



Manual

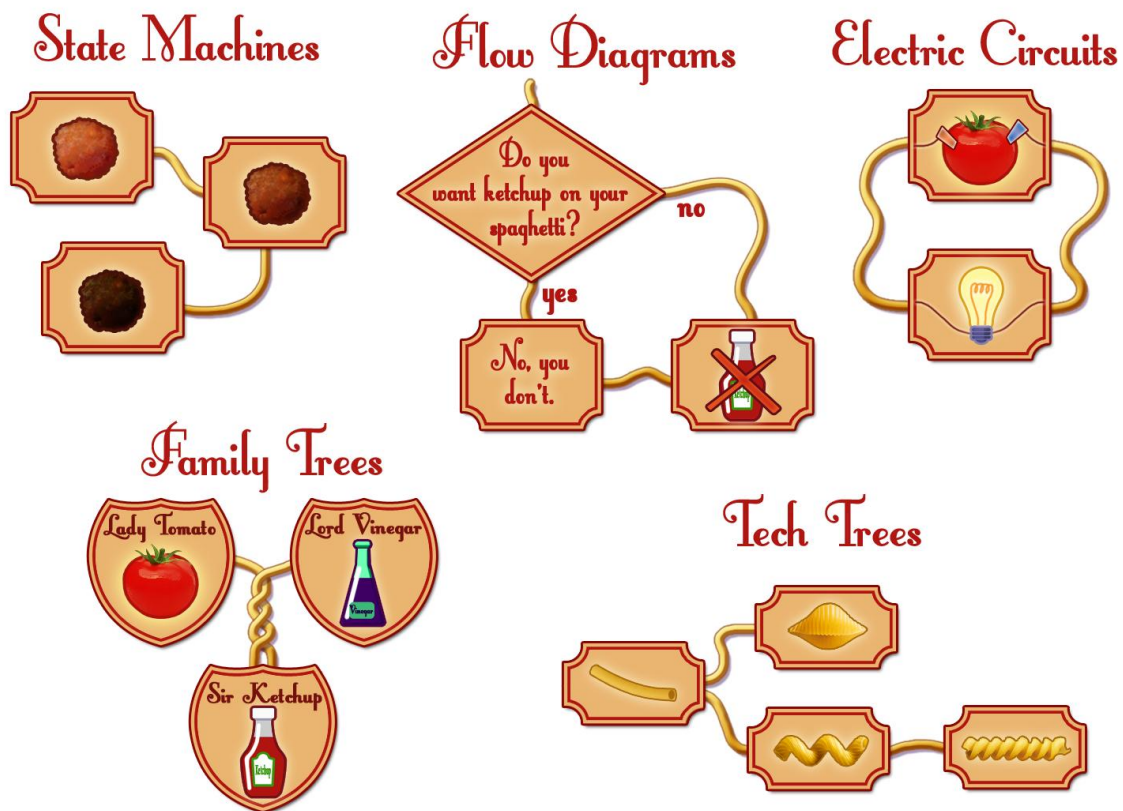


INTRODUCTION

WHAT IS THE SPAGHETTI MACHINE?

The Spaghetti Machine (TSM) provides a visual interface for editing graphs (for example state machines, dialog trees, flow diagrams or tech trees), and methods for importing them into your application.

For those among you who wonder what the heck I'm talking about: A graph is basically *stuff connected by lines*. Since the dawn of mankind¹, people have visualized abstract dependencies by connecting stuff by lines, from family trees up to subway plans. Here just some examples of stuff connected by lines:



In all these examples you see clearly the two main components of any graph: *Nodes* (the "stuff" connected by lines, and *edges* (the actual lines).

In many cases, the edges have a direction; they go from node A to node B and not the other way round. In this case we speak of a directed graph.

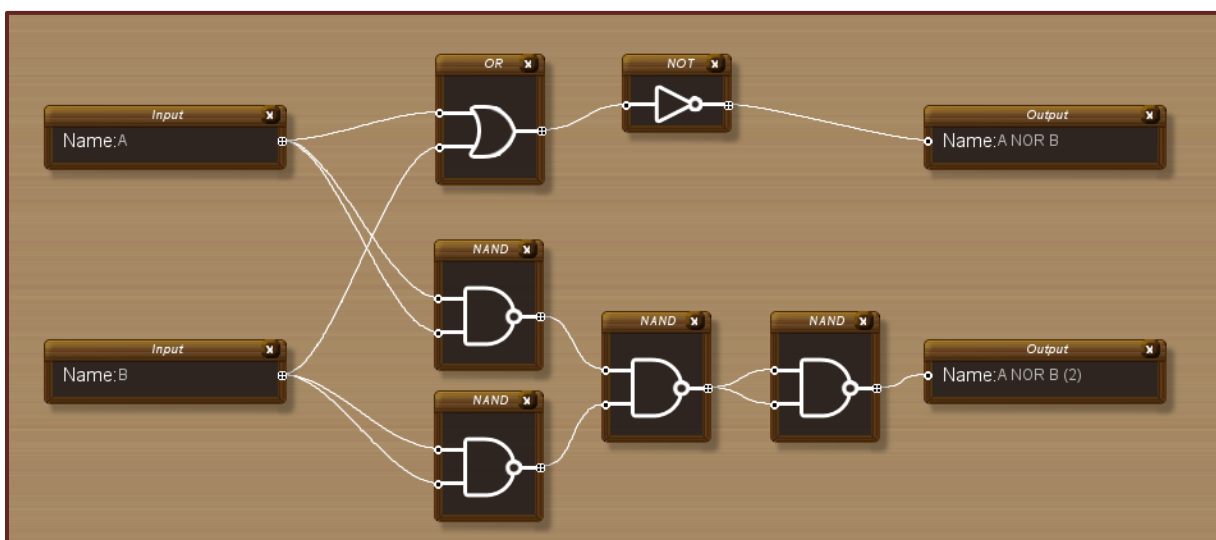
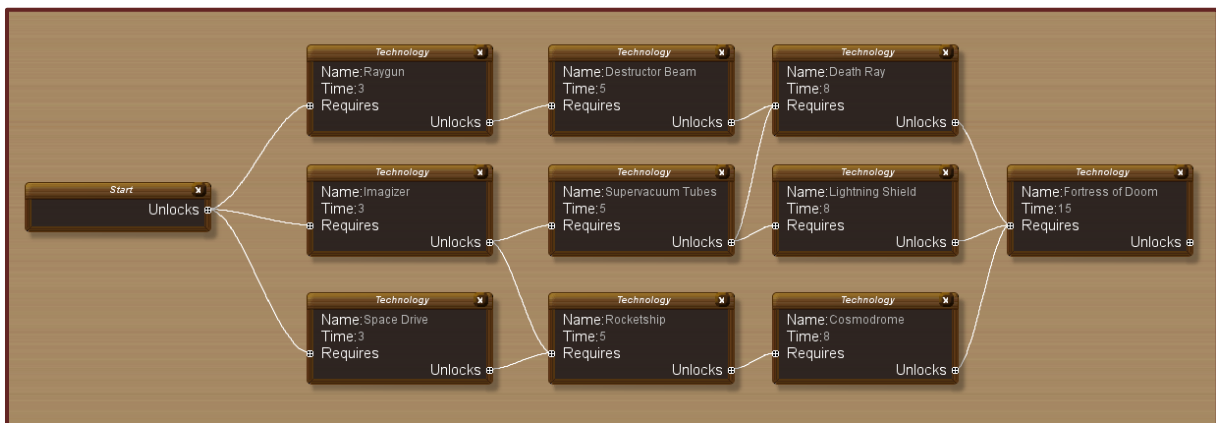
¹ I'm not making this up. In Gua Tewet, Borneo, there is a prehistoric cave painting, over 10000 years old, which actually consists of hands connected by lines.

Now, you don't need an application to draw a graph – all you need is a pen and a piece of paper, or a paper napkin, or whatever. But unfortunately, Unity provides no method *ImportFromPaperNapkin*. That's where The Spaghetti Machine comes in: It allows you creating whatever kind of graph you need, in a format that can be read by your scripts. More precisely, you can:

- **Define** a set of nodes you will need in an XML file
- **Edit** graphs with the visual editor, based on one or more node sets
- **Import** those graphs into your application
- **Access** the nodes, edges and data attached to nodes

WHAT DO THOSE GRAPHS LOOK LIKE?

Here two examples of what a spaghetti machine graph might look like in the editor:



The name "Spaghetti Machine" comes from the similitude of the curved edges with the homonymous Italian pasta, and from the fact that one can use those "spaghetti" to build state machines and many other kinds of virtual "machines".

GETTING STARTED

INSTALLATION

- To integrate the Spaghetti Machine to any project, import the *Spaghetti Machine* package.
- To open the Spaghetti Machine editor window, select "TSM Editor" in the "The Spaghetti Machine" menu.

EXAMPLE 1: FIRST STEPS

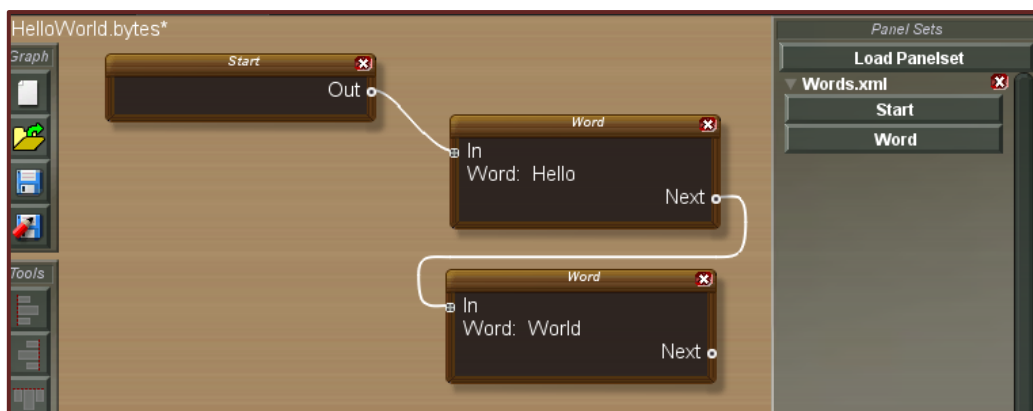
Create an empty project. Import the *Spaghetti Machine* package and the *Examples* package. Click on the menu "The Spaghetti Machine > TSM Editor" to open the editor.

