# NOTTORUS

## Getting Started

V1.00

2016

# 1. Introduction

Nottorus Script Editor is a visual plugin for generating and debugging C# Unity scripts. This plugin allows designers, artists or programmers without any limitation create C# scripts of any complexity, but require some basic programming skills. It contains all the features of the C# language.

# 2. Installing

Download and import plugin unitypackage from the Asset Store. After installation all plugin components it will be located in "**%ProjectFolder%/Assets/Nottorus**" folder. Plugin options and scripts autosaves will be located in "**%ProjectFolder%/Nottorus**" folder.

# 3. Updating plugin

Just import the updated unitypackage from Asset Store. If you have some troubles with updating plugin please delete the plugin folder "**Assets/Nottorus**".

# 4. Launching

Nottorus script editor can be launched from Unity top menu bar "**Tools/Nottorus/Nottorus script editor**" or select any Nottorus script data file (with extension .nts) and pres '**Open in Nottorus**' button in Inspector panel.

# 5. Creating new script

New scripts can be easily created with script creation wizard, which will configure the script depends on the selected presets:

- **Monobehaviour** scripts are used on GameObjects. This is the main scripts responsible for gameplay logic.
- **For Inspector**. Inspector script is an editor script, which allows drawing custom inspectors for Monobehaviour scripts.
- **Editor Plugin** allows creating custom editor window with interface that can be float free or docked as a tab, just line native windows in the Unity interface.
- **Empty script.** For creation custom classes or other code objects.
- **Parse .sc script**. Try to convert SCharp code from selected source file into the nodes.

# 6. Creating scripts

## 6.1. Script objects

Script can contain different types of code objects (classes, functions, variables, etc) that can be created in the left toolbar of editor. All unused members can be hidden in bottom menu of this toolbar. In this chapter described each of them.

### 6.1.1. Class

Basically each script contains class. A class is a construct that enables you to create your own custom types by grouping together variables of other types, properties, functions etc. A class is like a blueprint. It defines the data and behavior of a type.

### 6.1.2. Struct

Structs share most of the same syntax as classes, although structs are more limited than classes. For more information check structs documentation page.

### 6.1.3. Constructors

Constructor is called when a class or struct is created. A class or struct may have multiple constructors that take different arguments.

### 6.1.4. Fields (variables)

A field is a variable of any type that is declared directly in a class or struct.

### 6.1.5. Functions (methods)

A function is a code block that contains a series of nodes (statements). A program causes the statements to be executed by calling the function.

### 6.1.6. Properties

A property is a member that provides a flexible mechanism to read, write, compute the value or execute some code before setting or getting it.

### 6.1.7. Indexers

Indexers allow instances of a class or struct to be indexed just like arrays. Indexers resemble properties except that their accessors take parameters.

### 6.1.8. Enums

The enum keyword is used to declare an enumeration, a distinct type that consists of a set of named constants called the enumerator list.

### 6.1.9. Delegates

A delegate is a type that represents references to methods with a particular parameter list and return type. When you instantiate a delegate, you can associate its instance with any method with a compatible signature and return type. You can invoke (or call) the method through the delegate instance.

### 6.1.10. Interfaces

An interface contains only the signatures of functions, properties, events or indexers. A class or struct that implements the interface must implement the members of the interface that are specified in the interface definition.

