** it accepts. For example, if all *Helmets* inherit from a *Head* item, using the *Head* template item will allow to equip all helmets in this slot.
- The **Bone** is a reference to the chosen skeletal bone. If the targeted character is a *Humanoid*, the bone can be picked from a dropdown list. If the character is a non-humanoid, the bone must be referenced using its hierarchy path.

Using the Equipment

Once the **Equipment** asset is created, this can be linked to a **Bag** component so the character knows which equipment slots it has available and where each is mapped to which bone.

Example

For example, the equipment that comes with the **Inventory** module has 4 equippable slots (head, body, right and left hand), plus three extra slots for consumable items:

Equipment Example

We can assign this **Equipment** asset to a **Bag** and all available slots will appear below.

Equipment Example to Bag

After assigning an **Equipment** asset to a **Bag**, the bone that is linked to each slot can be overridden. This is specially useful for non-humanoids, where their bone hierarchy names might not match.

2.4.3 Loot Tables

Loot Tables are probablility sheets that when executed, pick an option from its entries based on a weighted chance and send the chosen element (if any at all) to a **Bag** component.

To create one, right click on the *Project Panel* and select Create Game Creator Inventory Loot Table.

Loot Table

To add a new loot entry, click on the *Add Loot* button. A new entry will appear with the following options:

- **Rate**: A number that represents the weight of the chance. The higher the value, the greater the chance.
- **Loot**: A dropdown that allows to pick an **Item** or a **Currency**.
- **Amount**: The amount picked if the entry is chosen. It can either be a constant value or a random one.

Weight vs Probability

It is important to note the distinction between a **Rate** (or weight) and a probability percentage.

The **Rate** depends on the total sum of all rates from all entries. For example, two entries with a **Rate** of 1 is equal to two entries with a **Rate** of 5. In both cases, the chance of picking them is 50%.

Optionally there is a **No Drop Rate** field that enables the **Loot Table** to pick nothing.

To execute a **Loot Table** it is as easy as using the **Loot Table** instruction and choosing both a **Loot Table** asset and the targeted **Bag** where the items/currency will be sent to.

Loot Table Instruction

Run multiple times

Note that each time a **Loot Table** is executed, it picks one entry from the table. A **Loot Table** can be used multiple times in sequence to fill, for example, a Chest with multiple items.

Chest with Random Loot

One easy way to randomize the loot of a level is to populate them with a Chest prefab that has an **On Start** Trigger. This Trigger then runs one or more times a **Loot Table** and sends its contents to the Chest's **Bag** component.

This allows to very easily populate all the Chests of a level with different content, while at the same time controlling the kind of content they contain.

2.5 Currencies

To determine the value of an **Item**, Game Creator uses the concept of **Currency**.

A **Currency** is an asset that contains one or more **Coins**. Each **Coin** has a value relative to a single unit. To create one, right click on the *Project Panel* and select Create Game Creator Inventory Currency.

Single Currency

Most games make use of a single **Currency**. However, some mobile games and hard-core resource management games use multiple ones.

Currency

In the example above, the **Currency** just has a single **Coin** called **Gold** which value is 1. This is the most simple currency one can create and it's the most commonly used in most games.

No decimals

It is important to note that a currency cannot have a decimal value. If you wish to represent a value with 2 decimals, one can multiple the value x100 and then shift the comma two units left.

However, some games make use of a multi-coin **Currency** where each coin represents a different value.

Copper, Silver and Gold

Let's say we are making a game where the currency has three different coins, each with a different value:

- A Copper coin is the smallest one.
- A Silver coin is equal to 25 of Copper coins.
- A Gold coin is equal to 5 Silver coins.

In that case, we would create a **Currency** asset with three coins:

- **Copper**: Is the smallest possible value, so it has a value of **1**.
- **Silver**: Is equal to 25 copper coins, so it has a value of **25**.
- **Gold**: Is equal to 5 silver coins, which cost 25 copper coins each, so it has a value of **125**.

Currency In-Game

It is important to note that when adding or subtracting a value of a particular **Currency** the value used is relative to the unit. Following the example above, if we want to give one *Gold Coin* to the Player, we simply increase its wealth by **125**.

2.6 Merchants

The **Inventory** module comes with a built-in system that allows two **Bags** to trade their contents in exchange for a specified Currency.

Merchant

2.6.1 Merchant Component

To initiate a trade between two **Bags**, one of them (the merchant) must have a **Merchant** component attached along a **Bag** component.

- The **Bag** component provides the stock of items available.
- The **Merchant** component determines the type of transactions made.

Merchant Component

Merchant Info

The *Merchant Info* section allows to give the **Merchant** a name and a description. This is completely optional, but can be useful to display the type of trading made by a certain Merchant.

Example

For example, having a merchant called *Herbologist* already gives a clue of the type of **Items** this merchant trades with.

Configuration

- **Infinite Currency**: If checked, the Merchant will have an infinite amount of currency supply to buy Items from the client (Player). Otherwise it will use the Bag's wealth.
- **Infinite Stock**: If checked, the number of available Items will not decrease after the client (Player) purchases them. Otherwise, the available stock decreases with each purchase made.
- **Allow Buy Back**: If checked, every Item sold by the client (Player) is automatically added to the Merchant's stock. Otherwise, any Item sold cannot be recovered.
- **Sell Niche Type**: If checked, it allows to filter the type of Items sold by this merchant, regardless of its Bag content. For example, if a Merchant only sells *Herbs*, even if its Bag contains a Sword, it will not be available for sale.

The **Buy Rate** is the discount coefficient that the Merchant provides when buying Items from the client (Player). A value of *1* indicates the Items sold have no discount. To provide a 90% discount on all Items, this field should be set to *0.9*.

The **Sell Rate** is the coefficient applied when the Merchant purchases Items from the client (Player). In most games, the selling price of an Item is lower (commonly half the price) than its real one.

The **Bag** field is a reference to the Bag component from where the Merchant takes its stock.

Reference a Bag

If your Bag is placed along another game object, you can change the value of this field from *Self* to *Bag* and manually reference the correct object.

Skin UI is the user interface skin used by this merchant.

2.7 Tinkering

Tinkering

The process of transforming items into other ones is called **Tinkering**, which includes:

- **Crafting**: Creating a single item from multiple ones.
- **Dismantling**: Destroying an item in order to recover multiple ones.

To open a **Crafting** or **Dismantle** interface, use the **Open Tinker UI** instruction.

Open Tinkering UI

This instruction uses a **Tinker Skin** that determines whether the UI crafts new items or dismantles existing ones.

The **Input Bag** and **Output Bag** are the bags used by the tinker process. In most games, both bag references will match, but there might be some cases where the game outputs the new items onto another bag, from where the player can pick them.

The **Filter Item** field determines the type of items displayed.

Filtering by Type

Blacksmithing and brewing potions use the exact same process. The only difference between an Alchemy station and a Forge is that the first one filters the types of items to craft by *Potion* type and the latter filters by *Equipment* type.

To know more about how to create your own custom tinkering UI elements, see the Tinker UI section and the examples that come with the **Inventory** module.

2.8 Visual Scripting

2.8.1 Visual Scripting

The **Inventory** module symbiotically works with **Game Creator** and the rest of its modules using its visual scripting tools.

- **Instructions**
- **Conditions**
- **Events**

Each scripting node allows other modules to use any **Inventory** feature, and adds a list of **Properties** ready to be used by other interactive elements.