On the left side, click on "Load Panelset" and load the file

SpaghettiMachine\Examples\Example1 - Hello World\PanelSets\Words.xml.

A new window "Words.xml" appears on the right side, containing two buttons "Start" and "Another panel type". Click on these buttons to create some panels. You can also duplicate a panel with Ctrl-D.

Try to move the panels around and connect them by clicking on an output plug and then an input plug, or the other way round. Right-click on a plug to delete its connections. You may notice that some connections have a strange "S" or "inverse S" shape; this makes the connection better visible in the frequent case that a panel is linked to another panel below or above the first one.



You can select multiple panels by dragging a box with the left mouse button. This allows you to move multiple panels at once. You can also move the whole graph by "dragging" on the ground with the right mouse button. When you do so, you see a "minimap" in the center of the screen. This facilitates navigating in larger graphs.

You may also notice the fields "Name" and "Stuff" in the panels. Those contain custom data which can be modified in the editor. Try it! Finally, you can click on "Save as" on the left side and save your graph.

Congratulations! You just edited your first spaghetti graph! In a game you would now have earned your first achievement badge.



If you want to test your graph, save it under the name "HelloWorld.bytes" in the diagrams folder of example 1. Load the scene "HelloWorld" and run it : The program contains a script which writes the words one by one in the console. In further chapters we will learn how to write such a program.

```
! Hello  
UnityEngine.Debug:Log(Object)  
! World  
UnityEngine.Debug:Log(Object)
```

❧ BASIC NOTIONS ❧

PANELS, SLOTS AND PLUGS

To avoid confusion with XML nodes, the thingies you connect in TSM are called *panels*. Each panel contains the following elements:

	TYPE The <i>type</i> is a name you give to a certain kind of panel you define. Different types of panels have in general different elements.
	SLOTS Each panel contains one or more <i>slots</i> . Usually, a slot is a "line" on the panel. A slot may contain a plug, some data ("content") or both.
	LABEL Each slot has a <i>label</i> which serves to identify the slot within the application. The labels are defined with the type and can't be changed in the editor.
	PLUGS The <i>plugs</i> serve to connect panels. A slot can have no plug, an input plug (on the left), an output plug (on the right), or occasionally a hybrid "in-out" plug (details later).
	CONTENT Some slots contain data that can be modified during edition: A string, a floating point number, an integer etc. Those data are called <i>content</i> .

Note that a slot can contain some content *and* a plug at the same time, for example a slot representing a menu button containing the button name and an output plug.

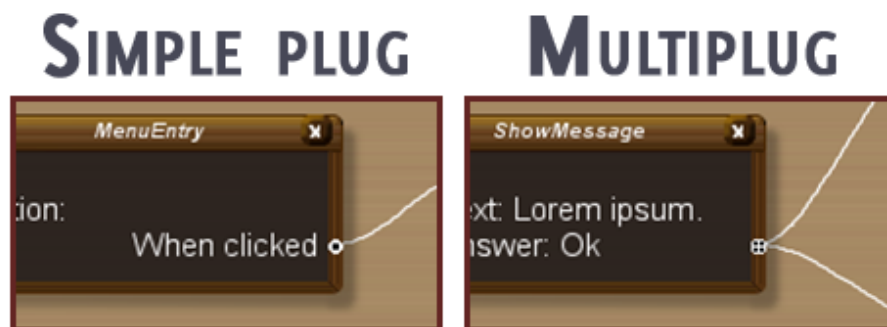
COLORS

Each plug has a color, the default color being white. You can only connect plugs having the same color – the connection line will have the same color. The purpose of colors is to distinguish different kinds of connections, and to prevent “senseless” connections.

Consider, for example, panels describing locations for an old-style text adventure. White lines may represent paths (when you go west, you end up in the forest) and green lines connect to objects (there is a small mailbox here). This makes the graph more readable and prevents editing errors (like going south and ending up in the small mailbox).

MULTIPLUGS

Some plugs (input or output) have one single hole, others have four tiny holes. The latter ones are called “multiplugs”. They can be connected to an arbitrary number of panels, simple plugs only to one panel. As for the colors, the purpose is to prevent erroneous, such as a character having two fathers, or a menu button leading to two different submenus.



PANEL SETS

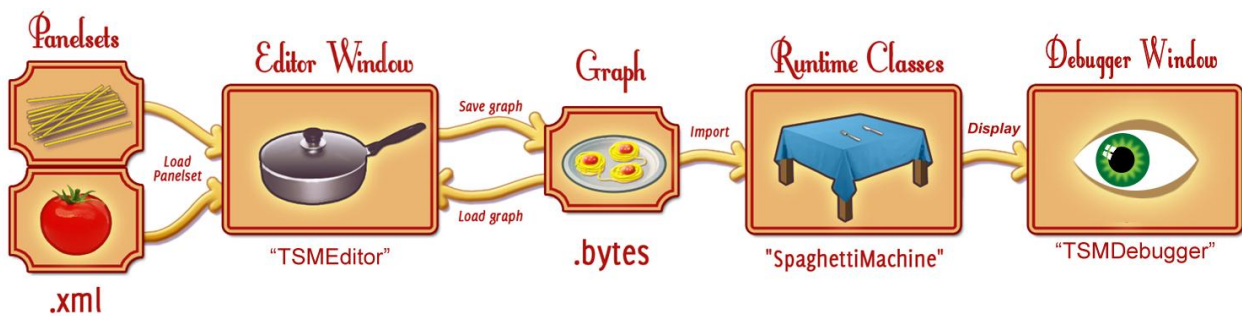
When creating a family tree graph, you don’t need the same kinds of panels as for a flow diagram or a menu system. That’s the purpose of panel sets: A panel set is a set of panel types you define for a specific purpose. You might, for example, use a panel set “family tree” which contains the node types “character” and “relationship”. Each panel set is defined in an XML file (details later).



THE WORKFLOW

To create and use spaghetti machine graphs, you basically proceed as follows:

- You define one or more panel sets (by editing simple XML files with a text editor)
- In the Editor Window, you use these panel sets to edit your graph
- You save the graph as .bytes file
- In your game, you use the Spaghetti machine runtime classes to load and use the graph
- You can observe the graph during runtime in the debugger window and see activated panels, custom variables, plug potentials etc.



AN EXAMPLE

EXAMPLE 2: TECH TREE MANAGEMENT

Imagine you are developing a real-time strategy game in an “old-school Sci-Fi” setting, and you need a tech tree system to manage development of rocket ships, death rays and other technologies². In this chapter, we will do exactly that. More precisely, we will

- Define a panel set for tech trees graphs
- Edit a tech tree graph
- Learn how to import the graph into the game at runtime
- Write a simple tech tree management system which interprets the graph
- Observe the program in the Debugger window

All files for this example can be found in Examples/Example2

CREATING THE PANEL SET

Create a new xml file called "TechTreeSet.xml" and open it with a text editor. Every panel set file starts with `<panelset>` and ends with the tag `</panelset>`:

```
<panelset>

</panelset>
```

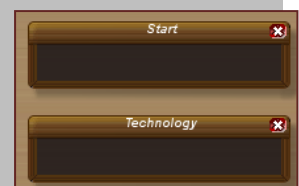
Now we start defining the panel types we need. Basically, we need two kinds of panels: A "Start" panel type, and a panel type for technologies:

```
<panelset>
  <paneltype type = "Start">

  </paneltype>

  <paneltype type = "Technology">

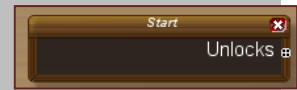
  </paneltype>
</panelset>
```



² This example was inspired by the “Captain Proton” holodeck programs in Star Trek Voyager.

The Start panel will serve as starting point for the program; it will be connected to the technologies unlocked from the beginning. So we only need one slot (let's call it "Unlocks") with an output plug:

```
<paneltype type = "Start">
  <slot label = "Unlocks" plug = "output" />
</paneltype>
```



Don't forget the slash / at the end of the slot line³.

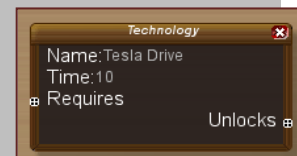
The "Technology" panel will have an input plug (for required technologies) and an output plug (for technologies it is required for):

```
<paneltype type = "Technology">
  <slot label = "Requires" plug = "input" />
  <slot label = "Unlocks" plug = "output" />
</paneltype>
```



Furthermore we need some data: The name of the technology, and the time it will need to develop. So we add two more slots, one with "string" content and one with "float" content:

```
<paneltype type = "Technology">
  <slot label = "Name" content = "string" />
  <slot label = "Time" content = "float" default = "1.0" />
  <slot label = "Requires" plug = "input" />
  <slot label = "Unlocks" plug = "output" />
</paneltype>
```



That's all. The whole XML file should now look like this:

```
<panelset>

  <paneltype type = "Start">
    <slot label = "Unlocks" plug = "output" />
  </paneltype>

  <paneltype type = "Technology">
    <slot label = "Name" content = "string" />
    <slot label = "Time" content = "float" default = "1.0" />
    <slot label = "Requires" plug = "input" />
    <slot label = "Unlocks" plug = "output" />
  </paneltype>

</panelset>
```



Save the file. We are done.

³ I keep forgetting them all the time...

EDITING THE GRAPH

Let's go to the Spaghetti editor window. Start a new graph (button "new" in the upper left corner). Now you want to load the panel set you just created. Click on "Load panel set" and select your xml file.

You see two new buttons on the right, one for each panel type. Click on "Start" to create a start node, and on "Technology" to create the first technology node. Place them on the surface, connect them and enter name and time for the technology. Add more technologies, either with the "Technology" button or by duplicating the first technology (select panel and click ctrl-D). Connect the technologies to create your tech tree. When you're done, it could look like this:



Save your graph under the name, say, "Techtree01.xml".