## 6.2. Making code

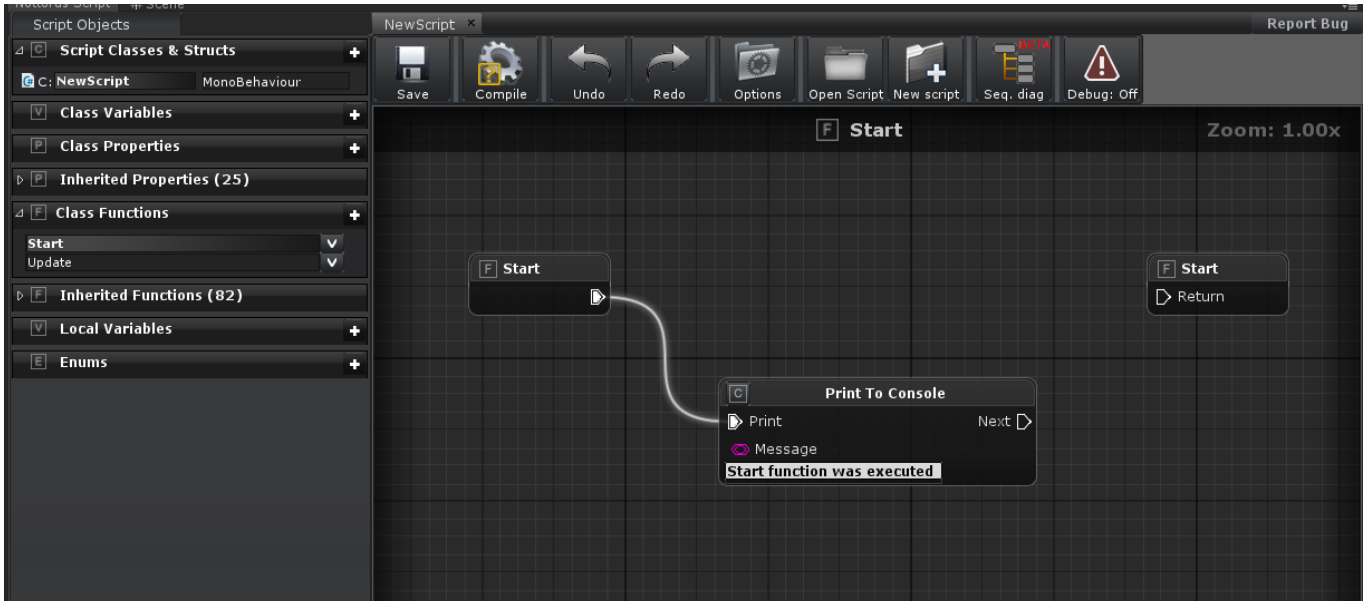
After script was created you can start making your code. If script was created from one of presets in script creation wizard (Monobehaviour/Inspector script/Editor Plugin) you will got few already created functions, which are most used among the default unity script functions of this script type.

## 6.3. Node edit graph

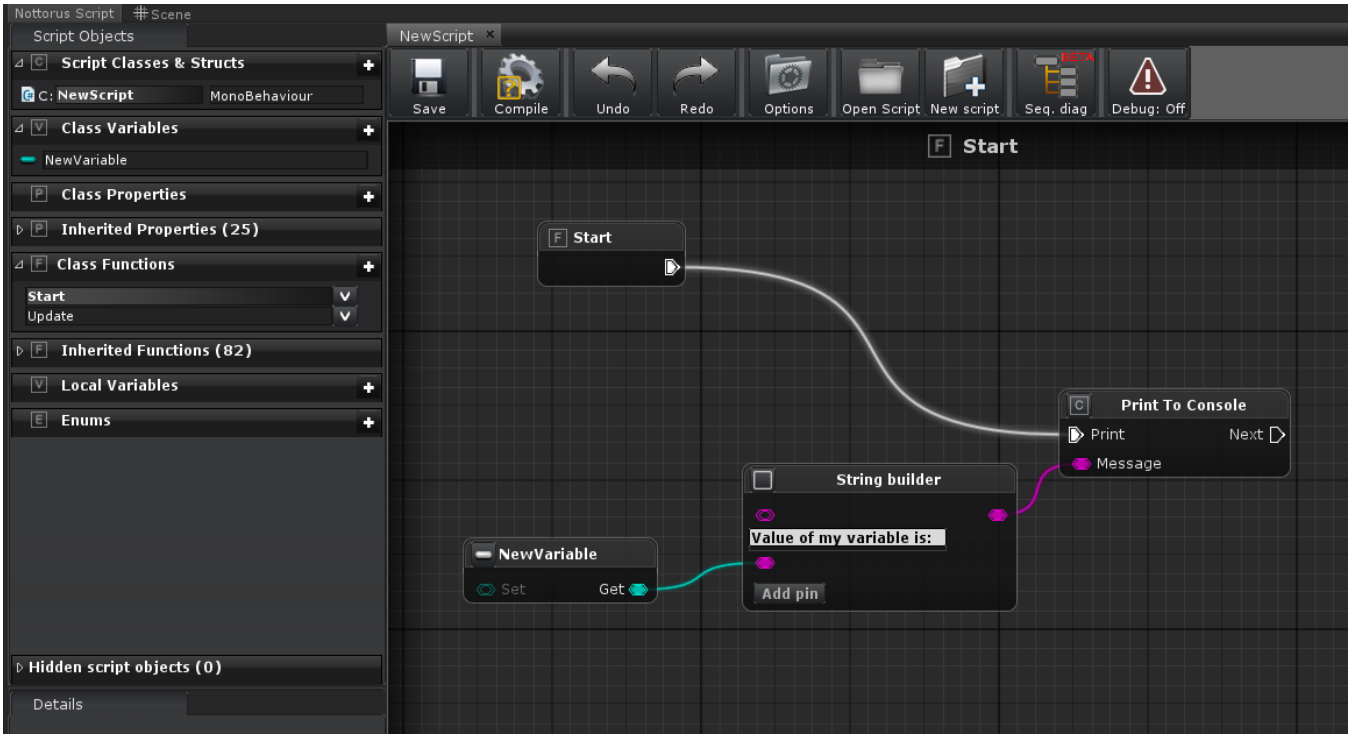
Each script function, property or class constructor has its own node edit graph, which holds all nodes of code sequence and it has by default one **entry** node and one **return** node. All code in edit graph starts executing from entry node, and finish executing when it comes to the **last node** of sequence (no next node to execute) or when reached to **return** node, which terminates executing current function and also used for returning value from function.

