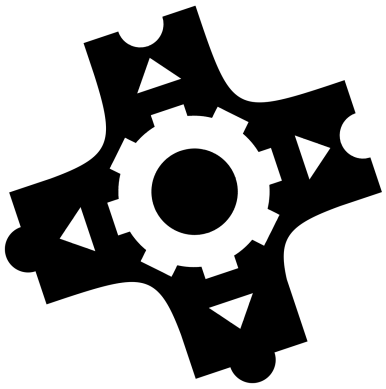


GAMEFLOW

User Manual

v0.8 Beta

evasiongames

Overview

GameFlow is a plugin or extension for the Unity engine to simplify and accelerate the development of video games to greatly reduce the barrier for many game designers and artists supposed scripting or programming logic and special effects.

GameFlow is perfectly integrated in the Unity editor adding a support event-based visual programming blocks of shares that allows designers / as games with basic programming skills to build perfectly functional programs in a fraction of the time it would take to do so through languages scripting as C # or Javascript.

GameFlow also includes a set of tools to facilitate the work of mounting levels of play, such as a visual editor trajectories as well as a full repertoire of prefabs (parts of prefabricated game) configured and ready to use.

GameFlow is designed for reuse and has an own extensibility API for advanced users who want to create their own blocks of stock or bespoke events.

Visual programming

Call or visual programming “visual scripting” to one in which to create a program is not necessary to write lines of code in a text editor, but only manipulate visual elements. For GameFlow, these visual elements are blocks of different types (actions, conditions, etc.) that can be associated to different game objects (hereinafter *GameObjects)* there at the scene.

Features

Programming

GameFlow lets you design the logic or the effects of a game through programs (sequences of actions) we can associate the *GameObjects,* all without leaving the editor of Unity itself.

Actions

GameFlow comes standard with a repertoire of 150 shares to cover much of the basic necessities required in developing a game. A complete list is available in the Actions section.

Variables

All actions and tools GameFlow accept not only specific values for their properties but also variables, which allows to build flexible and versatile programs. The variables, of course, are also defined from the editor.

Events

GameFlow facilitates based scheduling (see an explanation in Chapter Events) through a variety of special programs *(Event Programs)* that run automatically when certain events occur.

Lists

GameFlow supports dynamic lists of items in the editor itself. These lists can be used for many purposes during the game, but are especially useful to apply the same action to multiple objects in one step.

Trajectories

GameFlow incorporates an editor of both linear paths as based on curves that can be combined with some action to get the game objects follow them over time and also for other purposes such as automated scenario generation with curves (one road, example).

Forces

GameFlow allows defining forces (vectors with a direction and a magnitude represented as arrows) can be edited visually in the editor and can be applied on demand over the objects you want. It's an easy way to communicate with the physics engine Unity to add motion effects to our game.

Timers

Timers *(Timers)* are components that allow the implementation of programs at regular intervals or when a timeout exhausted. Can also be used to construct type markers time countdown timer or type.

Pools

The pools are storehouses of objects of the same type that improve the performance of a play in those situations where it is necessary to continuously display a multitude of objects on the screen. GameFlow Pools offers support integrated and easy to use.

GameFabs

GameFlow incorporates a repertoire of object templates *(Prefabs* in the jargon of Unity) we call *Gamefabs* that are designed as parts ready to use and can considerably accelerate the installation of a video game. In this directory we find from keyboard drivers and mouse up generators objects and ready to use markers.

Persistence

With just one click, GameFlow deals with data persistence of your game so that it is easy to program games that remember the state of the game.

Parameterization

GameFlow allows the construction of parameterized *Prefabs* from the editor without having to write code. This way you can create lists parts for reuse in other projects.

Automation

Perhaps the feature “hidden” GameFlow most powerful is that it allows the execution of programs not only at runtime but also editing time. That is, you can easily build programs that generate or manipulate scene elements from the editor, thus facilitating the construction of game levels.

First Steps

Installation

If GameFlow was acquired in the Unity Asset Store your package (.unityPackage file) already installed on your system and ready to use in our projects. The only required step is to import the package within our project in one of two ways:

1. If you create a new project, check the appropriate check the package *GameFlow.unityPackage* window creating new project to display directly imported.
2. If you wish to incorporate GameFlow to an existing project, find the menu option *Assets*> *Import Package*> *GameFlow* and click the "Import" button in the Import window that appears below.

If instead GameFlow was not purchased in the Unity Asset Store installation package is also very simple, it just open or create a project where we want to import GameFlow and then double click on the file *GameFlow.unityPackage* to immediately begin your Import.

If you want you can also copy the file to *GameFlow.unityPackage Standard Packages* folder should appear in the installation path of Unity for thus it is available as a package purchased in the Unity Asset Store.

After import and regardless of how we have done, we should note that in the *Project* window Unity GameFlow a new folder containing a folder of documents, some .DLL files and a *Readme* file contains release notes appear.

Update

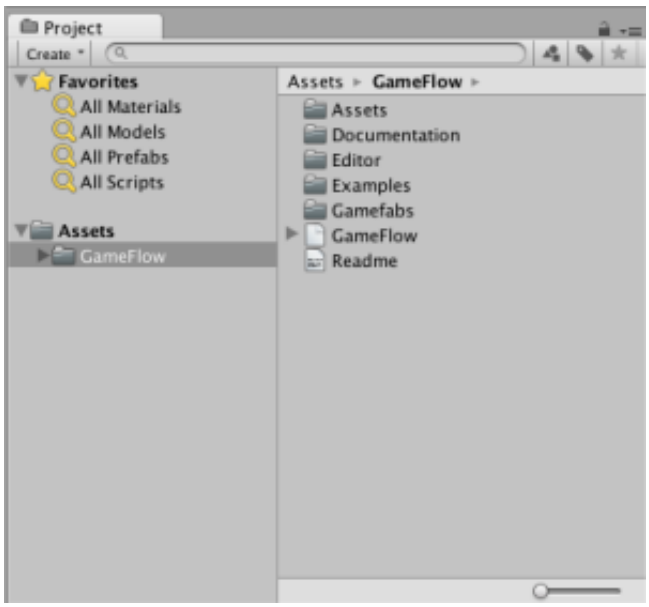
The update process is exactly like the installation, as could be considered a re-import the package. It should, however, always read the release notes for the new version prior to updating to ensure that it does not

include changes that may affect the proper functioning of our game or application.

Help

Documentation

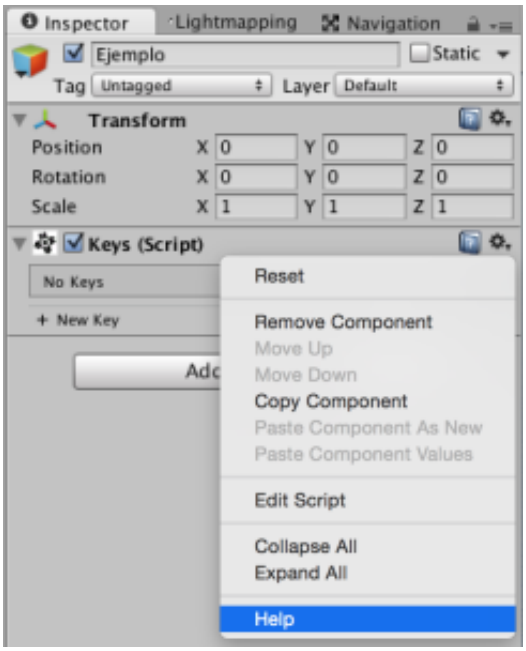
The documentation containing the user manual and reference programming API can be found in the relevant section of the website GameFlow.This same documentation is also available for consultation offline under the *GameFlow / Documentation* of Unity project where we GameFlow importing the package folder.