2.8.2 Conditions

Conditions

SUB CATEGORIES

- Inventory

Inventory

INVENTORY

Sub Categories

- Equipment
- Merchant
- Tinker
- Ui

Conditions

- Can Add
- Has Item
- Is Overloaded
- Is Type Of Item
- Is Usable

CAN ADD

```
Inventory » Can Add
```

Description

Returns true if the item can be added to the Bag component

Parameters

Name	Description
Item	The item type to add
To Bag	The target destination Bag

Keywords

`Inventory` `Give` `Put` `Set`

HAS ITEM

```
Inventory » Has Item
```

Description

Returns true if the Bag component contains, at least, the specified amount of an item

Parameters

Name	Description
Item	The item type to check
Amount	The minimum amount of a particular item
Bag	The targeted Bag

Keywords

`Inventory` `Contains` `Includes` `Wears` `Amount`

IS OVERLOADED

```
Inventory » Is Overloaded
```

Description

Returns true if the Bag's maximum weight is surpassed

Parameters

Name	Description
Bag	The Bag component

Keywords

`Inventory` `Weight` `Amount`

IS TYPE OF ITEM

```
Inventory » Is Type of Item
```

Description

Returns true if the item is equal or a sub-type of another one

Parameters

Name	Description
Item	The item source
Compare To	The item compared to

Keywords

`Inventory` `Compare`

IS USABLE

```
Inventory » Is Usable
```

Description

Returns true if the chosen Item can be used

Parameters

Name	Description
Item	The item type to check

Keywords

`Inventory` `Consume` `Drink`

EQUIPMENT

Equipment

Conditions

- Can Equip
- Is Equippable
- Is Equipped

Can Equip

```
Inventory » Equipment » Can Equip
```

Description

Returns true if the chosen Item can be equipped by the targeted Bag's wearer

Parameters

Name	Description
Item	The item type to check
Bag	The targeted Bag

Keywords

`Inventory` `Contains` `Includes` `Wears` `Amount`

Is Equippable

```
Inventory » Equipment » Is Equippable
```

Description

Returns true if the chosen Item can be equipped

Parameters

Name	Description
Item	The item type to check

Keywords

`Inventory` `Wear` `Equip`

Is Equipped

```
Inventory » Equipment » Is Equipped
```

Description

Returns true if the Bag's wearer has an Item of that type currently equipped

Parameters

Name	Description
Item	The item type to check
Bag	The targeted Bag

Keywords

`Inventory` `Wears`

MERCHANT

Merchant

Conditions

- Can Buy
- Can Sell

Can Buy

```
Inventory » Merchant » Can Buy
```

Description

Returns true if the item can be bought from a Merchant

Parameters

Name	Description
From Merchant	The Merchant component
Item	The item type attempted to purchase
To Bag	The destination Bag for the item

Keywords

`Inventory` `Purchase` `Get` `Bargain` `Haggle`

Can Sell

```
Inventory » Merchant » Can Sell
```

Description

Returns true if the item can be sold to a Merchant

Parameters

Name	Description
From Bag	The Bag where the item is sold
Item	The item type attempted to sell
To Merchant	The Merchant target

Keywords

`Inventory` `Vend` `Trade` `Exchange` `Part` `Bargain` `Haggle`

TINKER

Tinker

Conditions

- Can Craft
- Can Dismantle
- Is Craftable
- Is Dismantable

Can Craft

```
Inventory » Tinker » Can Craft
```

Description

Returns true if the item can be crafted

Parameters

Name	Description
From Bag	The Bag where ingredients are picked
Item	The item type attempted to craft
To Bag	The target destination Bag after creating the new Item

Keywords

`Inventory` `Create` `Make` `Cook` `Smith` `Combine` `Assemble`

Can Dismantle

```
Inventory » Tinker » Can Dismantle
```

Description

Returns true if the item can be dismantled

Parameters

Name	Description
From Bag	The Bag where item is picked
Item	The item type attempted to dismantle
To Bag	The destination Bag for all ingredients after dismantling the Item

Keywords

`Inventory` `Apart` `Disassemble` `Deconstruct` `Tear` `Separate`

Is Craftable

```
Inventory » Tinker » Is Craftable
```

Description

Returns true if the chosen Item can be crafted

Parameters

Name	Description
Item	The item type to check

Keywords

`Inventory` `Create` `Forge` `Alchemy` `Brew`

Is Dismantable

```
Inventory » Tinker » Is Dismantable
```

Description

Returns true if the chosen Item can be dismantled

Parameters

Name	Description
Item	The item type to check

Keywords

`Inventory` `Destroy` `Tear` `Break`

UI

Ui

Conditions

- Is Bag Ui Open
- Is Merchant Ui Open
- Is Tinker Ui Open

Is Bag UI Open

```
Inventory » UI » Is Bag UI Open
```

Description

Returns true if the there is a Bag UI open

Keywords

`Inventory` `Close` `Stash` `Loot` `Container` `Chest`

Is Merchant UI Open

```
Inventory » UI » Is Merchant UI Open
```

Description

Returns true if the there is a Merchant UI open

Keywords

`Shop` `Exchange` `Trader`

Is Tinker UI Open

```
Inventory » UI » Is Tinker UI Open
```

Description

Returns true if the there is a Crafting/Dismantling UI open

Keywords

`Close` `Craft` `Dismantle` `Assemble` `Disassemble` `Smith` `Upgrade`

2.8.3 Events

Events

SUB CATEGORIES

- `Inventory`

Inventory

INVENTORY

Sub Categories

- Equipment
- Merchant
- Sockets
- Tinker
- Ui

Events

- On Add
- On Drop Item
- On Instantiate Item
- On Remove

ON ADD

```
Inventory » On Add
```

Description

Executes after adding an item to the specified Bag

Keywords

`Bag` `Inventory` `Item` `Add`

ON DROP ITEM

```
Inventory » On Drop Item
```

Description

Detects when a Bag's item is dropped onto the Trigger

ON INSTANTIATE ITEM

```
Inventory » On Instantiate Item
```

Description

```
Executes after dropping an item from a Bag to the scene
```

ON REMOVE

```
Inventory » On Remove
```

Description

Executes after removing an item from the specified Bag

Keywords

`Bag` `Inventory` `Item` `Take`

EQUIPMENT

Equipment

Events

- On Equip
- On Unequip

On Equip

```
Inventory » Equipment » On Equip
```

Description

Executes after equipping an item from the specified Bag

Keywords

`Bag` `Inventory` `Item` `Add` `Wear`

On Unequip

```
Inventory » Equipment » On Unequip
```

Description

Executes after unequipping an item from the specified Bag

Keywords

`Bag` `Inventory` `Item` `Remove` `Wear`

MERCHANT

Merchant

Events

- On Buy
- On Sell

On Buy

```
Inventory » Merchant » On Buy
```

Description

Executes after successfully purchasing an item from any Merchant

On Sell

```
Inventory » Merchant » On Sell
```

Description

Executes after successfully selling an item to any Merchant

SOCKETS

Sockets

Events

- On Socket Attach
- On Socket Detach

On Socket Attach

```
Inventory » Sockets » On Socket Attach
```

Description

Detects when an Item's Socket gets another Item attached

On Socket Detach

```
Inventory » Sockets » On Socket Detach
```

Description

Detects when an Item is detached from another Item's Socket

TINKER

Tinker

Events

- On Craft
- On Dismantle

On Craft

```
Inventory » Tinker » On Craft
```

Description

Executes right after successfully crafting any item

On Dismantle

```
Inventory » Tinker » On Dismantle
```

Description

Executes right after successfully dismantling any item

UI

Ui

Events

- On Close Bag Ui
- On Close Merchant Ui
- On Close Tinker Ui
- On Open Bag Ui
- On Open Merchant Ui
- On Open Tinker Ui