**Entry node** can't be **duplicated** in edit graph, unlike **return node**, which can be **copy-pasted** and used at any place in the code. **Entry** and **return** nodes can't be completely **deleted** from edit graph.

All **sequence nodes** must be **linked** with white **bold line**, which starts from entry node connector and goes from one node to another. This line displays nodes execution order. The connection is made by dragging one connector to other. One of connectors must be **'in'** direction and other is **'out'** direction:

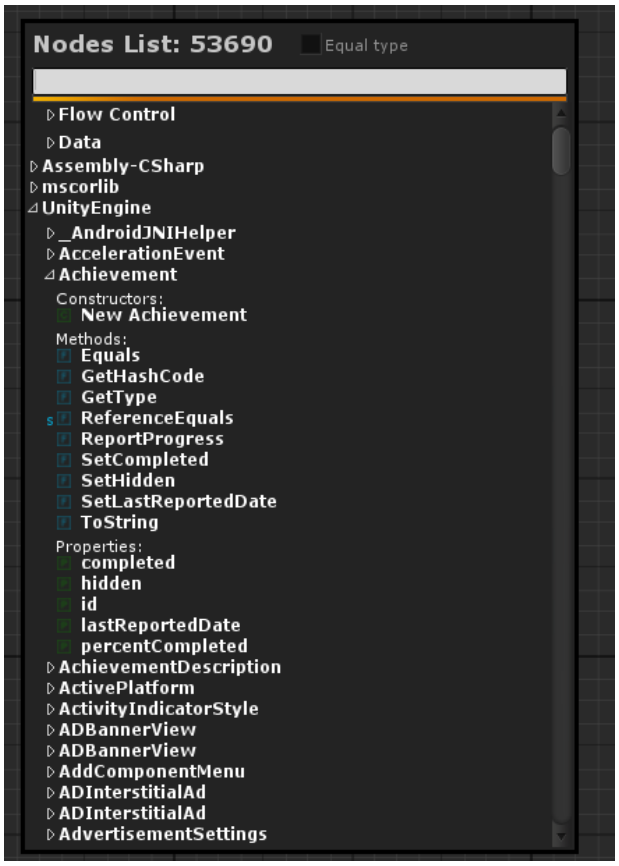



Data nodes have one **out pin** and must be connected to **in pin** of other nodes. The color of line is defined by type of data variable:



## 6.4. Using nodes

All available nodes can be found in **New Node Menu**, which opening by clicking right mouse button on edit graph:



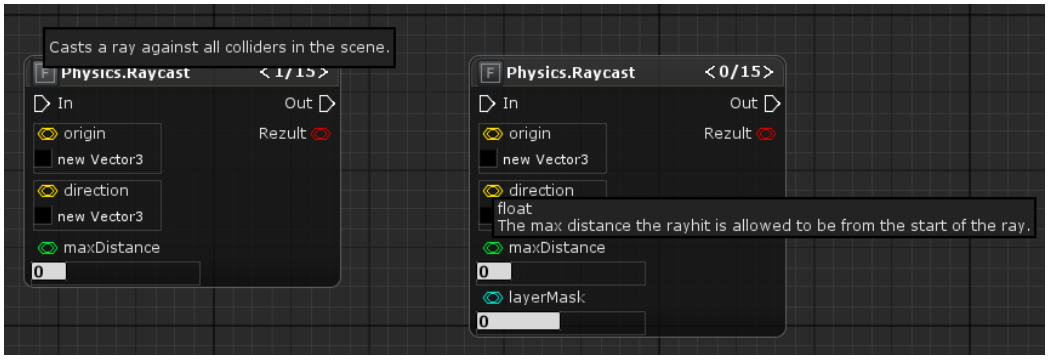

Above in the list the default nodes are located. Most of them are the standard functionality of C# language.

- **Flow control** nodes used for controlling sequence of code and can be executed by sequence connector;
- **Data** nodes used for data calculation, conversion, binary operations, etc.

After default nodes goes other nodes, which generated (by C# reflection) from used assemblies (dll). They provide access to all functions, properties, constructors of Types from those assemblies.

## 6.5. Nodes description

Almost all nodes has own description which shown as a tooltipe of node and pin icon (or icon in new node window nodes list).



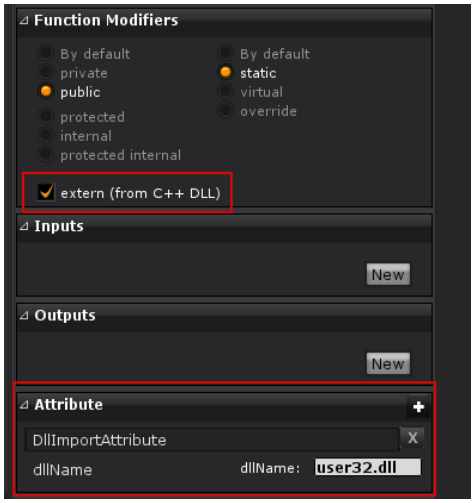
## 6.6. Used libraries

Connected assemblies can be checked in ‘Assemblies’ button in New Nodes List.
Description of some libraries:

- **mscorlib** – default library for C# language, which contains all basic Types (bool, int, string, etc.). Recommended always be turned on.
- **UnityEngine** – base library for Unity Engine, which contains all basic Types (GameObject, Transform, Vector3, etc.). Recommended always be turned on when you work with Unity Engine objects (types).
- **UnityEditor** – library which takes access to all Types for creating Unity editor scripts.
- **Assembly-CSharp** – gives **access** to all Types in **custom scripts** of current project. Can be turn off when no need to access to other project scripts or while developing scripts which not dependents on other project scripts.
- **Assembly-CSharp-Editor** – same as Assembly-CSharp, but used for **access to custom editor scripts** in current project.

### 6.6.1. Custom libraries

Custom libraries must be placed in any Assets folder. In case C# library it must be checked in Asseblies options in New Nodes List and after reopening it all library types will be available in New Nodes List. In case C++ library must be created a **static** function with **‘extern’** option, with name as external function name and attribute DllImport:



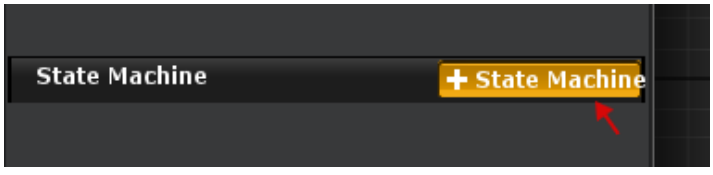

## 6.7. Compiling script

Script compilation is performing be pressing ‘Compile’ button. Before script file will be written to the disk the **pre-compilation** will be performed. It will compile our script code and detect all possible errors to **prevent the script with an error was written to a project**.

Pre-compilation is disabled on MAC due to incorrect work MONO compiler in Unity on this platform.