The manual is in PDF format while the reference documentation is in HTML format, currently only in English and Spanish.

Contextual Help

As to aid in the Unity editor, it is always available as a last option in the context menu of each element. The context menu is displayed by clicking any of the mouse buttons on the gear icon appears in the upper right corner of any component or block GameFlow in the Inspector window.



Quick help summarized in the windows for selecting actions and conditions, including an icon that gives us access to the selected item page manual is also displayed.

Tutorials

In the section of tutorials on the GameFlow web, new video tutorials that explain through short examples developing specific techniques step by step that may be useful when developing video games are regularly published.

Forum

Users who need more help can be found on the official forum GameFlow, a site aimed at sharing experiences, questions and knowledge with the rest of the community.

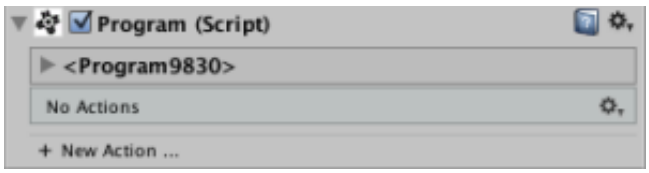
Basic Concepts

Basic Components

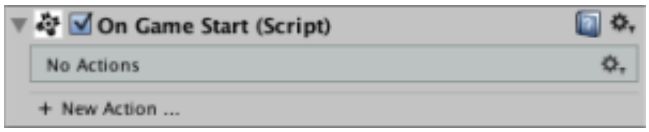
Containers

Premium components are displayed in the Inspector window and components of Unity, whose basic feature is that they contain other minor components called blocks. In GameFlow the following types of containers are distinguished:

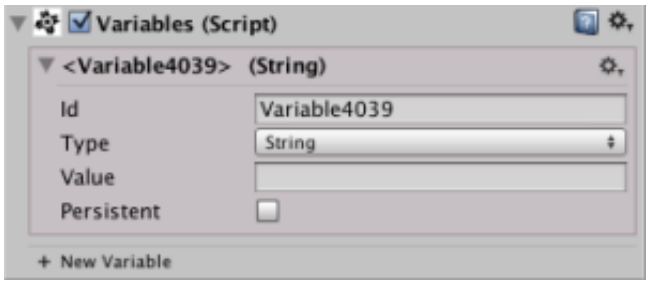
- **Program:** A sequence of actions to be inactive (ie, not running) until his execution was ordered explicitly.



- **Program event (On ...):** A sequence of actions that start automatically when an event such as recognized by the program detected.



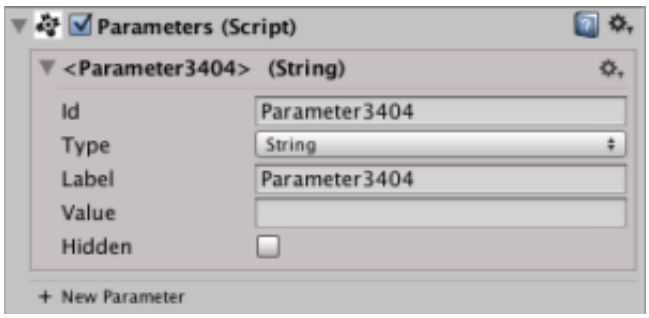
- **Variables:** A list showing all variables defined within the object.



- **Keys:** A list showing all the keys defined within that object.

- **Forces:** Displays all forces defined within the object.

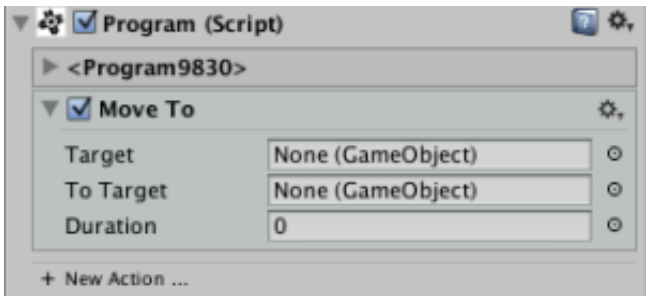
- **Parameters:** Shows all parameters defined in the object. The parameters are a special type of variables as discussed in the chapter.



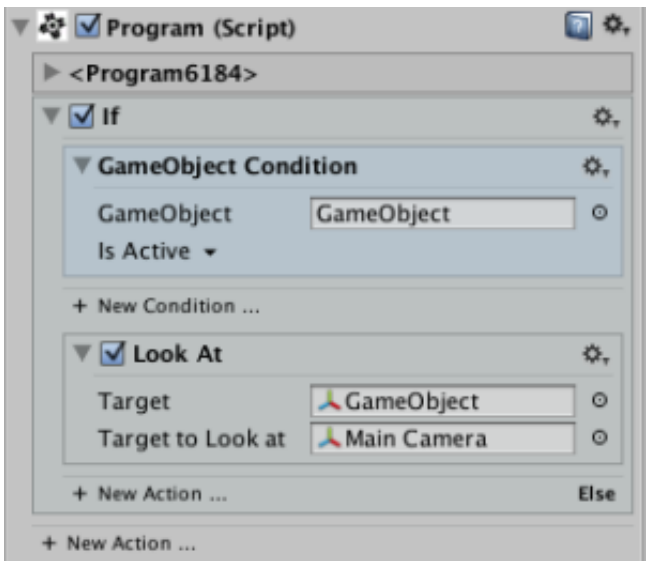
Blocks

These are elements that can not live independently outside of a container and playing a particular role in the functionality of the container. In GameFlow find the following types:

- **Action:** A program element that makes (if enabled) a very specific task based on parameters or properties. Most actions are executed instantaneously, but others on the contrary will run during the time interval that you specify.

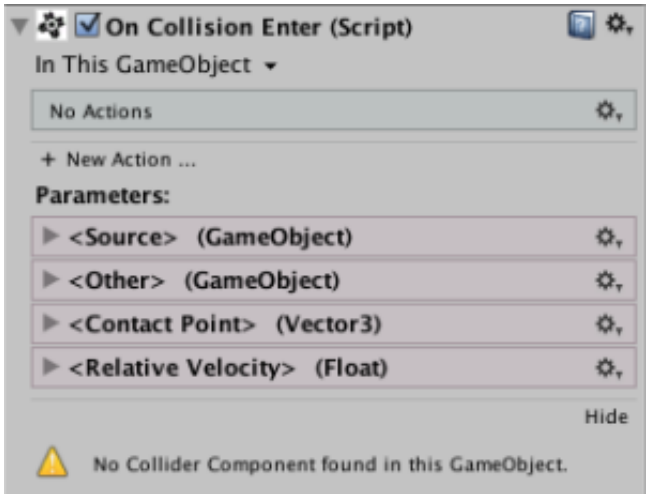


- **Condition:** An element of the program that performs a specific assessment to return a “True” or “False” result that can be used for conditional actions as *If* or *While* controlling the flow of a program.