IMPORTING THE GRAPH

Create a new scene and a new script *TechTreeMachine.js*. In Javascript, this script implicitly contains a class *TechTreeMachine* - but now we will declare it explicitly (like in C#) because we want it to be derived from *SpaghettiMachine* (namespace *Spaghetti*):

```
import Spaghetti;

class TechTreeMachine extends SpaghettiMachine
{
    var mstrTechTreeFile : String = "Examples/Example1/Diagrams/TechTree01";

    function Start()
    {
        // Load the graph
        LoadFromFile( mstrTechTreeFile );
    }
}
```

Attach the script to a new game object⁴. Note that this is possible because *SpaghettiMachine* is derived from *MonoBehavior*.

⁴ Or, if you are really feeling lazy, to the Main Camera :)

When you press "Start", the graph is loaded into the TechTreeMachine - although it doesn't do anything yet.

USING THE GRAPH

To implement our "Tech Tree Machine", we need to keep track of the status of technologies, whether they are locked, unlocked, under development or finished. The simplest way to do so is to store this information in the technology panels. We do so by attaching to each panel a custom variable "status" and initialize it with the value "locked":

```
// Make all panels locked
for( var panel : Panel in GetPanels() )
{
    if( panel.GetPanelType() == "Technology" )
    {
        // Attach a custom variable "status" to the panel and set its value to "locked"
        panel.SetVariable( "status", "locked" );
    }
}
```

Now we want to unlock all technology panels connected to the "Start" panel:

```
// Find the "Start" panel
var startpanel : Panel = FindPanelByType("Start");
startpanel.SetVariable( "status", "n/a" );

// Find slots connected to the "Start" panel's "Unlocks" slot
var aSlots : Slot[] = startpanel.FindSlot("Unlocks").GetConnectedSlots();
for( slot in aSlots )
{
    // Mark panel as unlocked
    panel = slot.GetPanel();
    panel.SetVariable( "status", "unlocked" );
}
```

The initialization is done. Now we have to implement the actual tech tree development. You will need techniques similar to the code above; we won't go into details in this document. You might try it on your own, or have a look at the script in the script folder of Example 2 (it's well commented).



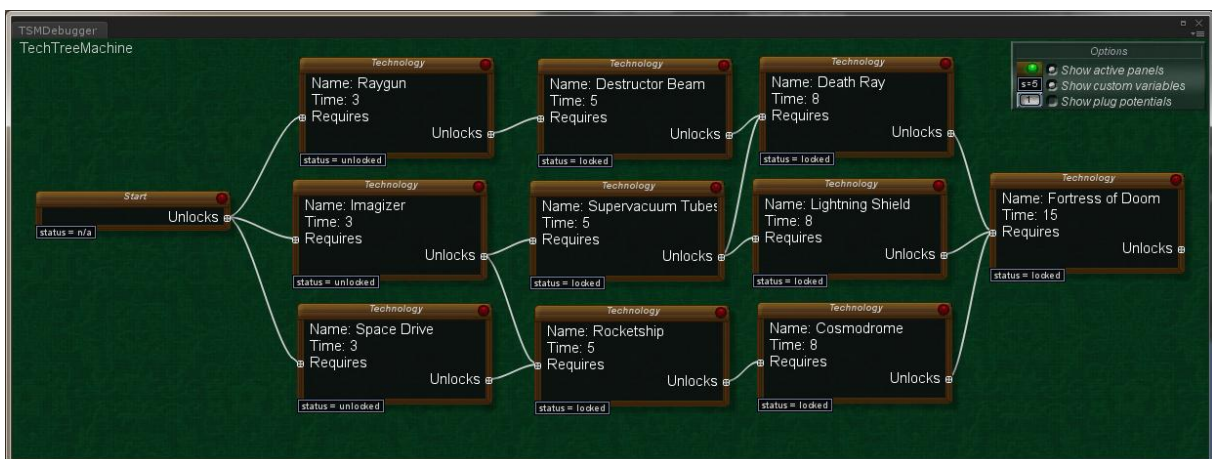
In the menu "The Spaghetti Machine", you can open TSM Debugger Window. It allows observing a Spaghetti Machine's behavior at runtime: Simply drag and drop the object containing the Spaghetti Machine from the hierarchy onto the window, and the graph will be displayed as soon as it is loaded.

This window works similar to the TSM Editor Window, except that you can't edit the panels (that's what the editor is for). You might consider the TSM Editor the "kitchen" where your spaghetti plate (=graph) is cooked, and the TSM Debugger the "restaurant" where they are consumed, and where you can discuss possible problems with the waiter. The "working table" and "tablecloth" backgrounds underline this metaphor.

In the upper right corner of the TSM Debugger, you see a little box with three checkboxes:

- Show active panels
- Show custom variables
- Show plug potentials

In this example, we only need to check the second checkbox, as we don't use the panel activation mechanism or the plug potentials.



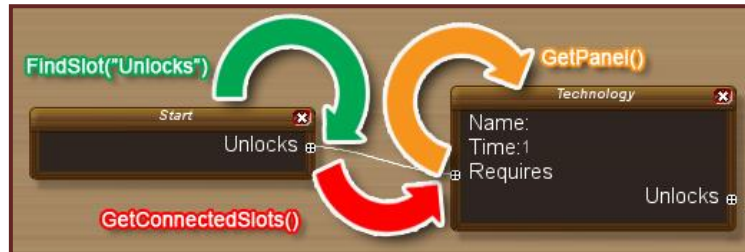
You see below each panel a little box "status = ...". This displays the custom variable "status" we have attached to each panel. As you run the program, you can observe your variable values changing. This is very useful if ever your program doesn't do what it's supposed to do.

In some cases you want to attach custom variables to slots, for example in a dialog tree, to mark dialog options that have already been chosen. Those variables are also displayed when you check "Show custom variables".

We will come back to the editor screen in further examples, to explain the other features.

TIP: WALKING ON THE GRAPH

In the last example, we have seen how to get from a panel to connected panels. We might call this technique "Walking on the Graph". This is usually done in three steps:



If the slot's plug isn't a multiplug, we can also use `GetConnetedSlot()` instead of `GetConnetedSlots()`, which allows going from one panel to the next one in one expression :

```
nextpanel = panel.FindSlot("Unlocks").GetConnetedSlot().GetPanel();
```

or, as a shorthand:

```
nextpanel = panel.FindSlot("Unlocks").GetConnetedPanel();
```

When you are working with the Spaghetti Machine, you will use this kind of expression a lot. Beware, thought, that this will return null if there is no connected panel, and go to a random panel⁵ if there is more than one.

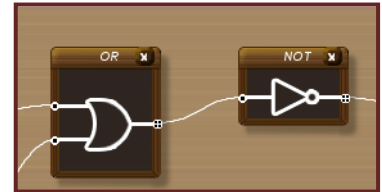
⁵ Actually, the first panel in the list. As connections of a slot are ordered by y coordinates of linked panels, this is the uppermost linked panel.

❧ ADVANCED CUSTOM PANELS ❧

EXAMPLE 3: LOGIC GATES

Say we want to create a panel set allowing to create logic graphs like this:

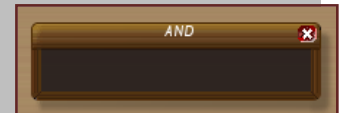
First thing we need are the bitmaps for the logical gates. You find the bitmaps for AND, OR, NOT and NAND gates in Examples/Example3/Resources. (To use a bitmap as background image in a panel type, you have to put the bitmap in a Resources folder, otherwise the program won't find it.)



Now we create a new panelset file LogicGates.xml. Let's start with an empty "AND" type:

```
<panelset>
  <paneltype type = "AND">

  </paneltype>
</panelset>
```



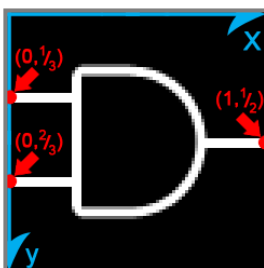
To use the "AndGate.png" background image, simply add an attribute *image* and specify the file name (without ".png"):

```
<panelset>
  <paneltype type = "AND" image = "AndGate">

  </paneltype>
</panelset>
```



You may notice that the window automatically adapts its size to the bitmap. (If it appears too big, select your bitmap in the project hierarchy and check if you set the Texture Type to "GUI" - otherwise Unity will change the dimensions to powers of two.)



Now let's add the first input plug. We have to specify the plug's position relative to the image. This is done with relative coordinates: The point (0,0) is in the upper left corner of the image, the point (1,1) in the lower right corner. You might want to keep that in mind when creating the bitmap, in order to have "simple" coordinates.

In our case, the first plug is on the left side, a third of the way down, thus at the coordinates $x = 0$, $y = \frac{1}{3}$. This is specified by the attributes *plug_x* and *plug_y*:

```
<panelset>
  <paneltype type = "AND" image = "AndGate">
    <slot label = "In_1" plug = "input" plug_x = "0.0" plug_y = "0.33" multiple = "false" />
  </paneltype>
</panelset>
```


Note that whenever you specify `plug_x` and `plug_y`, the label is *not* displayed as it won't probably be displayed anyway. It really only makes sense to use those when you use a background image.

You might also notice the attribute `multiple = "false"`. This specifies that the input plug isn't a multiplug. (Having multiple inputs wouldn't make much sense for a logic gate, would it?)