# 7. State Machine

A finite state machine (SM) used for simplifying creation of gameplay logic or A.I. behavior. It must be created in **class** that derived from **Monobehaviour** by pressing ‘**Add state machine**’ button:

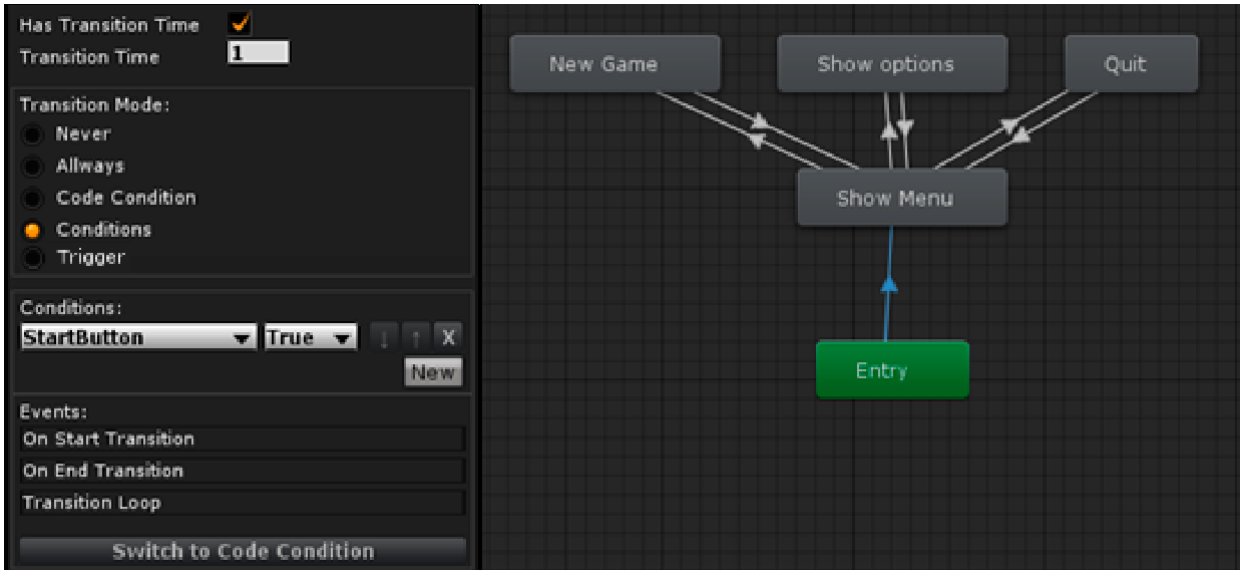


State machine change node (state) to other through links (transitions) if it passes link condition. State will wait until any first link will pass the condition then transfer to other state. If there are few links pass condition – state will use the first priority link. Entry node it's an entry point of sequence in state machine.

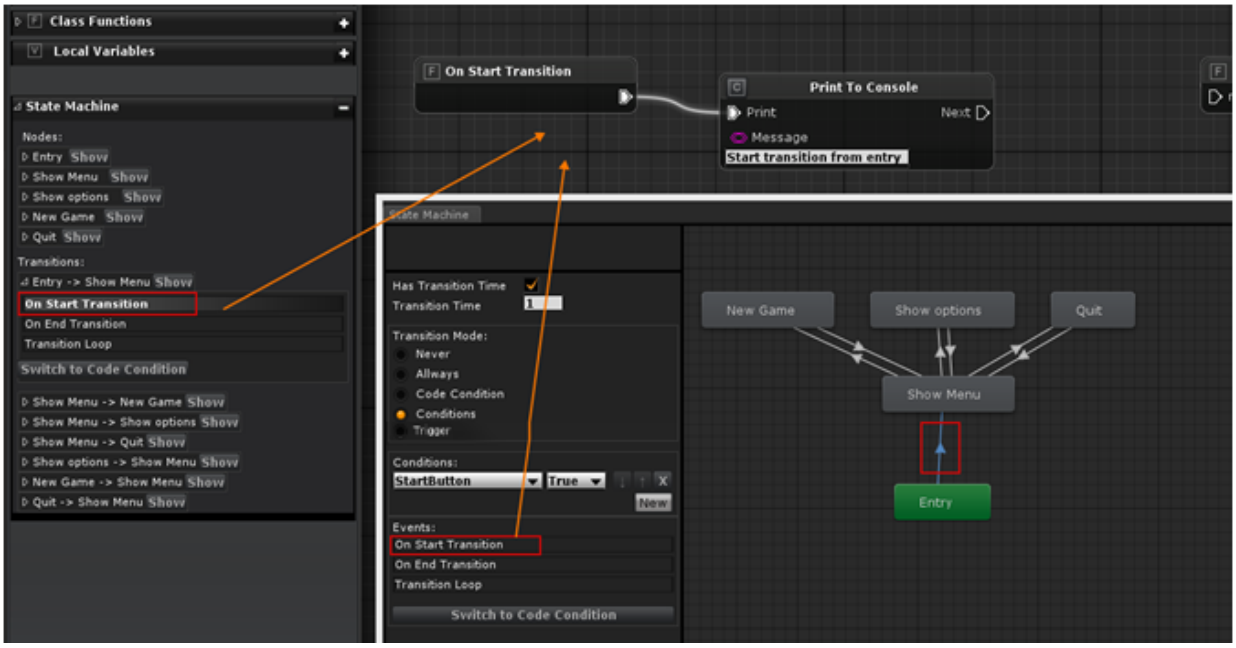
Transitions have 5 modes for transition:

- **Never** – transition is disabled. Can be used for temporary disabling transition;
- **Always** – always allow to transit;
- **Conditions** – transition allowed if all conditions will be passed. Each condition checks the value of script variable;
- **Code Condition** – transition permission based custom script function, which returns bool value;
- **Trigger** – based on bool variable. After trigger pass, the value of this variable will be set to false.

**'Switch to Code Condition'** button will check on **Code Condition** option and opening this function.




Each state and transition has events, which can be used for executing nodes in Nottorus Script Editor:



Nodes events:

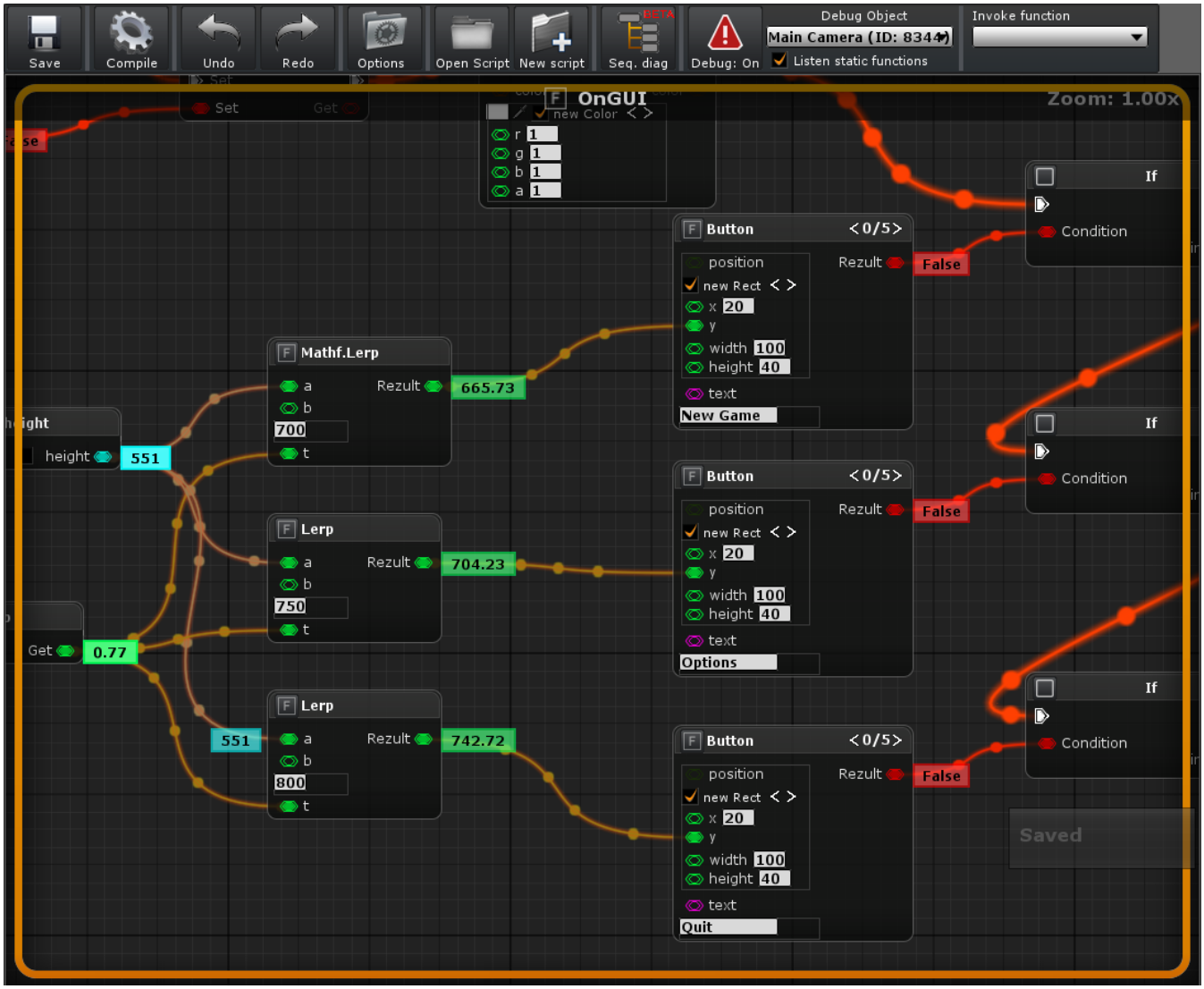
- **Do Before Entering** – calls before SM enter to this node;
- **Do Before Leaving** – calls before SM leaves from this node;
- **State Loop** – calls while SM stay in this node (waiting for any link will pass condition). Will not be called if SM enters to node which has any out link that pass the condition.