- **Variable:** A dynamic storage unit that can store a data of a certain type. The great power of the variables is to be used as the value for the properties of virtually any element of GameFlow: actions, conditions, tools or even for other variables.
- **Built-in Variable:** A special type of variable that can not be edited because its type and value are already predefined.Usually serve to consult system information like time and date, resolution, platform, etc.

- **Parameter:** A special variable thought that serves both to display the parameters of event programs (pictured right) to allow the construction of parameterized Prefabs.



- **Key:** A key definition or way to tell GameFlow we want to monitor the status of that particular key.
- **Force:** A definition of a force that can be displayed in the editor.

Tools

They are basic independent components that have their own logic and its own form of visual editing, but are integrated with GameFlow to withstand variables and being manipulated by a series of actions that work specifically with them. We can access them through the browser and selecting components GameFlow> Tools. In GameFlow the following tools are distinguished.

- **List:** A component that stores a sequential set of values with the same base type. Like variables, GameFlow lists are dynamic and can be manipulated to add, insert, replace, or delete items in both runtime and editing time.
- **Timer:** A component that sends timing events at regular time intervals which, in combination with a program event as *Expire On Timer*allows execution of actions from time to time either by exhausting a timeout.
- **Path:** A component that defines a path in the space of the scene

from a series of points, and that can be manipulated visually and be covered by the objects of our game by *Follow Path* action.

- **Pool:** A component that defines a collection of pre-instantiated objects that can be used at runtime. The use of pools improves performance in situations where it is necessary to continuously display many objects on screen, an example would be a game ships with many screen shots.
- **Area:** A component that lets you define a cubic area in the scene and while activated and deployed in the Inspector will be drawn at all times. It is used to define areas of the game without having to resort to using Box Colliders.
- **Event Controller:** A component that lets you control what events are detected and which are ignored by the object.
- **Note:** A simple component that allows you to add a text note to the object, which can be useful, for example, to remember why we set certain parameters in a particular way at the time.

Accessories

GameFabs

The *GameFabs* (contraction Game Prefabs) are *GameObjects* or hierarchies configurable *GameObjects* that offer components and / or logical user often used in video games and whose incorporation into a juevo is trivial, so can dramatically accelerate mounting prototype games. GameFabs list is available on the GameFabs chapter.

Utilities

They are integrated tools that do not qualify components and typically deal with conduct special operations that can go beyond the issue. An example of use would be integrated GameFlow incorporated into a future version debugger.

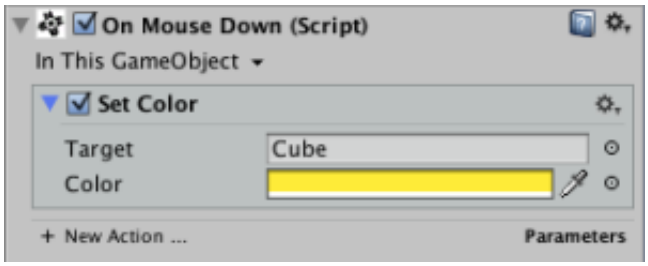
Templates

Readymade Templates are incorporating a system configuration guide and a higher level of *GameFabs,* as it is responsible for defining the configuration and interrelation of these. The template support will be incorporated in future versions of GameFlow.

Programs

A program is a component that contains an ordered list of actions that the GameFlow engine will run sequentially allowing both to give life to our playthings and controlling the flow of the game, hence the name of the tool.

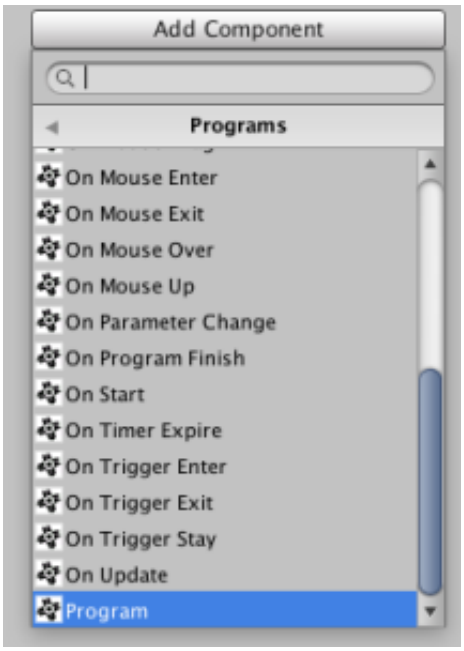
A program may, for example, responsible for controlling the movement of our main character, while another series of programs can deal with managing the enemies, controlling the game markers and switch the music to reach certain parts of the stage. Basically programs give life to a game.



Creating programs

As every component, a program begins to exist when added an object of particular game. This can be done using the *Add Component* button at the bottom of the list of components in the Inspector window, looking

below the *GameFlow> Programs* menu and then selecting one of the types of programs available.



As discussed below, considering the way a program runs in GameFlow two basic types of programs are distinguished:

- **Idle Program:** A program that remains inactive until the execution is ordered by either an *Execute Program* action or the Execute context menu option.
- **Event Program:** A program that, if enabled, starts running automatically recognized at an event type in the range of listening detected.

Program is the only program inactive, with all other programs of events. Besides all programs event begins with the word “On” so it is easy to distinguish them.

Adding actions

The actions are the building blocks from which programs are built. They can be considered small, specialized components that perform very specific tasks based on properties that you specify within their own area of editing in the Inspector window.

To add actions to a program we have added we can do it using the *Add Component* button Inspector and looking below the *GameFlow> Actions*menu, but we'll have more control if we use the + button *New Action* that appears at the bottom of the program area in the Inspector.

Since this button appears at the bottom of each container of shares, whether the program itself that is an action can contain other actions as *While,*use allow us to decide where exactly the program we want to add the action, which will be more comfortable to walk then rearranging the shares.

Also, use this button will allow us to find shares in an easier manner as in this case instead of using the menu of standard components of Unity, a special window for selecting actions prepared especially for this task will open.

This new window will show us in a unique list all available actions, both which brings the tool set as actions created as user via API GameFlow, along with fast integrated help that will allow us to get a better idea the purpose of each action.

To choose an action just have to move with the up and down arrows and press *Enter,* or double-click directly on the action you wish to incorporate into the program.

The window also has a search filter that will allow us to quickly filter actions so that only those remaining in their name entered text. In the future the filtering mechanism of this window will be enhanced to also

incorporate search by tags, so that the user is more likely to find a suitable for their purposes action.