On Close Bag UI

```
Inventory » UI » On Close Bag UI
```

Description

Detects when a Bag UI is closed

On Close Merchant UI

```
Inventory » UI » On Close Merchant UI
```

Description

Detects when a Merchant UI is closed

On Close Tinker UI

```
Inventory » UI » On Close Tinker UI
```

Description

Detects when a Tinker UI is closed

On Open Bag UI

```
Inventory » UI » On Open Bag UI
```

Description

Detects when a Bag UI is opened

On Open Merchant UI

```
Inventory » UI » On Open Merchant UI
```

Description

Detects when a Merchant UI is opened

On Open Tinker UI

```
Inventory » UI » On Open Tinker UI
```

Description

Detects when a Tinker UI is opened

2.8.4 Instructions

Instructions

SUB CATEGORIES

- `Inventory`

Inventory

INVENTORY

Sub Categories

- Equipment
- Ui

Instructions

- Add Item
- Change Currency
- Instantiate Item
- Loot Table
- Move Content To Bag
- Move Wealth To Bag
- Remove Item

ADD ITEM

```
Inventory » Add Item
```

Description

Creates a new item and adds it to the specified Bag

Parameters

Name	Description
Item	The type of item created
Bag	The targeted Bag component

Keywords

`Bag` `Inventory` `Container` `Stash` `Give` `Take` `Borrow` `Lend` `Buy` `Purchase` `Sell` `Steal` `Rob`

CHANGE CURRENCY

```
Inventory » Change Currency
```

Description

Modifies the value of a Bag's currency

Parameters

Name	Description
Currency	The currency type to modify
Amount	The value and operation performed
Bag	The targeted Bag component

Keywords

`Bag` `Inventory` `Container` `Stash` `Give` `Take` `Borrow` `Lend` `Buy` `Purchase` `Sell` `Steal` `Rob` `Coin` `Cash` `Bill` `Value` `Money`

INSTANTIATE ITEM

```
Inventory » Instantiate Item
```

Description

Instantiates the prefab of an item on the scene

Parameters

Name	Description
Item	The type of item created
Location	The position and rotation where the item instance is placed

Keywords

`Drop` `Inventory` `Instance`

LOOT TABLE

```
Inventory » Loot Table
```

Description

Picks a random choice from a Loot Table and sends it to the specified Bag

Parameters

Name	Description
Loot Table	The Loot Table that generates the Item instance
Bag	The targeted Bag component

Keywords

`Bag` `Inventory` `Container` `Stash` `Give` `Take` `Borrow` `Lend` `Corpse` `Generate`

MOVE CONTENT TO BAG

```
Inventory » Move Content to Bag
```

Description

Moves all the contents of a Bag to another Bag

Parameters

Name	Description
From Bag	The Bag component where its contents are removed
To Bag	The targeted Bag component where the contents end up

Keywords

`Bag` `Inventory` `Container` `Stash` `Chest` `Take` `All` `Give` `Take` `Borrow` `Lend` `Buy` `Purchase` `Sell` `Steal` `Rob`

MOVE WEALTH TO BAG

```
Inventory » Move Wealth to Bag
```

Description

Moves all wealth from one Bag to another one

Parameters

Name	Description
From Bag	The Bag component where its wealth is taken from
To Bag	The targeted Bag component where the wealth ends up

Keywords

`Bag` `Inventory` `Container` `Stash` `Chest` `Take` `All` `Give` `Take` `Borrow` `Lend` `Buy` `Purchase` `Sell` `Steal` `Rob` `Currency` `Cash` `Money` `Coins`

REMOVE ITEM

```
Inventory » Remove Item
```

Description

Removes an Item from the specified Bag

Parameters

Name	Description
Item	The parent type of item to be removed
Bag	The targeted Bag component

Keywords

`Bag` `Inventory` `Container` `Stash` `Give` `Take` `Borrow` `Lend` `Buy` `Purchase` `Sell` `Steal` `Rob`

EQUIPMENT

Equipment

Instructions

- Equip Item
- Unequip Item

Equip Item

```
Inventory » Equipment » Equip Item
```

Description

Equips an Item from the Bag that inherits from the specified type

Parameters

Name	Description
Item	The parent type of item to equip
Bag	The targeted Bag component

Keywords

`Bag` `Inventory` `Equipment` `Put` `Wear` `Inventory` `Wield`

Unequip Item

```
Inventory » Equipment » Unequip Item
```

Description

Unequip an Item from the Bag that inherits from the specified type

Parameters

Name	Description
Item	The parent type of item to equip
Bag	The targeted Bag component

Keywords

`Bag` `Inventory` `Equipment` `Take` `Sheathe` `Inventory` `Remove`

UI

Ui

Instructions

- Open Bag Ui
- Open Merchant Ui
- Open Tinker Ui
- Set Bag Ui

Open Bag UI

```
Inventory » UI » Open Bag UI
```

Description

Opens an inventory UI of a specific Bag

Parameters

Name	Description
Bag	The Bag component
Wait to Close	If the Instruction waits until the UI closes

Keywords

`Item` `Inventory` `Catalogue` `Content` `Sort` `Equipment` `Hotbar` `Consume`

Open Merchant UI

```
Inventory » UI » Open Merchant UI
```

Description

Opens a trading window for a specific Merchant

Parameters

Name	Description
Merchant	The currency type to modify
Client Bag	The client's Bag component
Wait to Close	If the Instruction waits until the UI closes

Keywords

`Trade` `Merchant` `Shop` `Buy` `Sell` `Junk`

Open Tinker UI

```
Inventory » UI » Open Tinker UI
```

Description

Opens an Tinkering UI for a specific Bag

Parameters

Name	Description
Tinker Skin	The skin that is used to display the UI
Input Bag	The Bag component where items are chosen
Output Bag	The Bag component where new items are placed
Wait to Close	If the Instruction waits until the UI closes

Keywords

`Craft` `Make` `Create` `Dismantle` `Disassemble` `Torn` `Alchemy` `Blacksmith`

Set Bag UI

```
Inventory » UI » Set Bag UI
```

Description

Changes the targeted Bag of a Bag UI component

Parameters

Name	Description
Bag UI	The Bag UI that changes its target
Bag	The new Bag component

2.9 User Interface

2.9.1 User Interface

The **Inventory** module comes with a large collection of components so you have complete freedom to make your own game UI.

UI Examples

To get started, it is recommended to install the UI examples that come with this module, which include a HUD, a classic inventory, as well as a merchant and crafting/dismantle interfaces.

Skins

Skins are assets that contain a prefab with a specific UI component. There are three types of skins:

- **Bag Skins**: These skins are linked to Bag components and require a Bag UI component at the root of the prefab.
- **Merchant Skins**: These skins are linked to Merchant components and require a Merchant UI component at the root of the prefab.
- **Tinker Skin**: These skins are directly accessed when opening a Craft/Dismantle interface. They require a Tinker UI component at the root of the prefab.

Skins

The Inventory module comes with a lot of components that make it very easy to build a user interface that synchronizes with a Bag, Merchant or Tinkering object. Each component has a very specific use-case that is covered in each relevant sub-section.

Component Dependency

Some UI components depend on others that feed information to them. For example, the **Coin UI** component depends on the **Price UI** component, that instantiates and reuses a prefab with a Coin UI component for each currency coin.

2.9.2 Bag UI

The **Bag UI** is the root component for any UI prefab that displays information about a Bag. There are two types of **Bag UI** components, which depend on the type of **Bag** used:

- **Bag List UI**: Used for list-like Bags
- **Bag Grid UI**: Used for grid-like Bags

Lists vs Grids

This documentation focuses on Bags with a List-type, as they are most commonly used. The use of a Grid-type requires a deeper understanding on how each UI component works, but the concepts and components used are mostly the same.

Bag List UI

Prefab Cell is a prefab game object with a **Bag Cell UI** component. This component is automatically instantiated and updated by its parent, for each Item in the Bag displayed.

Filter by Parent is an optional Item-type filter. If none is provided, it will display all Items of all types. This is particularly useful when creating tabs or sections.

Content is the parent game object where all prefab cells will be instantiated - One for each Item in the Bag.

Can Drop Outside determines whether an Item can be dragged outside of the UI canvas to drop it into the scene world.

Max Drop Distance determines the maximum distance that an Item can be dropped from the Bag object.