Let's add the other two plugs. Note that the output plugs should be multiplugs:

```
<panelset>
  <paneltype type = "AND" image = "AndGate">
    <slot label = "In_1" plug = "input" plug_x = "0.0" plug_y = "0.33" multiple = "false" />
    <slot label = "In_2" plug = "input" plug_x = "0.0" plug_y = "0.66" multiple = "false" />
    <slot label = "Out" plug = "output" plug_x = "1.0" plug_y = "0.50" multiple = "true" />
  </paneltype>
</panelset>
```



Now we add the other logic gates, plus an input and an output panel type:

```
<panelset>

  <paneltype type = "Input">
    <slot label = "Name" content = "string" plug = "output" multiple = "true" />
  </paneltype>

  <paneltype type = "AND" image = "AndGate">
    <slot label = "In_1" plug = "input" plug_x = "0.0" plug_y = "0.33" multiple = "false" />
    <slot label = "In_2" plug = "input" plug_x = "0.0" plug_y = "0.66" multiple = "false" />
    <slot label = "Out" plug = "output" plug_x = "1.0" plug_y = "0.50" multiple = "true" />
  </paneltype>

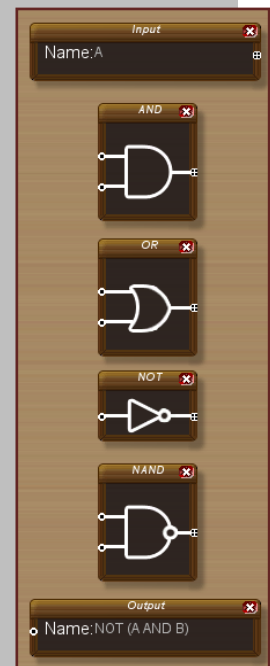
  <paneltype type = "OR" image = "OrGate">
    <slot label = "In_1" plug = "input" plug_x = "0.0" plug_y = "0.33" multiple = "false" />
    <slot label = "In_2" plug = "input" plug_x = "0.0" plug_y = "0.66" multiple = "false" />
    <slot label = "Out" plug = "output" plug_x = "1.0" plug_y = "0.50" multiple = "true" />
  </paneltype>

  <paneltype type = "NOT" image = "NotGate">
    <slot label = "In_1" plug = "input" plug_x = "0.0" plug_y = "0.50" multiple = "false" />
    <slot label = "Out" plug = "output" plug_x = "1.0" plug_y = "0.50" multiple = "true" />
  </paneltype>

  <paneltype type = "NAND" image = "NandGate">
    <slot label = "In_1" plug = "input" plug_x = "0.0" plug_y = "0.33" multiple = "false" />
    <slot label = "In_2" plug = "input" plug_x = "0.0" plug_y = "0.66" multiple = "false" />
    <slot label = "Out" plug = "output" plug_x = "1.0" plug_y = "0.50" multiple = "true" />
  </paneltype>

  <paneltype type = "Output">
    <slot label = "Name" content = "string" plug = "input" multiple = "false" />
  </paneltype>

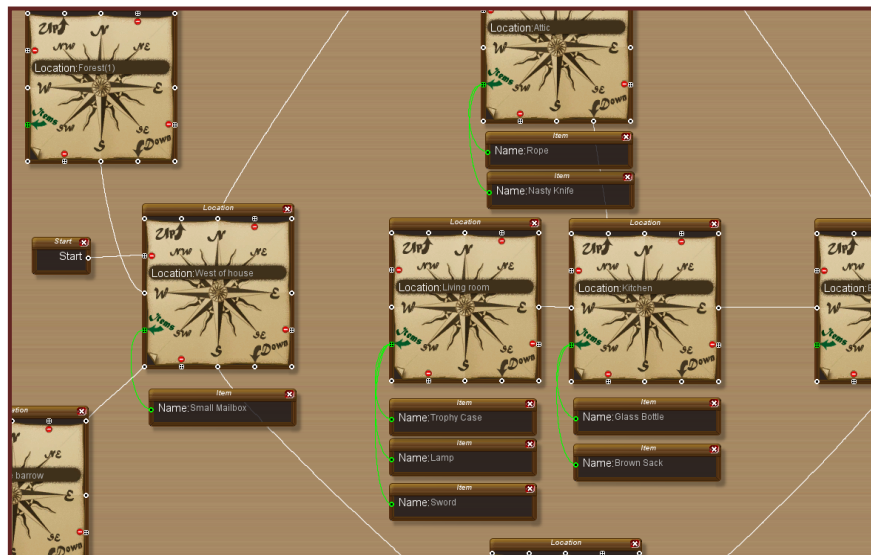
</panelset>
```



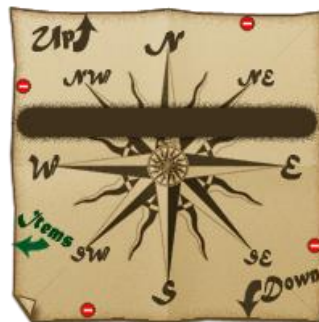
We're done. Now you can load your panel set into your editor and create the logic diagram of your dreams. Yay!

EXAMPLE 4: TEXT ADVENTURE MAP

Say you want to create an old-style text adventure where you explore a map going N or SW, examining stuff and using objects, etc. The Spaghetti Machine would be a great tool to edit the said map, with one panel for each location:



How would you do that? Well, let's start with creating the bitmap for the "Location" panels:



The brown bar is for the "Location" slot. For now place it where you would like to have the slot - we will adjust its position later. (Don't forget to save your bitmap inside a "Resources" folder.)

Our goal is to get a panel like this:



Now let's create the panelset. Besides the "Location" panel, we will need a "Start" panel and an "Item" panel:

```
<panelset>

  <!-- A typical start panel. (We reduce the panel width so it takes less place) -->
  <paneltype type = "Start" width = "80" >
    <slot label = "Start"    plug = "output" multiple = "false" />
  </paneltype>

  <paneltype type = "Location" image = "AdventurePanel">

    <!-- Some empty slots to position the location slot -->
    <slot label = " " />
    <slot label = " " />
    <slot label = " " />
    <slot label = " " />

    <!-- The actual location slot -->
    <slot label = "Location"    content = "string" />

    <!-- Plugs for directions -->
    <slot label = "NW"  plug = "inout" plug_x = "0.0" plug_y = "0.0" multiple = "false"/>
    <slot label = "N"   plug = "inout" plug_x = "0.5" plug_y = "0.0" multiple = "false"/>
    <slot label = "NE"  plug = "inout" plug_x = "1.0" plug_y = "0.0" multiple = "false"/>
    <slot label = "E"   plug = "inout" plug_x = "1.0" plug_y = "0.5" multiple = "false"/>
    <slot label = "SE"  plug = "inout" plug_x = "1.0" plug_y = "1.0" multiple = "false"/>
    <slot label = "S"   plug = "inout" plug_x = "0.5" plug_y = "1.0" multiple = "false"/>
    <slot label = "SW"  plug = "inout" plug_x = "0.0" plug_y = "1.0" multiple = "false"/>
    <slot label = "W"   plug = "inout" plug_x = "0.0" plug_y = "0.5" multiple = "false"/>

    <!-- Plugs for going upstairs/downstairs -->
    <slot label = "Up"    plug = "inout" plug_x = "0.25" plug_y = "0.0" multiple = "false"/>
    <slot label = "Down"  plug = "inout" plug_x = "0.75" plug_y = "1.0" multiple = "false"/>

    <!-- Some plugs for incoming "one way" connections -->
    <slot label = "In1"   plug = "input" plug_x = "0.75" plug_y = "0.0"/>
    <slot label = "In2"   plug = "input" plug_x = "1.0" plug_y = "0.75"/>
    <slot label = "In3"   plug = "input" plug_x = "0.25" plug_y = "1.0"/>
    <slot label = "In4"   plug = "input" plug_x = "0.0" plug_y = "0.25"/>

    <!-- Finally a green plug to attach items -->
    <slot label = "Items" plug = "output" plug_x = "0.0" plug_y = "0.75" color = "green" />

  </paneltype>

  <paneltype type = "Item">
    <!-- A slot combining green input plug and name -->
    <slot label = "Name"  plug = "input" color = "green" multiple = "false" content = "string" />

  </paneltype>
</panelset>
```

You might notice the empty slots at the start of the Location panel. Their goal is to position the "Location" slot as good as possible on the brown bar in the picture. You might need some tries. When you are done, take a screenshot and use it to reposition the slot on the bitmap. This way you get a bitmap which perfectly fits the slot placement.

Notice the "inout" plug type ? These are plugs without direction, like a door you can pass in either way. Of course, if you connect an inout plug to an input plug like "In1" (or to an output plug), the connection can only be used in one way.

You won't need inout plugs very often, but in cases like this one they come in very handy.

Once your panel set is created, you can edit an adventure map.

If you plan to actually code a text adventure, you might want to add some more slots (for example for the description of the location), and some additional panel types (for example an "Passageway Item" panel with a green input and a white output, or panels for possible actions). If you plan your panels carefully, it won't be difficult to code the Adventure Machine which allows players actually playing the game, exploring underground empires and collecting treasures..

Unless, of course, they are eaten by a grue.



SPAGHETTIMACHINE

This is the most important runtime class; it loads and manages one graph. SpaghettiMachine is inherited from MonoBehaviour, so you could attach it directly to a GameObject - but what you usually do is to derive your own class (e.g. MyMachine) from SpaghettiMachine, add some code of your own and attach it to a GameObject. We have already seen this in the Tech Tree example.

It is possible to handle simultaneously different graphs in the game; all you have to do is to create a SpaghettiMachine for each graph.

PANEL & SLOT

Those classes handle a specific panel resp. a specific slot. They are created automatically when a SpaghettiMachine loads a graph.

Those three classes are all you need. For details see the Runtime Class Reference, page [33](#).

USEFUL RUNTIME MECHANISMS

PANEL ACTIVATION

If you need to handle “active” panels, for example for a state machine, this mechanism is your friend. Several methods allow activating a panel, or activating all panels connected to a certain slot. When a panel is activated / deactivated, its master (by default the Spaghetti Machine object) receives a message *OnPanelActivated* resp. *OnPanelDeactivated*. (Note that if an active panel is activated again, no new message is send; idem for deactivation.)