Order of execution of actions

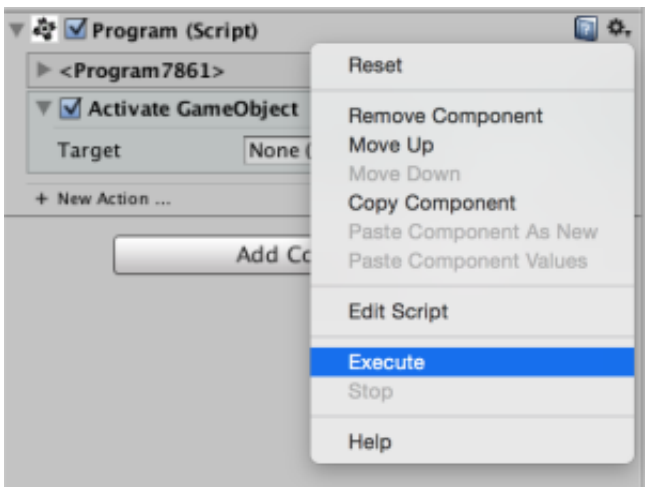
As indicated at the beginning of the chapter, the actions in a program are executed in sequential order, ie start with the first action program and continues its execution until, at which point it begins with the execution of the second action from the list, and so on until the end of the program.

As discussed in the chapter on flow control, this is the default order of execution for a linear program, but by using special actions such as *Group*or *If* you can vary this order of execution so that it is not entirely sequential and better suited to different use cases.

Execution Contexts

Most programs only event will be executed at runtime (ie when we have given the *Play* button) and not during editing time (ie while we are working on the editor of Unity) because the Most events are not generated until the game is not running.

However, it is important to note that GameFlow allows any *Program* (inactive program) is also implemented in time editing using the *Execute* option to the context menu, which can be very useful both for fast performance tests without booting the stake to automate editing tasks programmatically.



Actions

The actions are the building blocks from which programs are built. They can be considered small, specialized components that perform very specific tasks based on properties that you specify within their own area of editing in the Inspector window.

Most actions are executed in a timely and instantly when their turn comes within their program, but other actions, however, are designed to run during a time interval, reason why typically these actions include a *Duration* property.

Besides this, another difference of action on a typical Unity script *(MonoBehaviour)* is that actions are designed to be applied on a target that does not necessarily have to match the GameObject that contains the program it belongs to the action.

GameFlow actions are represented as collapsible blocks having its own interface as a subcomponent. If action is collapsed, it only shows its title bar. If the action is expanded or unfolded then it will also display all its properties. The color of these blocks is intentionally slightly different to the default background color of the Inspector window.

There are actions that can in turn contain other actions, such as *For* or *Repeat* actions that allow build repeat loops. In these cases, the actions contained are displayed within the area of the containing action with a level of indentation.

Editing operations

Most operations that allow editing time shares are available in your context (gear icon on the right side of the title area of action) menu.

Collapse / Expand

An action can be either expanded or collapsed by clicking on the arrow in the upper left corner of its title, or by clicking anywhere in the title area

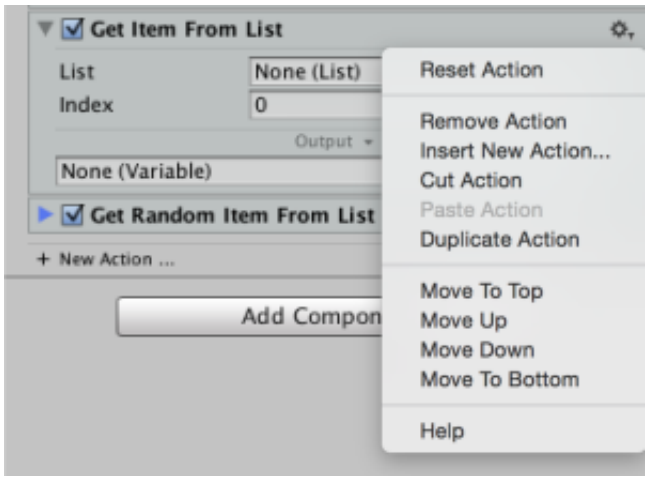
of action.

Activation / Deactivation

To enable or disable an action simply check or uncheck the checkbox that precedes its title. Enabling or disabling editing does not affect the properties of the action, only its implementation. Off action will not be executed by the program, which will continue execution at the next triggered action.

Rearrangement

An action can be moved to another place in the program either through the actions of movement from its context menu *(Move to Top, Move Up, Move Down, Move to Bottom)* or via a drag and drop *(drag and drop)* initiated with a click on its title bar.



An action may also be transferred to another program by *drag and drop* long as both programs are contained within the same *GameObject.*

Finally you can also move an action program to program actions using *Cut* and *Paste Action Action* context menu, but again with the limitation that both programs, both the source of action and target are *GameObject* contained therein.

Note: there is currently no action *Action Copy,* but its implementation is planned in future releases.

Reset

An action can be restarted to take their default values using the *Reset* option from its context menu.

Removal

An action can be eliminated by *Action Remove* option from its context menu.

Insertion

You can insert a new action before another using the *Insert New Action* option from the context menu that action.

Duplication

You can duplicate an action using the *Duplicate Action* context menu option.

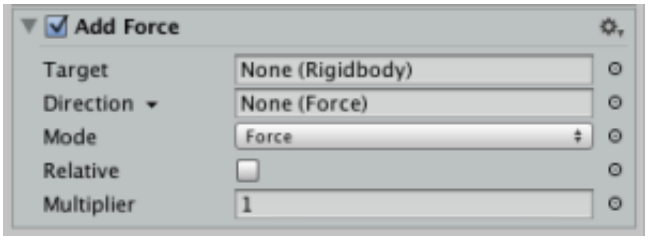
New action

In those actions that have the ability to contain other actions such as *If* by + *New Action* button appears at the bottom of the area of action when clicked will display the window for selecting actions to add actions.

Editing Properties

Editing properties of an action is performed as in any other part of Unity, with some slight differences:

- Some properties may have additional options as context menus associated with your tags, which are indicated in the interface with a small arrow pointing down to the right of the title. An example is found in the *Direction* property of the *Add* Action *Force,* which lets you specify the type of input value by this label incorporated into the context menu.



- Most properties are of any type, incorporated on the right side a small circle through which it is possible to associate, as discussed in the next chapter, a variable.
- Some actions (usually those whose names begin with the word *Get)* incorporate a special field to specify an output variable, and this field can be drawn as another variable for ease when composing programs.

Help

You can consult the help of an action under the Help option from the context menu.

Implementation of measures

Although the majority of things on GameFlow are designed to run at runtime, only some of these also work in editing time. When an action may not run on time editing GameFlow indicate this by sending a warning message to the console.

Moreover, there are also actions that perform only effective when executed on time editing and will be ignored at runtime not make sense in this context task.

Available Actions

The following is a list of actions currently incorporated in GameFlow organized by functionality.