Drop Amount determines whether a dropped object removes the whole stack of objects or just the top-most.

Dropping Items

Note that only Items that have a **Prefab** object in their Item definition can be dropped.

Components

There are a few extra components that can synchronize a **Bag**'s information with UI controls, which can either be linked to a Bag, or to the Bag linked to a **Bag List**/**Grid UI** component.

CELL UI

This component is automatically set up and refreshed by its **Bag List UI** or **Bag Grid UI** parent component.

Bag Cell UI

The **Cell Info** section contains an optional collection of UI control fields that can be plugged in order to be updated when the **Item**(s) associated with this inventory cell change.

Graphic component required

This component requires a **Graphic** component (either an Image or a Text) in order to receive input events, such as clicks and drags.

The **Merchant Info** field is optional and only useful if the **Bag Cell UI** component is part of a Merchant UI component.

The **Can Drag** toggle determines whether an **Item** can be dragged and dropped.

On Drop and **On Select** defines the behavior when this Item cell is dragged and dropped, and when it is focused.

Selected Cell UI, Socket UI and Property UI

When a **Bag Cell UI** is selected, any **Selected Cell UI** component will be refreshed with the information of the currently selected cell. This allows to display information about a particular cell outside from the cell itself.

In both **Bag Cell UI** and **Selected Cell UI** components, one can create a prefab with a **Socket UI**/**Property UI** component that displays the current sockets/properties.

EQUIP UI

This component is used for equipping items and assigning consumables to hotbars.

Bag Equip UI

The **Bag** and **Equipment** fields determine the targeted Bag and the equipment slot that this refers to.

There are two main sections:

- **Base UI**: Allows to display a collection of optional controls that reference the base-type **Item**
- **Equipped UI**: Allows to display a collection of optional controls that reference the currently equipped **Item** (if there is one).

The rest of the fields define the behavior when the **Bag Equip UI** is interacted with.

WEALTH UI

The **Bag Wealth UI** component is used to display the selected **Currency** and how much of it the Bag carries.

Bag Wealth UI

This component requires a prefab that represents each coin's **Currency** value, and must contain the **Coin UI** component.

WEIGHT

This component displays the current and max weight of the selected Bag.

Bag Weight UI

2.9.3 Merchant UI

The **Merchant UI** is a very simple component that acts as a middle-man between two **Bag UI** components - Allowing both ends to transfer or trade their contents based on a particular set of rules.

Merchant UI

This component has two fields at the top:

- **Merchant Bag UI**: A Bag UI component that contains information about the Bag that represents the merchant.
- **Client Bag UI**: A Bag UI component that contains information about the Bag that represents the client (usually, the Player).

Trading

When a **Bag UI** component is referenced by a **Merchant UI**, the **Bag UI** obtains information about the trading rules, which cascade and can be accessed from the *Merchant Info* section on a Bag Cell UI component.

There are also a couple of **Instruction** lists at the bottom that are executed when this **Merchant UI** executes a transaction.

Buy and Sell

Note that *Buy* and *Sell* are from the client's perspective (aka the Player). So the *On Buy* instructions run when the client purchases an item, and *On Sell* run when the client sells an item.

2.9.4 Tinker UI

Tinkering involves both **Crafting** and **Dismantling** items, and the **Tinker UI** component allows to display a list of UI controls that handle the transformation.

Tinker UI

There are two distinct sections in this component, but both work very similarly: There is a container object where all available recipes/items are displayed, from where the user can pick one and begin the transformation process.

- **Filter By Parent** allows to display only those **Items** that inherit, at some point, from the selected type. If none is set, it will not filter any items.
- **Selected UI** references a **Crafting UI** or **Dismantling UI** component, which is used to display the currently selected **Item** from the list.

The following two fields allow to populate the list of **Items**:

- The **Content** field must reference a UI game object which will be populated by an instance of a prefab for each element in the list.
- The **Prefab** field references a prefab game object, which will be instantiated in the container object.

Prefab requires component

The **Prefab** field requires a **Crafting Item UI** or a **Dismantling Item UI** component in order to work. This will be automatically synchronized and refreshed with the information provided by the Tinker UI list.

Crafting Item UI

The **Crafting Item UI** component is both used when selecting an **Item** from the recipe list as well as to display each entry from the list.

Crafting Item UI

This component is automatically refreshed with the correct information about the current **Item**.

On Start & On Complete

The **On Start** and **On Complete** instructions are executed when either a dismantle or crafting operation starts, and successfully finishes. This is the perfect place to add sound and visual effects.

Dismantling Item UI

The **Dismantling Item UI** component is both used when selecting an **Item** from the available item list as well as to display each entry from the list.

Dismantling Item UI

This component is automatically refreshed with the correct information about the current **Item**.

Recover Chance is a value between 0 and 1 that determines the chance to recover each and every one of the ingredients that constitute the dismantled **Item**.

2.10 Releases

2.10.1 Releases

Game Creator 2

The **Inventory 2** module is a **Game Creator 2** extension and will not work without it.

2.1.2 (Next Release)

Release Pending

NEW

- Items have usage conditions
- Equip/Unequip can inherit logic from its parents
- Using Items can inherit logic from its parents
- Condition: Can Equip to Bag
- Condition: Is Equippable
- Condition: Is Equipped
- Condition: Is Craftable
- Condition: Is Dismantable
- Condition: Is Usable
- Instruction: Change target Bag of Bag UI
- UI: Bag UI can have a default Bag
- UI: Properties with a value of 0 can be skipped
- Properties: Access to recent socketed Items

CHANGED

- Item price increments with socketed Items
- Compatibility with Game Creator 2.3.15

2.0.1

Released January 12, 2022

- First release

3. Dialogue

3.1 Dialogue

WIP

This module is currently under developement

4. Stats

4.1 Stats

Stats

Nearly all games one can play has some kind of *Stat* system; Whether it is a simple health bar with a fixed amount of hit points or a complex RPG with dozens of stats that influence the progress of the player and the outcome of any interaction.

The **Stats** module has been envisioned to help game designers more naturally and easily architect their games.

Get Stats

Requirements

The **Stats** module is an extension of **Game Creator 2** and won't work without it

4.2 Setup

Welcome to getting started with the **Stats** module. In this section you'll learn how to install this module and get started with the examples which it comes with.

4.2.1 Prepare your Project

Before installing the **Stats** module, you'll need to either create a new Unity project or open an existing one.

Game Creator

It is important to note that **Game Creator** should be present before attempting to install any module.

4.2.2 Install the Stats module

If you haven't purchased the **Stats** module, head to the Asset Store product page and follow the steps to get a copy of this module.

Once you have purchased it, click on Window Package Manager to reveal a window with all your available assets.

Type in the little search field the name of this package and it will prompt you to download and install the latest stable version. Follow the steps and wait till Unity finishes compiling your project.

4.2.3 Examples

We highly recommend checking the examples that come with the **Stats** module. To install them, click on the *Game Creator* dropdown from the top toolbar and then the *Install* option.

The **Installer** window will appear and you'll be able to manage all examples and template assets you have in your project.

- **Examples**: A collection of scenes with different use-case scenarios
- **Classes**: A template with Stats, Attributes and Classes to kickstart your game
- **UI**: Samples for creating a HUD and a Character Stats menu

Installer Stats

The **Examples** requires both the **Classes** and **UI** extensions in order to work.

Dependencies

Clicking on the **Examples** install button will install all dependencies automatically.

Once you have the examples installed, click on the *Select* button or navigate to `Plugins/GameCreator/Installs/Stats.Examples/`.

Stats Examples

4.3 Classes

4.3.1 Classes

Taking inspiration from classic pen and paper RPG games, the **Stats** module lets you create character **Classes** which contain a collection of **Stats** and **Attributes**. On the other end, **Classes** can be assigned to any number of characters or game objects using the **Traits** component.