In the SpaghettiMachine component, you can chose (in the inspector) between three behaviours of the spaghetti machine:

- **State Machine:** Only zero or one panels can be active. When you activate another panel, the current active panel is automatically deactivated (sending *OnPanelDeactivated* to the master). Typical uses: State machines, menu systems, dialog systems, non-linear campaigns etc.
- **Multistate Machine:** Several panels can be active simultaneously; activating one panel doesn't deactivate others. Typical uses: Unlocking locations, game systems where actions unlock new game elements. (However, multistate machines may be the least frequent variant.)
- **Trigger Machine:** When you activate a panel, it doesn't stay activated (actually it's deactivated some lines of code below), but nevertheless the *OnPanelActivated* message is sent. (*OnPanelDeactivated* is never sent in this mode.) Typical uses: Game systems where actions trigger events (for example trigger traps or start dialogs)

HYBRID SITUATIONS

- In some cases you might want a mixed **trigger/multistate** machine. Our advice for these situations is to use a multistate machine; in the “trigger” situations, you can deactivate the panel in *OnPanelActivated*.
- If you want to use a hybrid **trigger/state** machine, you can create a trigger machine and handle the “state” part by memorizing the “current state” panel and doing Walking On The Graph (see example 6).
- The same trick can be used to create a mixed **multistate/state** machine.

EXAMPLES

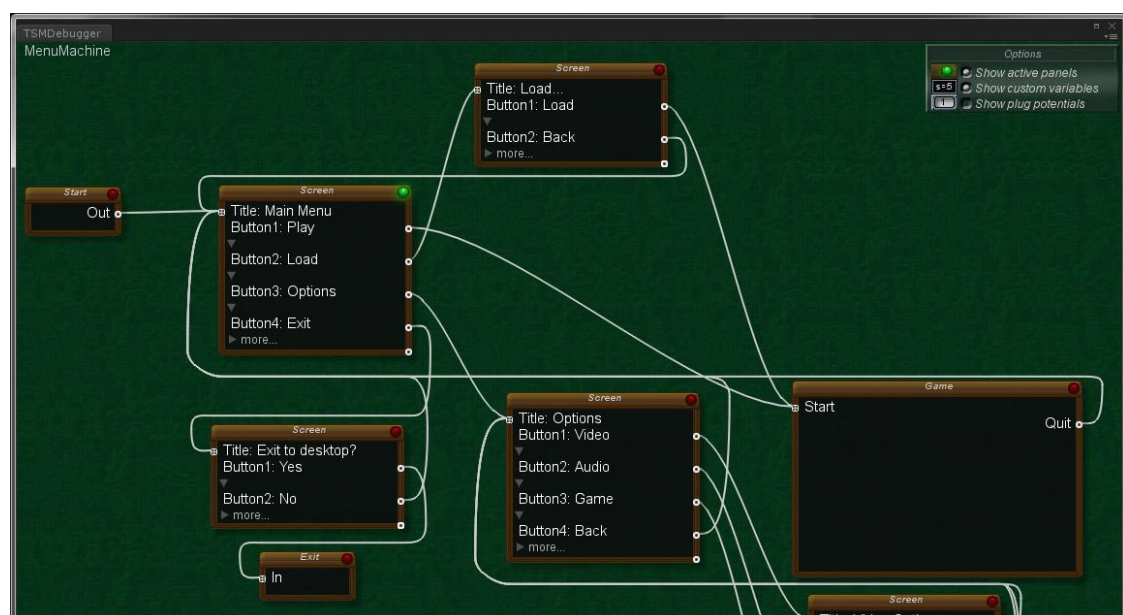
- The *AdventureMachine* in **example 4** shows a typical **state machine**. Each panel corresponds to a location, you are at one location at a given time.
- The *MenuMachine* in **example 5** shows a menu system, with panels corresponding to menu screens - another typical case of a **state machine**.
- The *GameMachine* in **example 6** is a **trigger machine** par excellence (white parts of the graph). The dialogs (blue branches) are state machines handled by Walking On The Graph. We will examine this example in detail in the next chapter.

DISPLAYING ACTIVE PANELS IN THE DEBUGGER

Load the scene "MyMenu" of Example 5. Open the TMS Debugger (menu "The Spaghetti Machine") and drag and drop the MenuMachine object from the hierarchy into the window. The MenuMachine is now linked to the window, as indicated by the name "MenuMachine" in the upper left corner of the window. As long as you don't hit "Play", the window will be empty - that's normal, as the MenuMachine doesn't yet have any graph loaded.



When you run the game, the graph will be displayed in the window. If you keep the checkbox "Show active panels" checked, each panel will have a little telltale in the upper right corner: Green if the panel is active, otherwise red. Furthermore, activation with the command "ActivateConnected" is displayed by flashing connections. When you browse through the menu, you see the panels activate accordingly. As we have a state machine, there is never more than one panel activated.



You can do the same with example 4.

On the other hand, when you display the graph in example 6, you will never see any panel activated. The reason is that this is a **trigger machine**, which means that activated panels deactivate themselves immediately. However, what you *will* see are the flashing connections indicating activation signals.

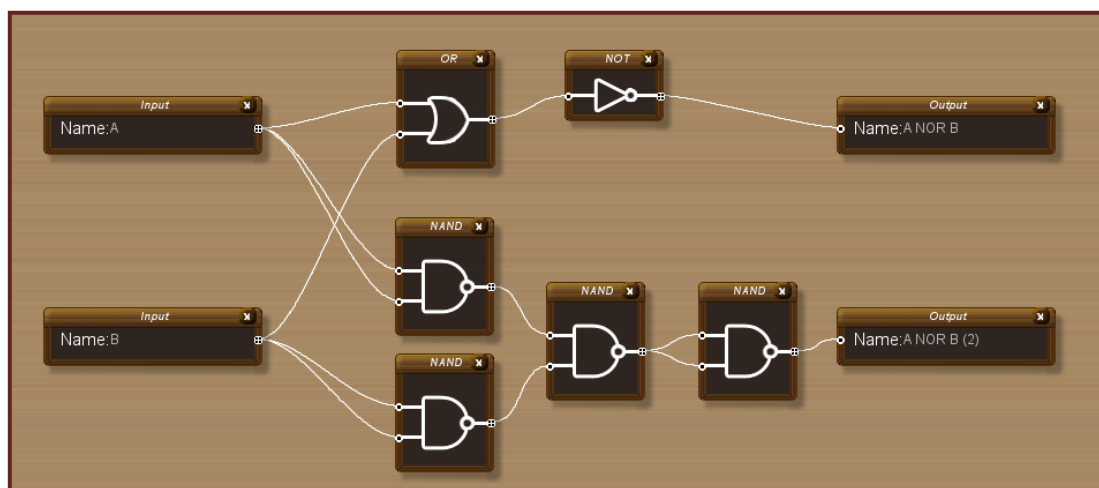
SIGNALS

The signal system allows sending signals (strings) from an output plug to all connected panels. When a panel receives a signal, a message *OnPanelReceivesSignal* is sent to the panel's master.

POTENTIALS

For some applications like logic gates or neural networks, sending single signals is not enough – you want to assign to different plugs potentials (booleans, voltage, current etc.) which are “continuously” sent to connected plugs. That's what the “potential” mechanism is for: Each slot has a float variable called “potential” whose value is automatically transmitted to all connected plugs (“downstream”, i.e. input plugs don't transmit their potential to connected output plugs). The default potential is zero.

The logic gates we have seen before are a typical case where potentials can come in useful:



Whenever you change the potential of a slot having an output (or inout) slot, the potential of all connected (non-output) plugs is changed as well and their masters receive a message *OnInputPotentialChanged*. What you usually do in this method is combining the incoming potentials (possibly using *GetSumOfInputPotentials*) to change the output potential. The new potential is then sent again to all connected plugs, and so on.

I know what you are thinking now: “What happens when I create a loop?” Well, in this case the loop is done about ten times, then the program abandons in order not to create a stack overflow (which would possibly cause a crash of Unity). So, if you want to send potentials through loops, you better don't update the output potentials in *OnInputPotentialChanged* but, say, in the Update method.

Another warning: Avoid multiplugs with multiple incoming potentials. Otherwise you may encounter inconsistent behavior: If two output plugs A1 and A2 are connected to an input plug B and you

change the potential of A1, the potential of B is set to the potential of A1, even if A2 has a different potential. The potential of A2 isn't changed as potentials are never transmitted "upstream".

EXAMPLE

The LogicMachine in example 3 (the logic gates) shows a typical application of potentials. The core part of the code is (up to some modifications for clarity) as follows:

```
public function HelloMaster( panel : Panel )
{
    UpdatePotential( panel );
}

public function OnInputPotentialChanged( panel : Panel )
{
    UpdatePotential( panel );
}

public function UpdatePotential( panel : Panel )
{
    var fOutputPotential : float = 0.0;
    switch( panel.GetPanelType() )
    {
        case "AND":
            fOutputPotential = ( panel.GetSumOfInputPotentials() >= 2.0 ) ? 1.0 : 0.0;
            break;

        case "OR":
            fOutputPotential = ( panel.GetSumOfInputPotentials() >= 1.0 ) ? 1.0 : 0.0;
            break;

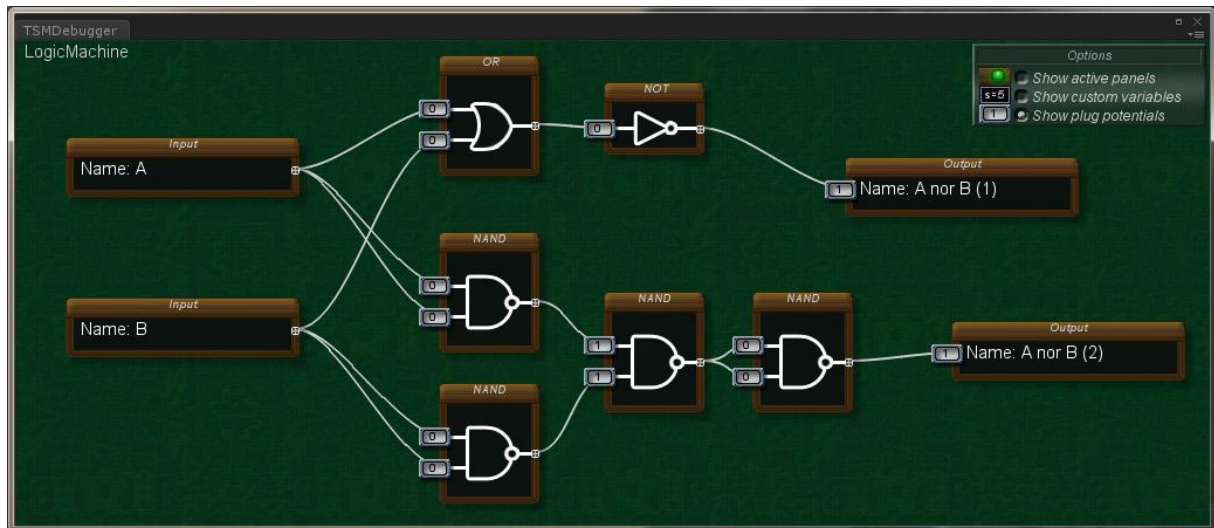
        case "NOT":
            fOutputPotential = ( panel.GetSumOfInputPotentials() == 0.0 ) ? 1.0 : 0.0;
            break;

        case "NAND":
            fOutputPotential = ( panel.GetSumOfInputPotentials() < 2.0 ) ? 1.0 : 0.0;
            break;
    }
    panel.FindSlot("Out").SetPotential( fOutputPotential );
}
```