Activation

- Activate GameObject
- Activate GameObjects In List
- Deactivate GameObject
- Deactivate GameObjects In List
- Disable Behaviour
- Disable Behaviours In List
- Disable Collider
- Disable Program
- Enable Behaviour
- Enable Behaviours In List
- Enable Collider
- Enable Program

Variables

- Decrement Variable Value
- Divide Variable Value
- Increment Variable Value
- Variable Limit Value
- Multiply Variable Value
- Set Variable Value
- Toggle Variable Value

Lists

- Add Item To List
- Clear List
- Get Item From List
- Insert Item In List

- Remove Item
- From List
- Set Item in List

Flow Control

- Break
- For
- Group
- If
- Loop
- Repeat
- Repeat Until
- Restart Program
- While

Execution

- Execute Program
- Start Program

Publisher

- Progress Close Window
- Setup Progress
- Show Progress Window
- Progress Update

Strings

- Concatenate Strings
- Get String Length

- Get Substring
- Replace String In

Documentation

- Comment

Creating Objects

- Clone
- Create Empty GameObject
- Instantiate

Destruction of objects

- Destroy

State of play

- Game Over
- Start Game
- Pause Game
- Resume Game
- Toggle Pause

Trajectories

- Follow Path
- Get Path Property
- Set Path Property

Motion

- Follow
- Interpolate

- Look At
- Move To
- Rotate
- Rotate To

Animation

- Play Animation
- Set Animator State

Transform

- Get Position
- Get Rotation
- Get Transform
- Get Transform Property
- Set Position
- Set Position From Screen Point
- Set Rotation
- Set Transform Property
- Get World Point From Screen Point

Audio

- Get Audio Property
- Set Audio Property
- Play Music
- Play Sound
- Play Sound At Source
- Stop Music
- Stop Sound At Source

- Toggle Audio Mute

GUI

- Get GUIText Property
- Get GUITexture Property
- Set GUIText Property
- Set GUITexture Property

Timers

- Wait For Timer
- Restart Timer
- Resume Timer
- Stop Timer

Cameras

- Get Camera Property
- Set Camera Property

Pools

- Get Object From Pool
- Get Pool Capacity
- Reset Pool

Random Values

- Get Random Color
- Get Random Item From List
- Get Random Number
- Get Random Point In Collider

- Get Random Point In Collider List
- Get Random Vector
- Set Random Seed

Restrictions

- Confine
- Set Distance
- Get Distance

Persistence

- Save Data

Mouse

- Hide Mouse Cursor
- Show Mouse Cursor

Scenes

- Load Scene
- Get Scene Property

Depuration

- Clear Console
- Log Message
- Pause Editor
- Hello World

Navigation

- Set NavMesh Agent Destination

Physics

- Add Force
- Get Rigidbody Property
- Get Velocity
- Velocity Limit
- Sleep
- Sleep List
- Wake Up
- Wake Up List
- Set Rigidbody Property
- Set Velocity

Time

- Set Time Scale
- Wait

GameObjects

- Get GameObject Property
- Set GameObject Property
- Set Parent

Vectors

- Get Vector Component
- Set Vector Component

Visual Effects

- Set Color

Application

- Set Application Property
- Exit Game

Variables

A variable is a component which behaves as a dynamic storage unit. This means that can contain a certain type of data (text, a number, a color, a reference to an object, etc.) and also be used in place of such a value where GameFlow supports the use of variables .

So if for example we have an action as *Set Color,* used to change the color to an object, and requires as one of its properties color we want to change the subject, we would have two ways to indicate what should be said color:

1. Assign the property a particular color, eg the red-directly through the color picker, or
2. Assigning a *Variable* property type *Color* indicating that we want to use as a color value of the variable (which will also be a color, eg green).

What happens when you run the game is that the action *Set Color* to be executed by a program will evaluate in turn the value of each of the variables which are assigned to determine its present value. In other words, and returning to our example, what will happen is that the action variable ask: “What color do you have at this time” and the variable will respond with a value of “green”, which will be the color that action eventually use for its purpose.

Since it is possible to change the value of variables as many times as needed throughout the life of the program, which offers integrated variables GameFlow support is the ability to build programs or use tools with input values dynamic that may vary depending on the state of the game or as other parameters.

For its dynamic nature, in a game we often find the need to work with variables. Some examples that suggest use would be: create a scoring track, control the damage state of our character, keep track of the number of collected objects, etc.

Create variables

The variables live like any another component within a *GameObject,* thus to create it is only necessary to add a component Variable type:

- Using the *Add Component* button in the Inspector window, or
- Using the menu *Component*> *GameFlow*> *Data*> *Variable.*

In either case if you are creating the first variable in the Variables GameObject one component (plural) that will act as a container for all blocks of variable rate we add the object is created.

From there we will have available a third way to add a new variable using the + *New Variable* button in the lower area of the component in the Inspector window.

We will also have several additional options in the context menu of this component Variables such as the *Reset* option or options *Collapse All* | *Expand All* that respectively allow us to collapse or expand all variables of the object in one step.

Definition of variables

To define a variable simply give it an appropriate identifier to the purpose we want to give the variable, indicate the type of data you want to store in it and give it an initial value. Optionally, we can also indicate whether we want the variable is persistent or not.

Type

The types supported by GameFlow variables are:

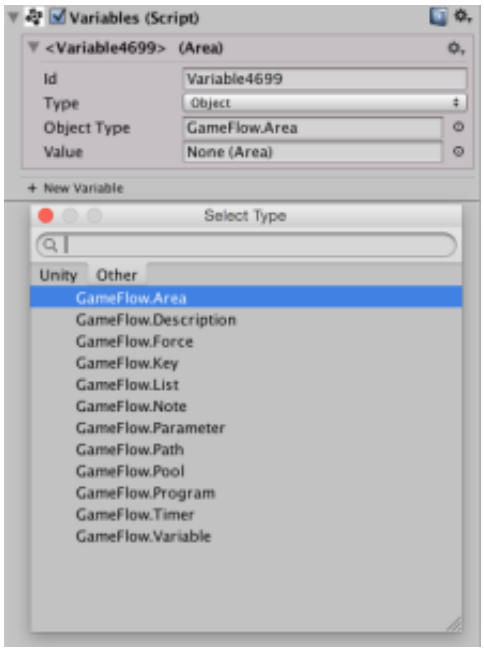
- **String:** Text.

- **Integer:** Integer Numbers.
- **Float:** Decimal numbers.
- **Boolean:** True or False.
- **Vector2:** 2-dimensional vector (X, Y).
- **Vector3:** 3-dimensional vector (X, Y, Z).
- **Rect:** Rectangle (X, Y, Width, Height).
- **Color:** Color.
- **Object:** A reference to an object.
- **Enum:** An enumeration value.
- **Toggle:** Marked or unmarked (equivalent to Boolean).
- **Tag**: GameObject Tag.
- **Layer**: GameObject Layer.

Type Specification

The *Object* and *Enum* types require additional specification of the particular object type or enumeration containing the variable that can be done by a special selection window types organized into two tabs:

- *Unity* **Tab***:* contains only the types of Unity usable in GameFlow.
- *Other* **Tab:** contains only GameFlow types and types that the user has defined through their own scripts.



Conversions type and value

We call *Type Casting* to the automatic modification of the type of a variable Variable GameFlow performed when this is modified by some action that returns values that are of a different type to the variable had at that time.