Stats Overview

Example

This concepts are more easily understood with an example. Let's say we want to create a Warrior character. In this case, we would create a **Class** called "Warrior" which would contain the following **Attributes**:

- `Health`
- `Stamina`

And the following **Stats**:

- `Strength`
- `Constitution`

Now that we have the Warrior class, we can create a scene Character with the **Traits** component and assign it the Warrior **Class** defined above. This same class can be reused for other characters, such as enemies and NPCs.

4.3.2 Stats

Stats are objects that represent a particular numeric trait of a character. This value can evolve throughout the whole game and its final value can be modified using a **Formula**.

Common Stats

Common stat values on games are `strength`, `dexterity`, `wisdom`, `luck`, ...

To create a **Stat** asset, right click on the *Project panel* folder you want to create it and select Create Game Creator Stats Stat.

Stat Asset

The **ID** value must be unique throughout the whole project and it is used to identify this particular numeric trait. It is also used in **Formulas** so be sure to give it a name that's easy to remember.

Naming Stats

We recommend sticking to acronyms or short and single worded names. For example, if the **Stat** represents the strength of the character, its ID should be `str` or `strength`.

The **Base Value** is the numeric value that the **Stat** starts with. It is worth noting this value is not necessarily the final value of the **Stat**, just a mutable numeric value.

The final value of a **Stat** is calculated applying a **Formula**. If none asset is provided, the final value is simply the **Base Value**.

Base and Formula

Let's say we have a stat with a **Base** value of 100 and a **Formula** that multiplies this value by the level (another stat value) of the character. In this case, the resulting final value of the stat would depend on the character's level.

For example, if the character is at level 1, the value would be 100 (100 * 1). At level 2, it would be 200 (100 * 2), at level 3 it would be 300 (100 * 3), etc...

The **UI** dropdown contains a list of fields that can be used to display information about this particular **Stat** on the game scene, including a name, acronym, description, color and icon.

4.3.3 Attributes

Attributes are objects that represent a numeric trait of a character, but its value is clamped between a min/max range.

Common Attributes

The most common attribute is the `health` of a character. Its value could a value clamped between 0 and 100.

To create an **Attribute** asset, right click on the *Project panel* folder you want to create it and select Create Game Creator Stats Attribute.

Attribute Asset

The **ID** value must be unique throughout the whole project and it is used to identify this particular numeric trait. It is also used in **Formulas** so be sure to give it a name that's easy to remember.

Naming Attributes

We recommend sticking to acronyms or short and single worded names. For example, if the **Attribute** represents the health of the character, its ID should be `hp` or `health`.

The **Min Value** and **Max Value** are numeric values that represent the minimum and maximum range of the value. The **Max Value** comes from a Stat as this value can change at runtime.

Max Value is a Stat

For example, if the attribute represents the health of the player, levelling up could increase the maximum health. In this case, increasing a **Stat** called "Max_Health" would automatically increase the max cap of the health **Attribute**.

The **Start Percent** field defines the percent at which the character's attribute starts. By default most games should start with their attributes completely filled.

The **UI** dropdown contains a list of fields that can be used to display information about this particular **Attribute** on the game scene, including a name, acronym, description, color and icon.

4.3.4 Classes

Classes are objects that represent a type of character or object with RPG traits, and contains a list of Stats and Attributes.

Classes in an RPG

Just like in most RPGs, a **Class** defines a type character with different values. For example, a *Mage* will have the same **Stats** and **Attributes** as a *Knight*, but their values and progression may differ, making the *Mage* grow his magic abilities at a much higher rate than the *Knight*, which focuses on its physical ones.

Class

To create a **Class** asset, right click on the *Project panel* folder you want to create it and select Create Game Creator Stats Class.

By default, a **Class** has an empty list of fields. The image below represents a **Class** filled with a collection of **Stats** and **Attributes**.

Class Asset

Eye Icon

The eye icon that appears next to all Attributes and Stats is a button that can be toggled. It has no impact on the game whatsoever. Instead it hides the option from the Traits component. This is useful if you have hundreds of Stats and Attributes and want to keep the important ones at a glance.

The **Class** and **Description** fields are used to display information about the current class in the game's user interface.

Attributes

The **Attributes** list defines all the attributes linked to this particular class.

To add a new **Attribute**, click on the "Add Attribute" button at the bottom and pick (or drag and drop) the desired **Attribute** asset.

Class Attributes

In this section, the selcted **Attribute**'s starting percent can be overriden, in case a particular **Class** has a different starting value than another.

Stats

The **Stats** list defines all the stats linked to this particular class, including the ones that define the max cap of Attributes.

To add a new **Stat**, click on the "Add Stat" button at the bottom and pick (or drag and drop) the desired **Stat** asset.

Class Stats

In this section, the selected **Stat** base value and formula can be overriden.

Override Stat Base and Formula

When creating multiple RPG classes, such as Mages, Knights and Archers, it's a good practice to have the same Attributes and Stats. In order to change their progression rates, their values can be overriden within the **Class** asset itself.

For example, the `wisdom` base stat value may have a much higher one in a *Mage* class than in a *Knight*.

4.3.5 Traits

Traits are components that link a Class asset with a scene game object.

Game Objects with Traits

It is important to note that, although Characters will most likely be the objects with a **Traits** component, these can be attached to any game object.

For example, to assign the *Player* with the *Knight* **Class** one just has to click on the *Player* game object "Add Component" button at the bottom of the *Inspector* and look for the **Traits** component.

Traits in Editor

Once the *Player* has the **Traits** component a message appears prompting to assign it a **Class** asset.

Traits missing Class asset

Drag and drop any **Class** asset onto the designated field and it will change its appearance to display the asset's information.

Traits with Class asset

Each **Attribute** and **Stat** can be expanded and their values can be overriden, just like in the **Class** asset.

Traits at Runtime

Once the game object has a **Traits** component linked with a **Class** asset, it is ready to interact in play mode.

To help the designer understand what's happening in play mode and debug any possible problems, the **Traits** component changes its *Inspector* appearance to display real-time information about its current Attribute and Stat values.

Traits in Playmode

4.4 Formulas

Formulas are at the core of the Stats module; They allow the game designer to elaborate simple or complex systems that intertwine different stat and attribute values.

Math Expressions

Formulas are written using math expressions. For example the following formula:

```
source.stat[attack] - target.stat[defense]
```

Can be used to calculate the damage dealt to an enemy. It calculates the output taking into account the `attack` stat from the player and subtracting the `defense` stat from the enemy.

It is up to the game designer defining how simple or complex these formulas should be.

4.4.1 Creating a Formula

To create a **Formula** asset, right click on the *Project panel* folder you want to create it and select Create Game Creator Stats Formula.

Formula Asset

The **Formula** asset has a text field at the top, where the the math expression can be written.

The *Help* section contains a list of all possible symbols that can be used. For example, to retrieve the final value of a **Stat** called "strength" from the caller, use the `source.stat[strength]` symbol.

Each section can be expanded and collapsed to keep the important information at a glance.

Formula Help

Symbols

Check the list of all symbols at the end of this page.

The *Table* field is an optional one, that can be used to reference a Table asset from within the formula expression.

4.4.2 Symbols

A formula expression is composed of a series of symbols, joined together by a math expression, such as the sum, subtraction, product and division.

For example, the attack power of a character could be it's base strength value multiplied by its level. In this case, the expression would be:

```
source.base[strength] * source.stat[level]
```

Stats

This section covers all values found inside a game object with a **Traits** component. A stat or attribute can either come from the **Source** object or the **Target** object. For example, when calculating the damage dealt to an enemy, **Source** references the attacker and **Target** the attacked object.

Source and Target

In some cases, there may be no distinction between source and target. For example, when calculating the level of a character. In this case, we recommend ignoring the **Target** symbols and use **Source**.

To get the value of a **Stat** or **Attribute**, the target object of the query is first specified, followed by a dot (.) and the value type. Between brackets, the *id* of the stat or attribute is specified.