As you see, the logic is done in UpdatePotential, a method called at initialization ("HelloMaster"), and whenever an input potential of a panel changes ("OnInputPotentialChanged").

In particular, the last line *panel.FindSlot("Out").SetPotential(fOutputPotential);* changes the potentials transmitted to the connected panels, which causes the call of UpdatePotential for the connected panels. The method is thus implicitly recursive.

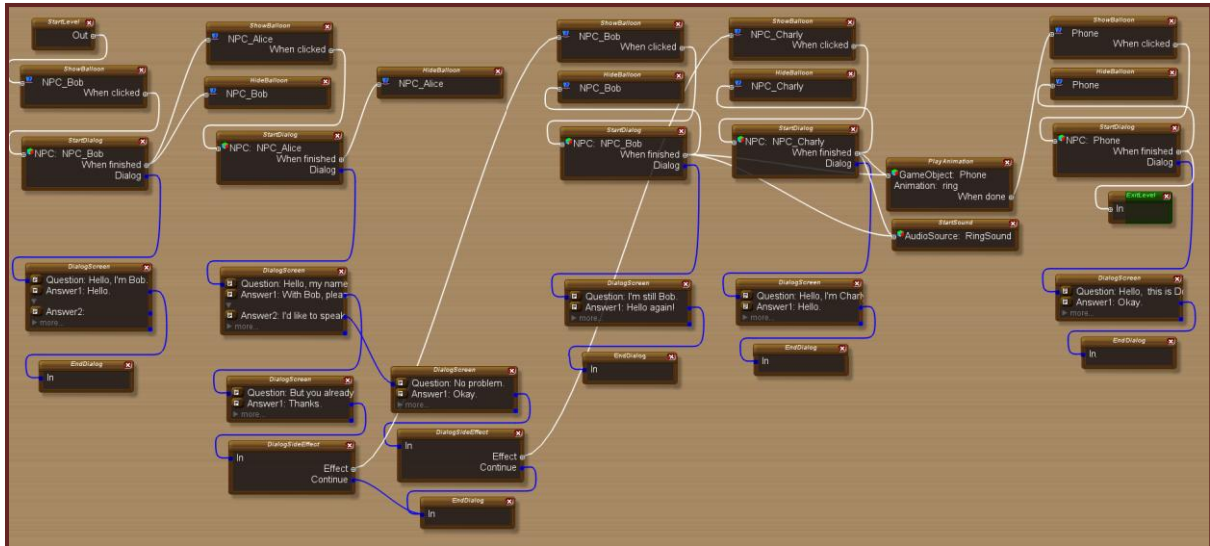
When drag and drop the LogicMachine onto the TSM Debugger window, the graph is displayed in runtime. Check the checkbox "Show Plug Potentials", and you see the potentials on all input plugs. When you change the input values in the game, you see the potentials change in real-time. Moreover, transmission of potentials is displayed as flashing connections.



VISUAL SCRIPTING WITH SPAGHETTI

A DIALOG/INTERACTION SYSTEM

When you have a look at example 6, you see not one, but two panel sets: **GameSystem_Actions** and **GameSystem_Dialogs**. When you load the example graph, you see how those panels are used.

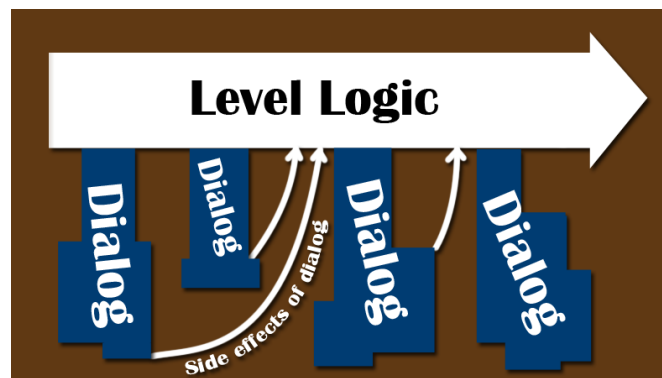


GameSystem_Actions contains some exemplary "action" panels like StartAnimation, Activate or PlaySound - panels that correspond to code snippets like `_.animation.Play(...)`, `_.SetActiveRecursively(true)` or `_.audioSource.Play()`. These panels can be interconnected with **white** connections. Note that not all those types are used in the example graph.

GameSystem_Dialogs contains dialog-related panels. The most important type is DialogScreen, each panel of this type represents one step in a dialog, linked to other steps by Answer slots. As for the other panels, StartDialog serves as link to the "action" panels, DialogScreen represents one step in a dialog, EndDialog does just that (ending the dialog), and DialogSideEffect allows triggering parts of the action part from branches of the dialog tree. The dialog panels are interconnected with **blue** lines.

A typical graph of this system consists of two parts:

- A white horizontal "trunk" of action panels, representing the level logic and roughly progressing from the left to the right;
- Blue vertical descending "branches", the actual dialog trees, with some ascending white lines for the side effects.



THE RUNTIME PART

Now let's have a look at the runtime part. Load the scene Level01 and start it. You see three characters; you can click on characters with a speech balloon to start a dialog. The dialog then makes new balloons appear (side effects) and so on. Towards the end, the phone rings and plays its "ringing" animation, corresponding to the panels "StartSound" and "PlayAnimation".

But how is this done? The scripts in use are

- NPC handles a single character; it manages display of the speech balloon and clicks on the character.
- DialogHandler handles the dialogs by Walking on the Graph. If you had a look at the menu example, it's pretty much the same.
- GameMachine: That's the most interesting part, a textbook example of a trigger machine.

Let's have a look (slightly simplified version):

```
import Spaghetti;

class GameMachine extends SpaghettiMachine
{
    var mstrGraphPath : String = "SpaghettiMachine/Examples/Example6 - Dialog and Interaction System/Diagrams/";
    var mstrGraphFile : String = "MyGame";

    function Start()
    {
        // This is a typical trigger machine (can also be set in the Inspector)
        mMachineType = MachineType.TriggerMachine;

        // Find the "StartLevel" panel and activate the connected panel
        FindPanelByType("StartLevel").GetSlot(0).ActivateConnected();
    }
}
```

That's pretty much what we already know. Now the interesting part:

```
// Called when a panel is activated which hasn't an explicit "master"
function OnPanelActivated( panel : Panel )
{
    switch( panel.GetPanelType() )
    {
        case "SetActive":
            // Panel to activate / deactivate game objects
            var bActive = panel.FindSlot("Active").GetDataBool();
            panel.FindSlot("GameObject").GetDataGameObject().SetActiveRecursively( bActive );
            break;

        case "StartSound":
            // Panel to play a sound
            panel.FindSlot("AudioSource").GetDataGameObject().GetComponent(AudioSource).Play();
            break;
    }
}
```

```

case "PlayAnimation":
    // Panel to play an animation of a game object
    strAnimation = panel.FindSlot("Animation").GetDataString();
    var gobject : GameObject = panel.FindSlot("GameObject").GetDataGameObject();
    gobject.animation.Play( strAnimation );
    yield WaitForSeconds( gobject.animation[strAnimation].length );
    panel.FindSlot("When done").ActivateConnected();
    break;

case "StartDialog":
    // Panel to start a dialog
    DialogHandler.GetInstance().StartDialog( panel );
    // The dialog handler takes care of calling panel.ActivateConnected() when the dialog is over
    break;

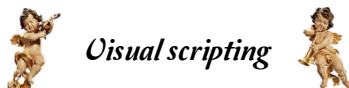
case "ExitLevel":
    // Panel to end the game
    Application.Quit();
    Debug.Break();
    break;

    // Those are just some examples.
    // Put other "commands" here.

}
}
}

```

As you can see, the code basically interprets the panels as "commands". When a panel has a white output plug, as PlayAnimation, it waits till the action is done (the *yield WaitForSeconds*) and then triggers whatever comes next. In other words (*drum roll...*) :



You might object that a visual scripting language that can only play sounds, play animations, make objects visible/invisible and start dialogs is pretty much limited. You are right. But this example is just a template for your *very own* visual scripting system - you can add whatever you want: Management of variables, conditions, loops, teleporting of objects, waiting for buttons, crazy rotations, cool particle effects, opening the web site of your favorite pasta restaurant... You name it.

A simple approach to learning what you need is to draw a flowchart what you want your script to do in a typical situation, and then transform the cases of the flowchart into panel types. If you don't need a panel to hack into the Pentagon's strategic mainframe, don't create it.

You might also add the possibility to process numbers with panels like "multiply", "add", "Mouse xy" etc., pretty much like we did with booleans in the Logic Gates example. (The excellent Strumpy Shader Editor does the same thing with 4-vectors.) You might want to use a different color for connections that transmit numbers, though.