This means that in GameFlow variables are dynamic type and automatically adapt when used to store a data type different from that initially had.This has its advantages, but also disadvantages should therefore be taken into account.

The other conversion operation is automatically performed GameFlow to try to convert the current value to the new type, when all you modify the type of the variable, which is not strange perform at authoring time.

Thus, if in a variable of type *String* have the alphanumeric value “3” and suddenly we decided to change the type of that variable to *Integer,* GameFlow what he will do is try to make the best possible conversion of user values. In this example the conversion is possible and the value shall be updated, but if the conversion is not possible the value of the variable would become the default for the new type chosen.

Editing operations

Editing operations are essentially the same for the actions listed in the previous chapter, except that the Cut and Paste operations are not available.

Assignment by drag & drop

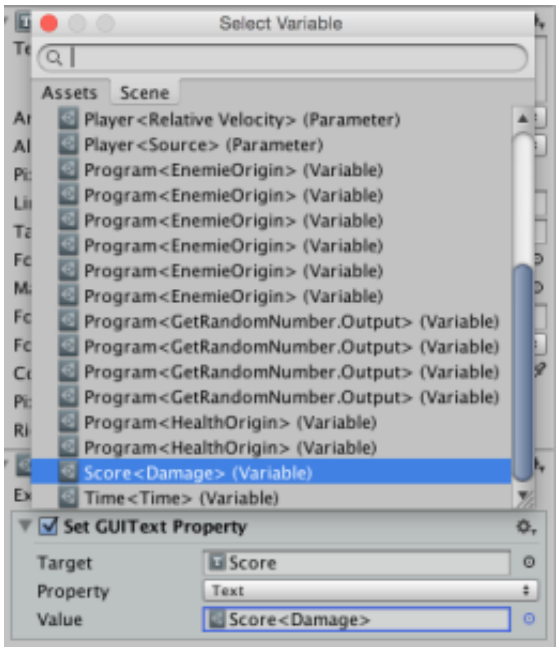
Perhaps the most important operation that we should know about the variables we can assign to virtually any property of any of the actions, conditions or tools including GameFlow.

To do this, simply click on the title block variable we want to drag up the property you want to assign the variable and once observe that the field of property is drawn with another fund, release the mouse button.

Assignment with object selector

The other way to assign a variable to a property is to click on the small circle that appears to the right of the field value of the property to show us a window GameFlow selection of special objects, which resembles the standard Unity but that is not so.

The first difference you observe regarding the standard window is that the selector GameFlow not left as the standard in the level of *GameObjects*but is able to reach the level of components, why can see each and every one of the variables that we created in the project and in the scene, each conveniently labeled with the identifier that would have put him.



The second difference is that GameFlow observe adds an extra tab that will allow us to see only the variables defined within the currently selected *GameObject* or if this belongs to a prefab, see only the variables defined within the hierarchy of the prefab. This is especially convenient when you're building prefabs, it helps us to find the variables that we seek and make sure we never choose an external variable to prefab faster.

It should also be noted that when the property we want to change is of type object, the selector will allow us to choose both objects of that type (whether they are components like *GameObjects)* as variables, so the work mode is identical to that discussed and if we want is to directly assign object references to the traditional way of Unity it can also be done without any problem.

Built-in Variables

Built-in Variables are a special kind of variables whose value can not be determined by the user but is already predefined (are variables read-only) and allow us to consult dynamic system settings, such as the current date or resolution screen, in a simple manner.

All the predefined variables can be found and added to the project as part of prefab *Built-in* of the *GameFlow / Assets* folder of your project, so you can find them at any time in *Assets* tab of the window object selection.

Then the integrated variables available are as follows:

- **Built-in <Day>:** Day of the month of the current date.
- **Built-in <Delta Time>:** Time elapsed since the last frame.
- **Built-in <Device Name>:** Name of device runtime.
- **Built-in <Hour>:** Current hour in 24h format.
- **Built-in <Minute>:** Current minute.
- **Built-in <Month>:** Number of month of the current date.
- **Built-in <Mouse Position>:** Current position of the mouse pointer.
- **Built-in <Native Resolution>:** Native resolution of the display.
- **Built-in <Screen Size>:** Screen Resolution / current window.
- **Built-in <Second>:** Current second.
- **Built-in <Year>:** Year of the current date in long format.

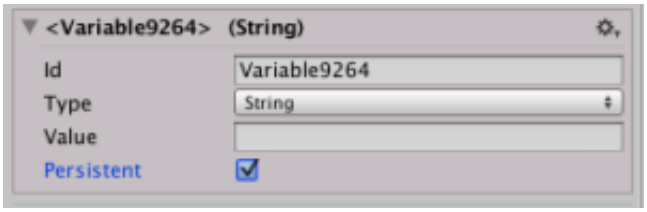
Persistence

When we speak of “persistence” talk about the ability of certain data is not deleted when you leave our game but remain stored in the nonvolatile memory device on which we run our game so we can restore them the next time you boot the game .

This is very useful in video games because it allows us to save valuable information on the progress of the player or the last state in which the game was so that we can implement in a relatively simple way games that allow “continue” a game in the last point being made or without going so far as to simply remember what was our best score.

GameFlow offers support transparent to the user and integrated variable persistence, but for technical reasons can only be used on variables that are NOT of type *Object.* All we have to do to get the value

of a variable “survive” at the end of the game is to check the box on *Persistent* interface in the Inspector window.



From that moment, GameFlow will ensure that the value of that variable is persistent, which we can verify even while we are working in the Editor, as each time you press *Play* the variable will be the last value that had at the end of the previous run.

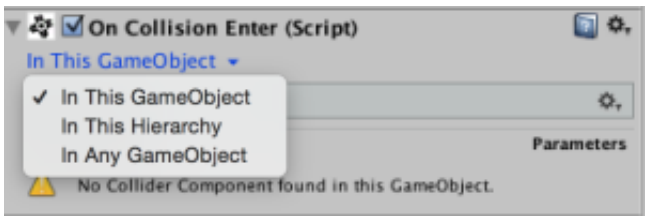
Events

As we saw in earlier chapters, event programs are programs that start running automatically enabled when being detected in its range of listening an event of a given type.

In this chapter we will see in detail all aspects of event programs to understand how to make the best of it to support event-driven programming GameFlow offers.

Listening range

The range listener will find it at the top of some programs of events such as a menu of options and functions as a filter program can be used to address only certain events based on their origin.



The range of listening will usually three possible values:

- *In This GameObject:* when we just want to attend events that have been detected by components of the same GameObject in which is included the program. This is the default and provides better performance.
- *In This Hierarchy:* when we attend events that come also both the parent object of the GameObject in which is included the program and any of its GameObjects children.
- *In Any GameObject:* when not want to do any filtering and want to attend the event whatever the source.

In most cases, the range of listening can be left at the default because it is good practice to limit the scope of a program to a minimum. Only where we need to make our program capable of acting on elements of the whole scene should expand the range of listening.

For instance, if we make a program that changes the color of your GameObject by clicking with the mouse on it will suffice minimum range, but if we seek to make a program to change the color of any object in the scene on where we click, we need necessarily extend the range to maximum.