Transition events:

- **On Start Transit** – calls before SM will use this link for transition;
- **On End Transit** – calls after SM ends transition using this transition;
- **Transition Loop** – calls while SM transit using this link (if transition has 'Transition time' option) and gives transition delta value.

## 8. Debugging

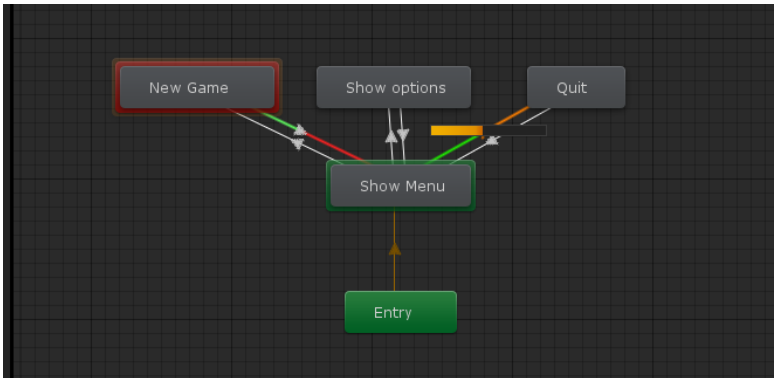
Visual debugging is a powerful tool, which allows debugging sequence of code, data values of each parameter in nodes and sequence of state machine.




To **enable visual debug** the **script must be compiled** with '**Debug: On**' option at the top of toolbar. The chosen value of the filter '**Object of debug**' will define the object which we want to debug. In case Monobehaviour scripts we can select a GameObject with our script on, otherwise we can chose 'Any object',

which allows receiving debug from any script (if we create editor plugin or other script which can't be used on GameObjects).

We will receive debug information while script will work.




## 9. Autosaves

Nottorus Script Editor automatically makes **backup** of a saves (every 5 min by default) into "**%ProjectFolder%/NottorusEditor/Autosaves**" folder with the possible compression as zip file. Autosave delay and other option of this feature can be found in **Options** dialog of editor.

## 10. Saving scripts

Nottorus Script Editor saves all script objects, nodes, and other plugin system data to the files with extension .nts. After compiling .cs script file will be created in a same folder as .nts file.

All **editor scripts** (Inspector scripts, editor Plugins and all related with them) must be **located** in any **Editor** folder in project. These scripts will be ignored during building the game and not be compiled to build. Plugin check their location while saving new script and show message in case invalid location.

**Monobehaviour** script class must be **named** as its **file name**, otherwise this script **can't be used** on **GameObjects**.

## 11. Examples

Example scripts and scenes can be found in '**Assets/Nottorus/Examples'** folder. Read **Examples.pdf** for more details.