Stat Example

For example, to retrieve the attribute "mana" from the source object it's done using:

```
source.attr[mana]
```

- `base`: The base stat value of the object.
- `stat`: The final stat value of the object.
- `attr`: The attribute value of the object.

Circular Formulas

It is up to the game designer to avoid circular dependencies, and Game Creator will not warn about them. A circular dependency happens when a formula requires a value, which must be calculated using the first formula. This locks the process in an infinite loop.

Variables

Variables work very similarly to retrieving Stats and Attributes. The targeted object is first specified, followed by a dot (.) and the keyword *var*. And between brackets, the name of the variable.

Example

For example, if a numeric Local Variable attached to the targetted object with the id "hit-counter" should be accessed, the expression would be:

```
target.var[hit-counter]
```

Local Variables

For the moment, a Formula can only access **Local Variables** by name. In a future update, **List Variable** access will be supported.

Random

Most skill checks use some sort of random values. The **Formula** analyzer provides three symbols to generate a random value.

- `random[min, max]` : Returns a value between *min* and *max*, both included.

  ```
  Random[min, max]
  ```

  Using `random[1, 4]` returns a decimal value between these ranges.

- `dice[rolls, sides]` : For those old-school game designers, you can roll X amount of dices of Y sides and this symbol will return the sum of values.

  ```
  Dice[rolls, sides]
  ```

  Using `dice[2, 6]` returns the result of rolling 2 dices of 6 sides (the most common one).

- `chance[value]` : Returns 1 if a random value between 0 and 1 is lower or equal than the value specified.

  ```
  Chance[value]
  ```

  Using `chance[0.2]` has a 20% chance of returning a value of 1 and an 80% chance of returning 0.

Arithmetic

Number manipulation is also useful and commonly used. For example, to round numbers or choosing between two.

- `min[a, b]` : Returns the lowest value between two.
- `max[a, b]` : Returns the greatest value between two.
- `round[value]` : Returns the value rounded up or down to the closest integer.
- `floor[value]` : Returns the integer part of the value.
- `ceil[value]` : Returns the next integer of the input value.

Tables

Tables are mostly used for player progression, as they map a certain input value to another value. For more information about **Tables** see this link.

```
Table asset
```

It is required to provide the **Formula** with a **Table** asset.

Table symbols start with `table` followed by a dot (.) and the type of value to retrieve. The value is specified between brackets afterwards.

Level from Experience

For example, let's say we have a stat called `experience` and we want to calculate the character's level based on that. We can use a **Table** that transforms the accumulated experience points to a value that represents the level. In this case, the expression would be:

```
table.level[experience]
```

- `level[value]` : Returns the *level* at from the table based on the input cummulative value.
- `value[level]` : Returns the cummulative value necessary to reach the input level.
- `increment[level]` : Returns the amount left to reach the next level.
- `current[value]` : Returns the value gained at the current level.
- `next[value]` : Returns the value left to gain to reach the next level.
- `ratio[value]` : Returns a unit ratio that represents the progress made at the current level.

4.5 Tables

Commonly used for character progression, **Tables** are charts that map a range of values to an integer.

4.5.1 Concepts

Here are some concepts to better understand how **Tables** work.

- **Level**: An integer value that is calculated based on the cumulative value.
- **Cumulative Value**: This is the total amount of value (or experience) accumulated.
- **Value**: The difference between the current level's cumulative value and the total cumulative value.

Table concepts

4.5.2 Creating a Table

To create a **Table** asset, right click on the *Project panel* folder you want to create it and select Create Game Creator Stats Table.

Table

A **Table** asset has a visual chart and a configuration box at the bottom. The chart can be scrubbed to reveal the different cumulative values at each level.

Example

In the example above, at *Level 13*, the cumulative value is 1248 and it will require 208 more (for a total of 1455) to reach *Level 14*.

4.5.3 Types of Progressions

A character can progress linearly, exponentially, or at a custom rate. That's why Game Creator provides a range of different tables for the user to choose from.

Tables Progression

To change the type of progression, click onl the **Table** field and choose one from the dropdown menu:

- **Manual**: Each level requires a pre-defined amount of experience.
- **Constant**: Each level requires the same amount of value (or experience).
- **Linear**: Each level requires a value equal to the product of a constant and the current level.
- **Geometric**: Each level requires a value equal to the current level multiplied by a fixed coefficient rate.

Recommendation

We recommend using **Linear Progression** for most cases, as it's the one commonly used in games where the player progressively receives more experience. **Geometric Progression** is recommended for short games where power ramps up very quickly (like in MOBAS).

4.6 Stat Modifiers

We've seen so far that objects with a **Traits** component can change their **Stat** and **Attribute** values at runtime using **Formulas** and **Tables**. However, characters in games can also increment/decrement their stats when equipping weapons and other kinds of wearables.

This is where **Stat Modifiers** come into play: They increase or decrease a **Stat** value by a certain amount, and can be added and removed at any time.

4.6.1 Adding Stat Modifiers

To add a **Stat Modifier** to a **Traits** component, use the visual scripting Instruction **Add Stat Modifier**. This instruction allows to specify a target object, which must have a **Traits** component, a **Stat** to affect and a value.

This value can either be a percentage or a constant and can be displayed separately in the UI.

Stat Modifiers in UI

Percentage and Constants

You may have raised an eyebrow when **Stat Modifiers** can use constant and percentage values, as the result is different when applying a product after an addition or vice versa. The **Stats** module always applies percentage based modifiers first, and then adds any constant modifiers.

Add a Stat Modifier

4.6.2 Removing Stat Modifiers

Removing a **Stat Modifier** is as easy as adding one. All that needs to be done is to use the visual scripting instruction **Remove Stat Modifier** and input the same values as a previously added one.

Remove a Stat Modifier

4.7 Status Effects

Status Effects are temporal ailments that affect a character.

Most RPG games use the same **Status Effects**, such as *Poison*, which drains the character's health for a period of time. However, you can create your own and completely customize the afliction.

4.7.1 Creating a Status Effect

To create a **Status Effect** asset, right click on the *Project panel* folder you want to create it and select Create Game Creator Stats Status Effect.

Status Effects

A **Status Effect** has an ID which is used to uniquely identify it among all other afflictions. It is very important to keep this value unique across the whole project.

The **Type** field determines whether this effect is positive, negative or neutral for the targeted character. This is useful when using the instruction **Remove Status Effects**, where you can choose to remove only those that have a negative impact.

Max Stack determines how many of the same **Status Effect** can be active at a give time on a target.

By default, most **Status Effects** will have a stack of 1, and adding subsequent effects refresh the duration. However, it is entirely possible to stack multiple (for example) *Poison* aflictions, increasing their health drain.

The **Save** toggle determines whether the **Status Effect** persists after saving and loading back the game. Saving a **Status Effect** keeps track of the remaining time.

Has Duration allows the **Status Effect** to run for a certain amount of time (specified in the **Duration** field, in seconds).

If this field is unticked, the **Status Effect** will continue until it's manually removing, using the appropriate visual scripting instruction.

Status Effects UI section

The **UI** section allows the user to define any information displayable to the player, such as the name, a description of what the ailment does, its color and even an icon.

Status Effects Start End and While Active sections

Inside the **OnStart**, **On End** and **While Active** sections is where the logic of the **Status Effect** goes and it uses Game Creator's visual scripting tools.

- **On Start**: A list of instructions executed as soon as the **Status Effect** is added onto a target.
- **On End**: A list of instructions executed when the **Status Effect** stops taking effect on a target.
- **While Active**: A list of instructions that runs every frame, as long as the **Status Effect** is active.

Poison

For example, a **Poison** status effect could start spawning a particle effect onto the targeted character using the **On Start** instruction list. To damage the player, it would use the **While Active** instruction list and subtract a bit of the Target's health every few seconds.