Conditions for implementation

For an event program can begin execution is automatically required:

1. The program itself is activated or enabled.
2. That the event has occurred in the range specified in the program listening event.

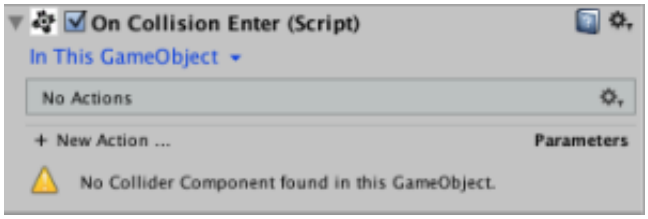
Except in some cases, such as On Start, for a running event program is not essential that the GameObject in which it is included is also active, especially if the listening range is set to maximum.

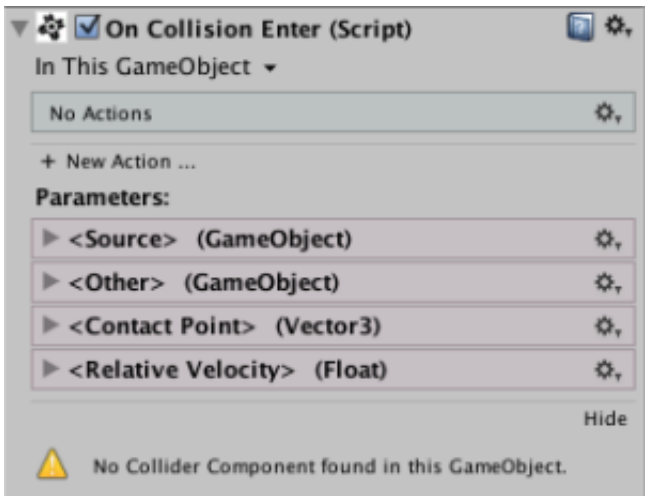
Event parameters

Most event programs offer a list of parameters whose value is completed with certain specific information about the event occurred just before the program starts running.

The event parameters significantly expand the possibilities of a program, since we can now make the program knows, for example, the GameObject that originated the event and go to one of the program's actions that parameter. It would, in other words, as allowing the program to act on an unspecified GameObject, at some point, when the event occurs, will be implemented to be one of the GameObjects of our game.

When the event program supports parameters, it displays a small label on the bottom right corner of your area in the Inspector with the Parameters text in bold. Clicking on the label, just a list of parameters under the action list with a format similar to variables blocks are displayed, because the parameters can be considered read-only variables.





Like variables, the event parameters can be dragged to the fields of program actions that support variables, but unlike variables, parameters can only be used in the program of event itself and does not allow drag them to other programs (even the same type) nor select them in the Object Selector, then are not reflected in it.

Supported Events

The following is a list of event types recognized and supported by GameFlow along with a brief description of the event programs dealing manage them.

Initialization events

They are the events related to the game initialization processes. This type of event is managed by the following programs of event:

- **On Start:** a program of this type run to start a game if the GameObject is active or was not active, the first (and only the first) once activated.

Activation events

Are the events related to the activation / deactivation of a GameObject. This type of event is managed by the following programs of event:

- **On Activate:** a program of this type will run whenever a GameObject pass from inactive to active. Not to be confused with On Start, though it may be equivalent to the case of a GameObject that when starting the game is off and is then activated at some later time.
- **On Deactivate:** a program of this type will run whenever a GameObject pass from active to inactive.

Events Update

Are the events that are launched regularly coinciding with processes themselves Unity engine update. This type of event is managed by the following programs of event:

- **On Update:** a program of this type will run at the beginning of each frame of play. It is the ideal type of program when we run fast and continuous actions.
- **On Late Update:** a program of this type will run the Update On completion of all programs.
- **On Fixed Update:** program running at fixed regular intervals and is used mainly to give orders to the Unity engine Physics.

Game Events

Are the events that are released when the general state of the game is changed. This type of event is managed by the following programs of event:

- **On Game Start:** a program of this type will be executed as a result of the change of state that produces the *Start Game* action. Indicates normally the game itself has begun.
- **On Game Over:** a program of this type will be executed as a result of the change of state that produces the *Game Over* action. Indicates usually the game is over.
- **On Game Pause:** a program of this type is executed when the game enters pause, usually as a result of an action or *Toggle Pause Pause Game.*
- **On Game Resume:** a program of this type will run when play resumes after being previously paused, usually as a result of an action or *Toggle Pause Resume Game.*

Mouse events

Are the events that are launched when the user perform actions with the mouse on elements that support it as GUI or *Colliders* elements. This type of event is managed by the following programs of event:

- **On Mouse Down:** a program of this type will run when you click any of the mouse buttons on a GUI or an element *Collider.*
- **On Mouse Drag:** a program of this type will run when mouse movement on a GUI or an *Collider* element is detected while any of the mouse buttons is kept pressed.
- **On Mouse Enter:** a program of this type is executed when the mouse enters the area occupied by a GUI element or an *Collider.*
- **On Mouse Exit:** a program of this type will be executed when the mouse pointer leaves the occupied by a GUI element or an *Collider* area.
- **On Mouse Over:** a program of this type will run when mouse move over a GUI element or an *Collider* is detected while not being pressed

any of the mouse buttons.

- **On Mouse Up:** a program of this type will run to release a mouse button was being pressed on a GUI or an element *Collider.*

Collision events

Are the events Physics engine Unity launches to report that there have been collisions between objects and also to indicate the status of such collisions. This type of event is managed by the following programs of event:

- **On Collision Enter:** a program of this type will be executed as soon as it detects that it has begun a collision between two objects.
- **On Collision Exit:** a program of this type will run as soon as it detects a collision between two objects no longer produced.
- **On Collision Stay:** a program of this type will run on every frame of the game while continuing to produce a given collision between two objects.
- **On Trigger Enter:** Collision On Enter similar but applicable when at least one of the objects has a marked as Collider *Trigger Is,* that is, that detects collisions but it is traversable by other objects.
- **On Trigger Exit:** Exit similar to On Collision but applicable when at least one of the objects has a marked Collider *Is Trigger.*
- **On Trigger Stay:** On Collision similar to Stay but applicable when at least one of the objects has a Collider *Is* marked as *Trigger.*

It is important to note that as these few events that always two objects involved, your notice is double because the collision is reported to both objects alike. This means you can associate a program of this kind to any of the two objects susceptible to collide and in both cases the program will run. The only difference, as we will be in the parameters sent to each program.

Time Events

They are weather-related events and components that work based on time, as the Timer *(Timer).* This type of event is managed by the following programs of event:

- **On Timer Expire:** a program of this type are executed when a *Timer* component notified that the time interval has elapsed for which had been scheduled.

Program events

These are events that launch the programs themselves to report changes in your state. This type of event is managed by the following programs of event:

- **On Program Finish:** a program of this type will be executed when a program in the range of listening finished executing.