PANEL SET DEFINITION

A panel set is defined in a XML file having the following structure:

```
<?xml version="1.0" encoding="utf-8"?>
<panelset>
  <paneltype type = "...">
    <slot label = "..." ... />
    ...
    <slot label = "..." ... />
  </paneltype>
  ...
  <paneltype type = "...">
    <slot label = "..." ... />
    ...
    <slot label = "..." ... />
  </paneltype>
</panelset >
```

You can create your own panel sets using any text editor – it's really not that difficult.

Let's have a look at the details:

<panelset>...</panelset>

Root element of the XML file.

- Attributes: None
- Child elements: One or more <paneltype>...</paneltype>

<paneltype>...</paneltype>

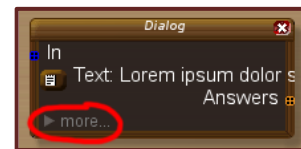
This element describes a type of panel.

- Obligatory attribute:
 - **type**: The name of the panel type, for example "AndGate".
- Optional attributes:
 - **width**: Usually panels have a width of 200 pixels. Use this attribute to set another width (in pixels).
 - **height**: Like *width*; to set a fixed custom height. (Usually the height is adapted to the number of slots.)
 - **image**: The name of a background image displayed in the panel window. The image must be placed in a folder named "resource". The size of the window is adapted to the image size (plus borders). In this case, any attributes *width* and *height* are ignored.
- Child elements: One or more <slot ... />

<slot ... />

This element describes a slot in a panel. Must be an empty-element tag ("`<slot ... />`", not "`<slot>...</slot>`").

- Obligatory attribute⁶:
 - **label**: The name of the slot. Is used to reference the slot in the panel.
- Optional attributes:
 - **plug**: The type of plug belonging to the slot, if any. Must be one of the following:
 - `"input"`: An input plug (usually on the left)
 - `"output"`: An output plug (usually on the right)
 - `"inout"`: A "bidirectional" plug which can be connected with any type of plug (on the right if not specified otherwise). Useful for example for maps.
 - **color**: The color of the plug (if any). Must be one of the following:
 - `"white"` (default color, same as omitting the color attribute)
 - `"red"`, `"green"`, `"blue"`, `"yellow"`, `"cyan"`, `"grey"`, `"gray"`⁷, `"black"`, `"magenta"`, `"orange"`, `"brown"`
 - `"octarine"` (the color of magic)⁸
 - **plug_x**, **plug_y**: Custom position of the plug relative to a background image. Both values should be between 0.0 and 1.0. The point (0.0, 0.0) is in the upper left corner of the image (without header and border). When these attributes are set, the label is *not* displayed (as it will not be besides the plug anyway). Makes only sense if the panel has a background image.
 - **multiple**: `"true"` or `"false"`; determines whether the plug is a multiplug (a plug that can be connected to more than one other plugs). If you omit this attribute, the plug will be a multiplug.
 - **content**: The type of data attached the slot, if any. Unlike the label, the content can be edited when editing the graph. Must be one of the following:
 - `"string"`: A (usually short) string
 - `"text"`: A string (usually longer). The difference with `"string"` is that `"text"` content can be edited in a separate popup window.
 - `"int"`: An integer
 - `"float"`: A floating point number
 - `"bool"`: A Boolean
 - `"more"`: Not really data, just a little arrow used to fold / unfold all following slots. Very useful to hide optional slots.
 - `"master"`: To assign to the panel a "master" object which will receive messages during runtime (for example `OnPanelActivated`).
 - `"gameobject"`: The name of a game object in the scene.
 - `"enum"`: An enum, requires the attribute `"enum"` (see below)
 - **enum**: For content type `"enum"`, the values of the enum (in one string, separated by commas, spaces being ignored). Example:



```
<slot label = "Sin" content = "enum" enum = "Lust, Gluttony, Greed, Sloth, Wrath, Envy, Pride" />
```

Internally, the value is stored as integer and can be accessed by `"GetDataInt()"` during runtime.

- **default**: The default value of the content. Optional.

⁶ The only slots that don't need a label are those with content type `"master"` and `"more"`.

⁷ Same as `"grey"`, for Americans.

⁸ We *strongly* recommend not to try any other fictional colors. If you ignore this advice, you do so at your own risk. This is not a joke. One of our beta testers once claimed he had successfully tried another fictional color invented by his favorite horror author - which is simply not possible, as such a color was never implemented. Shortly afterwards, he started to see strange things on his screen, he even claimed that his panels had become "alive"... Sadly, he is now confined to a psychiatric ward, so it's very likely that his claims about the additional color were the first symptoms of his mental illness and are not to be taken seriously. However, to be sure, you better don't try any undocumented colors. Just don't.

BASIC BUTTONS

On the left side of the editor window, you see two bars "Graph" and "Tools". The buttons on the "Graph" bar are pretty much self-explanatory:



- **New:** To clear the drawing area and start a new graph
- **Load:** To load a graph
- **Save:** To save the graph
- **Save as...:** To save the graph to another file

Note that there is no difference between "saved graphs for further editing" and "exported graphs for the game"; the graph files serve both purposes.

The "Tools" bar contains buttons useful for editing:

- **Align left/right/top/bottom:** Align the selected panels. For users who like their graphs tidy and neat.
- **Group:** To group the selected panels together. Groups can be nested.
- **Ungroup:** To break a group - only if one group (and no other panels) is selected
- **Search/Replace:** To search a string in the selected (or all) panels, and possibly replace it⁹. The panels (or groups of panels) which contain the string in one of the content fields are selected.

PANELSETS

The button **Load panelset** serves (who might have guessed it?) to load a panelset defined in an xml file, as described above. Once a panelset is loaded, it is displayed as an additional window with one button per panel type. Note that

1. You can load more than one panelsets
2. When you load a panelset with the same name as one that has already been loaded, the existing panelset window is replaced rather than adding a new window. This is useful if you modify a panelset xml during edition, for example to add new panel types.
3. However, when you update a panelset, graph nodes already created from this panelsets are *not* updated.

⁹ This function only affects string content (content type *string*, *text*) and game object names (content type *master*, *gameobject*), not numeric and boolean content or tags.

EDITING A GRAPH

- Left click on a plug : Start drawing a connection
- While drawing a connection:
 - Left click again on another plug: Link connection to that second plug
 - Left click not on a plug: Abandon drawing
 - If you left click on another plug and the drawing is abandoned, chances are that it wasn't the right plug: Either the color is wrong, or it isn't a multiplug and it has already a connection, or you tried to connect two input links, or two output links.
- Right click on a plug: Delete all connections of this plug
- Draw mouse with left mouse button hold down on empty area: Selection box
- Draw mouse with left mouse button hold down on panel: Move all selected panels
- Draw mouse with right mouse button hold down: Scroll all panels (a minimap is displayed at the center)
- Ctrl-D : Duplicate selected panels
- Del: Delete selected panels

TIP: ORDER OF CONNECTIONS

When you link several panels to the same multiplug, you sometimes want to have them stored in a certain order. (Example: A "Menu" panel with attached "Button" panels).

That's very simple to achieve: The panels linked to the same plug are *always* stored in "top-down" order, i.e. according to their vertical position. So just put the panels one below another in the right order, and in the game you will have this very order in the plug's connections list.

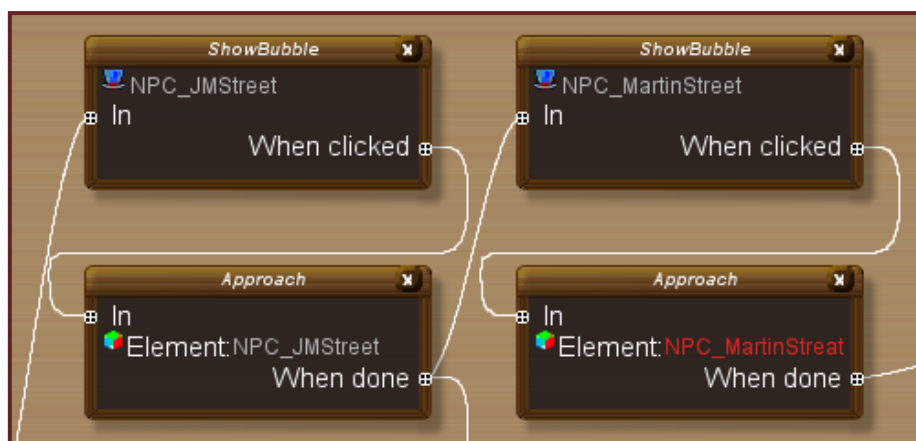
ATTACHING PANELS TO GAME OBJECTS

Each panel is attached to a game object called the panel's "master". In runtime, this master will receive messages related to the panel, such as "OnPanelActivated". By default, the master is the SpaghettiMachine object.

To allow a panel to have another master, create a slot with content type "master". In the editor, the slot is marked with a "top hat" icon 🎩 and contains text field where you can enter the name of the master object. The name is colored white when a game object with this name actually exists in the current scene, otherwise it's red.

For multiple panels attached to a "manager" object, the attribute "default" can come in quite useful. When you want, for example, that all your panels of type "Dialog" have an object called "DialogManager" as master, just add the attribute *default = "DialogManager"*.

Note that unlike other types of slots, the "master" slot doesn't require a label as you usually don't need to access explicitly to the slot. The runtime classes take care of identifying and assigning the panel's master (if any) and sending him messages whenever something interesting happens. Details see next paragraph.



When you want a slot to refer to a game object, but without making it the slot's master, use the content type "gameobject". Those slots are marked with the Unity game object icon 🎲. As for the master, the editor verifies at runtime whether the game object exists in the current scene; if it doesn't, the name is colored red.

∞ RUNTIME CLASS REFERENCE ∞

All runtime classes are in the namespace *Spaghetti*.

THE CLASS SPAGHETTIMACHINE AND ITS MEMBERS

SpaghettiMachine

Inherits from MonoBehaviour

Class to load, access and manage one graph. You usually derive a class from SpaghettiMachine, maybe add some code of your own and attach it to a GameObject.