4.7.2 Adding a Status Effect

To add a **Status Effect** onto a target you can use the visual scripting instruction **Add Status Effect**.

Add a Status Effect

All that needs to be done is to select the targeted character, which must have a **Traits** component, and specify the type of **Status Effect**.

4.8 User Interface

4.8.1 User Interface

The **Stats** module makes it really easy to build flexible user interfaces (UI) using Unity UI.

Stat User Interface

It comes with a few components that work fairly similar. You can attach each component to any UI game object and drag and drop any Text and Images to each of its fields.

- **Stat UI**
- **Attribute UI**
- **Formula UI**
- **Status Effects UI**

These components are all found under the *Add Component* submenu on any game object and navigating to Game Creator UI Stats. For example, this is the **Stat UI** component.

Stat UI example

The first two fields are required: **Target** is the game object with a **Traits** component and **Stat** is the asset to be referenced by this UI component.

All other fields are optional and will only be updated if a change is detected.

Stat UI

For example, dragging a **Text** component onto the *Value* field will change the contents to a numeric value that represents the selected **Stat** value.

4.8.2 Stat UI

The **Stat UI** component allows to display the runtime information about a specific target's **Stat**. To create one, click on a game object's *Add Component* button and navigate to Game Creator UI Stats Stat UI.

Stat UI

All fields are optional and all that needs to be done is to drag **Text** and **Image** components to the corresponding fields.

Stat UI

For example, to display the *Name* of a **Stat**, drag and drop the **Text** component onto the *Name* field and it will automagically update its content, even if the targeted game object changes.

4.8.3 Attribute UI

The **Attribute UI** component allows to display the runtime information about a specific target's **Attribute**. To create one, click on a game object's *Add Component* button and navigate to Game Creator UI Stats Attribute UI.

Attribute UI

All fields are optional and all that needs to be done is to drag **Text** and **Image** components to the corresponding fields.

Attribute UI

For example, to display the *Name* of an **Attribute**, drag and drop the **Text** component onto the *Name* field and it will automagically update its content, even if the targeted game object changes.

Transitions are a feature that allow the **Image** fill progress to animate and stall for a certain amount of time.

Attribute UI Transitions

Transitions

This is mostly used on health and mana bars, where getting hit makes the HP bar display a second bar below that decreases after a few seconds, in order for the player to get a sense of the amount of damage taken.

Ticking any of both options reveals two new options below.

- **Stall Duration**: Amount of seconds debounced between the value change and the start of the transition
- **Transition Duration**: Amount of seconds it takes to animate towards the targeted value.

4.8.4 Formula UI

The **Formula UI** component allows to display the result of an expression between two game objects with a **Traits** component. To create one, click on a game object's *Add Component* button and navigate to Game Creator UI Stats Formula UI.

Formula UI

All fields are optional and all that needs to be done is to drag **Text** and **Image** components to the corresponding fields.

Formula UI

For example, to display the resulting value of a **Formula** applied to the Player and another character, drag and drop the **Text** component onto the *Value* field and it will automagically update its content, even if any of the targeted game objects changes.

4.8.5 Status Effects UI

Status Effects have two components to display their information.

Status Effect List UI

- **Status Effect List UI**: Gathers information about a targeted game object and manages the concrete list of activet aflictions.
- **Status Effect UI**: Displays information about a particular afliction. It is spawned by the Status Effect List UI component.

Status Effect List UI

To create one, click on a game object's *Add Component* button and navigate to Game Creator UI Stats Status Effect List UI.

Status Effect List UI

The **Target** field should point at the game object with a **Traits** component.

Types allows to filter which status effects to display: Negative, Positive, Neutral, or any combination of them.

Container and **Prefab Status Effect** are the most important ones: For each afliction on the targeted character, the **Status Effect List UI** component will spawn (or reuse) an instance of a prefab. The spawn location is as a child of the **Container** rect transform.

Example

So if the Player has 3 ailments: *Poison*, *Paralyzed* and *Bleeding*, the **Status Effect List UI** component will spawn 3 instances of the prefab as a child of the **Container** transform.

Each spawned instance must have, at the root level, the component **Status Effect UI** component, which communicates with the **Status Effect List UI** which afliction to display.

Status Effect UI

To create one, click on a game object's *Add Component* button and navigate to Game Creator UI Stats Status Effect UI.

Status Effect UI

As can be seen, this component does not have a **Target** field. Instead, its the **Status Effect List UI** component that feeds it the target and concrete afliction.

All fields are optional and automatically update the values according to changes sent by the parent component.

4.9 Visual Scripting

4.9.1 Visual Scripting

The **Stats** module symbiotically works with **Game Creator** and the rest of its modules using its visual scripting tools.

- **Instructions**
- **Conditions**
- **Events**

Each scripting node allows other modules to use any **Stats** feature.

The **Stats** module also comes with a collection of custom **Properties**. Any interactive element can request the value of a **Stat**, **Attribute** and **Formula** using the value dropdown, as seen in the image below.

Properties

4.9.2 Conditions

Conditions

SUB CATEGORIES

- Stats

Stats

STATS

Conditions

- Check Formula
- Compare Attribute
- Compare Stat
- Has Status Effect

CHECK FORMULA

```
Stats » Check Formula
```

Description

Returns the comparison between the result of a Formula against another value

Parameters

Name	Description
Formula	The Formula used in the operation
Source	The game object that the Formula identifies as the Source
Target	The game object that the Formula identifies as the Target
Compare To	The value that the result of the Formula is compared to

Keywords

`Skill` `Throw` `Check` `Dice` `Lock` `Pick` `Charisma` `Speech`

COMPARE ATTRIBUTE

```
Stats » Compare Attribute
```

Description

Returns true if the Attribute comparison is successful

Parameters

Name	Description
Traits	The targeted game object with a Traits component
Attribute	The Attribute type value that is compared
Value	The type of value from the attribute to compare
Comparison	The comparison operation performed between both values
Compare To	The decimal value that is compared against

Keywords

`Health` `Mana` `Stamina` `Magic` `Life` `HP` `MP`

COMPARE STAT

```
Stats » Compare Stat
```

Description

Returns true if the Stat comparison is successful

Parameters

Name	Description
Traits	The targeted game object with a Traits component
Stat	The Stat type value that is compared
Comparison	The comparison operation performed between both values
Compare To	The decimal value that is compared against

Keywords

`Vitality` `Constitution` `Strength` `Dexterity` `Defense` `Armor` `Magic` `Wisdom` `Intelligence`

HAS STATUS EFFECT

```
Stats » Has Status Effect
```

Description

Returns true if the game object has a particular Status Effect active

Parameters

Name	Description
Target	The targeted game object with a Traits component
Status Effect	The type of Status Effect that is checked
Min Amount	The minimum amount of stacked and active Status Effects

Keywords

`Buff` `Debuff` `Enhance` `Ailment` `Blind` `Dark` `Burn` `Confuse` `Dizzy` `Stagger` `Fear` `Freeze` `Paralyze` `Shock` `Silence` `Sleep` `Silence` `Slow` `Toad` `Weak` `Strong` `Poison` `Haste` `Protect` `Reflect` `Regenerate` `Shell` `Armor` `Shield` `Berserk` `Focus` `Raise`

4.9.3 Events

Events

SUB CATEGORIES

- Stats

Stats

STATS

Events

- On Attribute Change
- On Stat Change
- On Status Effect Change

ON ATTRIBUTE CHANGE

```
Stats » On Attribute Change
```

Description

Executed when the value of a specific game object's Attribute is modified

Parameters

Name	Description
Target	The targeted game object with a Traits component
When	Determines if the event executes when the Attribute increases, decreases or both
Attribute	The Attribute from which the event detects its changes

Keywords

`Health` `HP` `Mana` `MP` `Stamina`

ON STAT CHANGE

```
Stats » On Stat Change
```

Description

Executed when the value of a specific game object's Stat is modified. Including due to Stat Modifiers