Variables events

They are throwing events GameFlow to notify that a variable (or parameter) has been amended. This type of event is managed by the following programs of event:

- **On Variable Change:** a program of this type will run in both runtime and editing when a variable in the range of listening has been modified.
- **On Parameter Change:** a program of this type will run in both runtime and editing when a parameter in the range of listening (which in this type of program is limited to GameObject father) has been modified.

Event Handlers

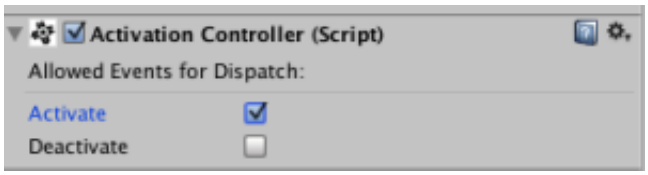
Event handlers are special components that serve as general event filters. When added to a GameObject allow you to specify which of the events generated by this GameObject that correspond to the type managed by the controller should be notified and which will simply be ignored.

This control is especially useful for optimizing the performance of a game as it reduces the number of internal messages that the system must process making less processing time per frame needed.

Controllers types supported events are:

- **Activation Controller:** Activation event handler.
- **Collision Controller:** Controller collision events.
- **Mouse Controller:** mouse event handler.
- **Trigger Controller:** Controller *Triggers* collision events.

For a type of event is notified its *checkbox* must be checked, while those events with the unlabeled be ignored.



Flow Control

The program flow is the order in which a program executes its action list. The program flow is sequential by default, that is, it will first run the first action enabled to completion, then the next action and so on until the end of the program, but this order does not always allow us to get the results we need in our game, that's why GameFlow gives us the ability to modify this flow in a simple way by using a kind of specialized actions in this task.

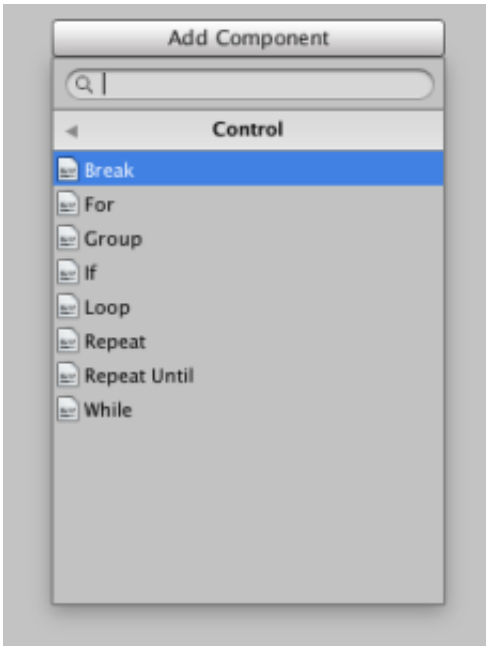
With these specialized actions, which will detail then be possible to implement sophisticated logic game in which:

- Parts of a program are executed repeatedly a number of times. This in programming methodology is called "loop".
- Certain parts of a program are executed only when met (or not met) certain conditions. This is called "fork" or "conditional execution".
- Certain parts of a program not run sequentially, ie action by action, but running all their actions simultaneously as a group.

The way we control the flow of programs largely defines the programming quality of our game, which is why we recommend you clear the basic ideas of programming methodology before attempting to build programs with a flow control very complex.

Flow Control Actions

As mentioned, GameFlow lets you control the flow of programs through the use of a panel of actions that can be found under the *Component> GameFlow> Control menu.*



The purpose of each of these actions is as follows:

- **If:** allows execution to branch based on the result of a list of conditions. If these conditions are met (the overall result is True) contained the main list of actions will be executed, while if the result is False a list of secondary shares (shares under the *Else* section) will run.
- **For:** create a loop execution for a list of actions contained (in other words, perform these actions repeatedly) a number of times which is given by the value of a variable that is modified at each iteration of the loop or turn and limit value against which you will compare.
- **Repeat:** F*or* a simplified version that requires only specify the number of times you want to repeat the actions contained, without variables or limit value.
- **Loop:** a simplified version of *Repeat* as default execution will loop will be repeated indefinitely well until the program is stopped or until the actions contained within one *Break* action (see description below) is executed.
- **While:** implement the actions contained if a certain condition is met and will continue to loop while still meets the condition.
- **Repeat Until:** implement the actions contained in loop until the list

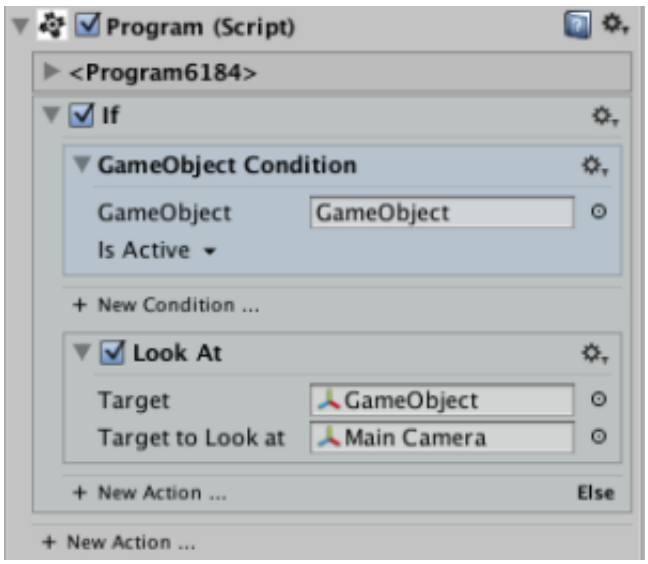
of associated conditions are met.

- **Group:** allows you to run all contained actions in parallel, ie, in this type of block is not expected that an action has finished running, but all actions are executed at a time and only when all actions end when you consider that the action is over.
- **Break:** stops a loop, making the program running in the next action after generating the loop is activated.

Terms

The conditions are a type of special blocks that use some actions to determine, in its assessment, how or when to change the program flow.

Visually conditions are very similar to those shares because like these are represented as collapsible blocks containing property fields that allow us to configure the operation of the condition. Only vary in color, which is different from the actions and their location, as they can only live in shares that have been designed for use conditions.



As for editing operations, conditions support the same editing operations that actions within their own contexts and through their context menus.

Evaluation

Evaluate a condition is to check if it is true or not true. In programming, a condition is met when we say that the result is True, False when not being met. The result of the evaluation of a condition is what we call a Boolean value, as only supports two possible values opposite each other.

This result and its interpretation depend largely on how the condition has been configured through the properties stated in its interface, which will normally include at least one comparison field to indicate what exactly we want to do within the condition.

GameFlow not only supports the evaluation of a condition but that creates a list of conditions linked by logical operators that always be evaluated using the last result and the result of the condition currently being evaluated. These operators can be:

- **And:** that will return True as result if (and only if) the previous result is True and the condition under which connects also evaluates to True. Otherwise, it will return False as result.
- **Or:** that will return results for both True if the previous result was true as if the result of the condition with which connects evaluates to True. Only returns False if both results are false.

As we have seen, the way in which a control action interprets the result of evaluating its list of conditions depends on the internal logic of each action. To clarify any doubts you can always consult the contextual help for each action.