Parameters

Name	Description
Target	The targeted game object with a Traits component
When	Determines if the event executes when the Stat increases, decreases or both
Stat	The Stat from which the event detects its changes

Keywords

`Health` `HP` `Mana` `MP` `Stamina`

ON STATUS EFFECT CHANGE

```
Stats » On Status Effect Change
```

Description

Executed when a Status Effect is added or removed from a Traits component

Parameters

Name	Description
Target	The targeted game object with a Traits component
Status Effect	Determines if the event detects any Status Effect change or a specific one

Keywords

`Buff` `Debuff` `Enhance` `Ailment` `Blind` `Dark` `Burn` `Confuse` `Dizzy` `Stagger` `Fear` `Freeze` `Paralyze` `Shock` `Silence` `Sleep` `Silence` `Slow` `Toad` `Weak` `Strong` `Poison` `Haste` `Protect` `Reflect` `Regenerate` `Shell` `Armor` `Shield` `Berserk` `Focus` `Raise`

4.9.4 Instructions

Instructions

SUB CATEGORIES

- Stats

Stats

STATS

Sub Categories

- Ui

Instructions

- Add Stat Modifier
- Add Status Effect
- Change Attribute
- Change Stat
- Clear Status Effects Type
- Remove Stat Modifier
- Remove Status Effect

ADD STAT MODIFIER

```
Stats » Add Stat Modifier
```

Description

Adds a value Modifier to the selected Stat on a game object's Traits component

Parameters

Name	Description
Target	The targeted game object with a Traits component
Stat	The Stat that removes the Modifier
Type	If the Modifier changes the Stat by a constant value or by a percentage
Value	The constant or percentage-based value of the Modifier

Keywords

`Slot` `Increase` `Equip` `Fortify` `Vitality` `Constitution` `Strength` `Dexterity` `Defense` `Armor` `Magic` `Wisdom` `Intelligence`

ADD STATUS EFFECT

```
Stats » Add Status Effect
```

Description

Adds a Status Effect to the selected game object's Traits component

Parameters

Name	Description
Target	The targeted game object with a Traits component
Status Effect	The type of Status Effect that is added

Keywords

`Buff` `Debuff` `Enhance` `Ailment` `Blind` `Dark` `Burn` `Confuse` `Dizzy` `Stagger` `Fear` `Freeze` `Paralyze` `Shock` `Silence` `Sleep` `Silence` `Slow` `Toad` `Weak` `Strong` `Poison` `Haste` `Protect` `Reflect` `Regenerate` `Shell` `Armor` `Shield` `Berserk` `Focus` `Raise`

CHANGE ATTRIBUTE

```
Stats » Change Attribute
```

Description

Changes the current Attribute value of a game object's Traits component

Parameters

Name	Description
Target	The targeted game object with a Traits component
Attribute	The Attribute type that changes its value
Change	The value changed

Keywords

`Health` `HP` `Mana` `MP` `Stamina`

CHANGE STAT

```
Stats » Change Stat
```

Description

Changes the base Stat value of a game object's Traits component

Parameters

Name	Description
Target	The targeted game object with a Traits component
Stat	The Stat type that changes its value
Change	The value changed

Keywords

`Vitality` `Constitution` `Strength` `Dexterity` `Defense` `Armor` `Magic` `Wisdom` `Intelligence`

CLEAR STATUS EFFECTS TYPE

```
Stats » Clear Status Effects Type
```

Description

Clears any Status Effects based on their type from the selected game object's Traits component

Parameters

Name	Description
Target	The targeted game object with a Traits component
Types	The type of Status Effects that are cleared

Keywords

`Buff` `Debuff` `Enhance` `Ailment` `Blind` `Dark` `Burn` `Confuse` `Dizzy` `Stagger` `Fear` `Freeze` `Paralyze` `Shock` `Silence` `Sleep` `Silence` `Slow` `Toad` `Weak` `Strong` `Poison` `Haste` `Protect` `Reflect` `Regenerate` `Shell` `Armor` `Shield` `Berserk` `Focus` `Raise`

REMOVE STAT MODIFIER

```
Stats » Remove Stat Modifier
```

Description

Removes an equivalent Modifier from the selected Stat on a game object's Traits component.

Parameters

Name	Description
Target	The targeted game object with a Traits component
Stat	The Stat that receives the Modifier
Type	If the Modifier changes the Stat by a constant value or by a percentage
Value	The constant or percentage-based value of the Modifier

Keywords

`Slot` `Decrease` `Unequip` `Weaken` `Vitality` `Constitution` `Strength` `Dexterity` `Defense` `Armor` `Magic` `Wisdom` `Intelligence`

REMOVE STATUS EFFECT

```
Stats » Remove Status Effect
```

Description

Removes a Status Effect from the selected game object's Traits component

Parameters

Name	Description
Target	The targeted game object with a Traits component
Amount	Indicates how many Status Effects are removed at most
Status Effect	The type of Status Effect that is removed

Keywords

`Buff` `Debuff` `Enhance` `Ailment` `Blind` `Dark` `Burn` `Confuse` `Dizzy` `Stagger` `Fear` `Freeze` `Paralyze` `Shock` `Silence` `Sleep` `Silence` `Slow` `Toad` `Weak` `Strong` `Poison` `Haste` `Protect` `Reflect` `Regenerate` `Shell` `Armor` `Shield` `Berserk` `Focus` `Raise`

UI

Ui

Instructions

- Change Attributeui Attribute
- Change Attributeui Target
- Change Statui Stat
- Change Statui Target
- Change Status Effects List Ui Target

Change AttributeUI Attribute

```
Stats » UI » Change AttributeUI Attribute
```

Description

Changes the Attribute from a Attribute UI component

Parameters

Name	Description
Attribute UI	The game object with the Attribute UI component
Attribute	The new Attribute asset

Change AttributeUI Target

```
Stats » UI » Change AttributeUI Target
```

Description

Changes the targeted game object of an Attribute UI component

Parameters

Name	Description
Attribute UI	The game object with the Attribute UI component
Target	The new targeted game object with a Traits component

Change StatUI Stat

```
Stats » UI » Change StatUI Stat
```

Description

Changes the Stat asset from a Stat UI component

Parameters

Name	Description
Stat UI	The game object with the Stat UI component
Stat	The new Stat asset

Change StatUI Target

```
Stats » UI » Change StatUI Target
```

Description

Changes the targeted game object of an Stat UI component

Parameters

Name	Description
Stat UI	The game object with the Stat UI component
Target	The new targeted game object with a Traits component

Change Status Effects List UI Target

```
Stats » UI » Change Status Effects List UI Target
```

Description

Changes the targeted game object of an Status Effects List UI component

Parameters

Name	Description
Status Effects List UI	The game object with the Status Effects List UI component
Target	The new targeted game object with a Traits component

4.10 Releases

4.10.1 Releases

Game Creator 2

The **Stats 2** module is a **Game Creator 2** extension and will not work without it.

2.0.3 (Next Release)

Release Pending

ENHANCED

- Easier to understand examples

CHANGED

- Classes installer has no dependencies

2.0.2

Released November 22, 2021

NEW

- Instruction: Change AttributeUI Attribute
- Instruction: Change StatUI Stat

CHANGED

- UI instructions are now found under Stats/UI/
- Disallow multiple Traits component per object

FIXED

- Event: Attribute Change not running

2.0.1

Released November 19, 2021

- First release

5. Quests

5.1 Quests

WIP

This module is currently under developement

6. Behavior

6.1 Behavior

WIP

This module is currently under developement

7. Perception

7.1 Perception

WIP

This module is currently under developement

8. Shooter

8.1 Shooter

WIP

This module is currently under developement

9. Melee

9.1 Melee

WIP

This module is currently under developement

10. Traversal

10.1 Traversal

WIP

This module is currently under developement