Variables		See page
mMachineType	Indicates the behavior of the panels" "Activate" mechanism	34

Functions		
LoadFromFile	Load the graph from a local .bytes file	34
LoadFromResources	Load the graph from a .bytes file in a resources folder	34
LoadFromURL	Load the graph from a .bytes file via the net	34
LoadFromTextAsset	Load the graph from a text asset	35
LoadFromString	Load the graph from a string	35
GetNumberOfPanels	Says how many panels the machine consists of	35
GetPanel	Access a panel by index	35
GetPanels	Get an array with all panels	36
FindPanelByType	Find a panel of a certain type	36

Received messages		
OnGraphLoaded	Called when the graph is loaded (from file, string or URL)	36

```
import Spaghetti;

public class MyGameMachine extends SpaghettiMachine
{
    public function Start()
    {
        LoadFromFile( "MyGraphs/Gamegraph" );
        FindPanelByType( "Start" ).FindSlot("Out").ActivateConnected();
    }

    // This is called when a panel is activated (except when you have assigned it another master)
    function OnPanelActivated( panel : Panel )
    {
        if( panel.GetPanelType() == "QuitGame" )
        {
            Application.Quit();
        }
    }
}
```

```
}  
}
```

SpaghettiMachine. mMachineType

var mMachineType : MachineType

Determines how the panel activation mechanism works. Can have the following values:

- MachineType.StateMachine : Only zero or one panels can be active at a time
- MachineType.MultistateMachine : Multiple states can be active at a time
- MachineType.TriggerMachine : Activation lasts only the time to send *OnPanelActivated*

Details see "Panel Activation", page 20.

SpaghettiMachine. LoadFromFile

function LoadFromFile(strLocalFilePath : String) : void

Loads a graph from a .bytes file from *Application.dataPath + "/" + strLocalFilePath*.

SpaghettiMachine. LoadFromResources

function LoadFromResources(strPathInResourceFolder : String) : void

Loads a graph from a .bytes file from *strPathInResourceFolder*. The path is relative to any Resources folder inside the Assets folder of your project.

SpaghettiMachine. LoadFromURL

function LoadFromURL (strURL: String) : void

Loads a graph from a .bytes file from the indicated URL. Note that the file is loaded in a coroutine - unlike *LoadFromFile*, the code calling *LoadFromURL* continues without waiting for the file being loaded.

When done, the message *OnGraphLoaded* is sent to the SpaghettiMachine object.

SpaghettiMachine. LoadFromTextAsset



function LoadFromTextAsset (textAsset : TextAsset) : void

Loads a graph from a TextAsset.

When done, the message OnGraphLoaded is sent to the SpaghettiMachine object.

```
LoadFromTextAsset ( myTextAsset );
```

SpaghettiMachine. LoadFromString

function LoadFromString (strContent: String) : void

Loads a graph from a string containing what is usually in the .bytes files. This is useful as "generic" method, for example when your graph data are in a TextAsset.

When done, the message OnGraphLoaded is sent to the SpaghettiMachine object.

```
LoadFromString ( myTextAsset.text );
```

This *should* make it possible to use the Spaghetti Machine on mobile platforms¹⁰.

SpaghettiMachine. GetNumberOfPanels

function GetNumberOfPanels () : int

Return the number of panels in the graph.

SpaghettiMachine. GetPanel

function GetPanel (i : int) : Panel

¹⁰ I wasn't able to test this, though. If the Spaghetti Machine still doesn't work on mobile devices, feel free to contact me (I'm "Zogg" on the Unity forum).

Returns the i-th panel in the graph.

SpaghettiMachine. GetPanels

```
function GetPanels ( ) : Panel[]
```

Returns all panels as array.

SpaghettiMachine. FindPanelByType

```
function FindPanelByType ( strType: String ) : Panel
```

Returns a panel with the type *strType*, or null if there is none. Use this when you know there is only one panel of this type (e.g. a "Start" panel), or when you don't care which panel of this type you get.

SpaghettiMachine. OnGraphLoaded

```
function OnGraphLoaded() : void
```

Is called when the graph is loaded. When loading a graph from an URL, use this for initialization of the graph, for example activating a "Start" panel.

THE CLASS PANEL AND ITS MEMBERS

Panel

The Panel class implements the panels of a graph. The panel objects are created when loading a SpaghettiMachine.

Functions		See page
GetPanelType	Type of panel (a string)	37
GetSlots	Get all slots as array	37
GetInputSlots	Get all input slots as array	38
GetOutputSlots	Get all output slots as array	38
GetNumberOfSlots	Gets the total number of slots	38
GetNumberOfInputSlots	Gets the total number of input slots	38
GetNumberOfOutputSlots	Gets the total number of output slots	38
GetSlot	Access a slot by index	38
FindSlot	Find a slot by label	39
FindInputSlot	Find an input slot by label	39
FindOutputSlot	Find an output slot by label	39
GetMaster	Access the master object of a panel	39
Activate	Activate a panel (example: State machine)	39
Deactivate	Deactivate a panel	40
GetInputPotentials	Get an array containing the input potentials	40
GetSumOfInputPotentials	Get the sum of the input potentials	40
SetVariable	Assigns a value to a custom runtime variable	41
GetVariable	Gets the value of a custom runtime variable	41

Panel. GetPanelType

```
function GetPanelType() : String
```

Returns the type of the panel, for example "StartDialog".

Panel. GetSlots

```
function GetSlots() : Slot[]
```

Returns the panel's slots as array (from top to bottom).

Panel. GetInputSlots

function GetInputSlots() : Slot[]

Returns the panel's input slots as array (from top to bottom).

Panel. GetOutputSlots

function GetOutputSlots() : Slot[]

Returns the panel's output slots as array (from top to bottom).

Panel. GetNumberOfSlots

function GetNumberOfSlots() : int

Returns the total number of slots of the panel.

Panel. GetNumberOfInputSlots

function GetNumberOfInputSlots() : int

Returns the number of input slots of the panel.

Panel. GetNumberOfOutputSlots

function GetNumberOfOutputSlots() : int

Returns the number of output slots of the panel.

Panel. GetSlot

function GetSlot(i : int) : Slot

Returns the slot corresponding to a certain index, starting with 0.

Panel. FindSlot

function FindSlot(strLabel : String) : Slot

Finds a slot by its label. This is the usual method to access the slots of a panel.

```
...  
panel = panel.FindSlot("NextPanel").GetConnectedSlot().GetPanel();  
...
```

Panel. FindInputSlot

function FindInputSlot(strLabel : String) : Slot

Like FindSlot, but limited to input slots.

Panel. FindOutputSlot

function FindOutputSlot(strLabel : String) : Slot

Like FindSlot, but limited to output slots.

Panel. GetMaster

function GetMaster() : GameObject

Return the panels' master object. This is usually the SpaghettiMachine, unless the panel has a "master" slot specifying another master.

Panel. Activate

function Activate() : void

Sets the panel's "active" flag. The message "OnPanelActivated" is sent to the panel's master. If the SpaghettiMachine's machine type is MachineType.StateMachine, this also deactivates any other active panel (and sends a message "OnPanelDeactivated").

Panel. Deactivate

function Deactivate() : void

Clears the panel's "active" flag. The message "OnPanelDeactivated" is sent to the panel's master. Note that this doesn't make the panel virtually "disappear", as when deactivating game objects.

Panel. IsActive

function IsActive() : boolean

Says whether or not the panel's "active" flag is set.

Panel. GetInputPotentials

function GetInputPotentials() : float[]

Returns an array containing the panel's input potentials. Every input slot corresponds to one entry (even if you never set their potentials).

Panel. GetSumOfInputPotentials

function GetSumOfInputPotentials() : float

Returns the sum of the panel's input potentials. Useful, e.g., for logic gates.

Panel. SetVariable

```
function SetVariable( strName : String, value : object )
```

A variable is a pair name/value which can be attached to the panel at runtime.

SetVariable assigns a value to a variable. The value can be anything derived from System.object. Internally, variables are stored in .Net hashtables.

```
import Spaghetti;

// Panel initialization
function HelloMaster( panel : Panel )
{
    // Initialize variable "visits_counter"
    panel.SetVariable( "visits_counter", 0 );
}

// Panel has been activated
function OnPanelActivated( panel : Panel )
{
    // Read, increment and write variable "visits_counter"
    var iVisits : int = panel.GetVariable( "visits_counter" );
    iVisits++;
    panel.SetVariable( "visits_counter", iVisits );

    Debug.Log("Panel with type "+panel.GetPanelType()+" has been visited "+iVisits+" times;");
}
```

Panel. GetVariable

```
function GetVariable( strName : String ) : object
```

A variable is a pair name/value which can be attached to the panel at runtime.

GetVariable gets the value of a variable, or null if the variable has never been set. See Panel.SetVariable for an example.

THE CLASS SLOT AND ITS MEMBERS

Slot

The Panel class implements the panels of a graph. The panel objects are created when loading a SpaghettiMachine.

Functions		See page
GetPanel	Returns the panel the slot belongs to	42
GetNumberOfConnectedSlots	Returns the number of slots this slot is connected to	43
GetConnectedSlot	Returns a connected slot	43
GetConnectedSlots	Returns all connected slots as array	43
GetConnectedPanel	Returns a connected panel	44
GetLabel	Returns the slot's label	44
GetPlugType	Returns the slot's plug type, if any	44
GetContentType	Returns the slot's content type, if any	44
GetColorName	Returns the name of the slot's color	45
GetSlotIndex	Returns the slot's position (uppermost = 0 etc.)	45
Below	Returns the slot below this slot, if any	45
Above	Returns the slot above this slot, if any	45
GetDataString	Returns the slot's string content, if any	45
GetDataFloat	Returns the slot's float content, if any	45
GetDataInt	Returns the slot's int content, if any	46
GetDataBool	Returns the slot's boolean content, if any	46
GetDataGameObject	Returns the slot's GameObject reference, if any	46
ActivateConnected	Activates all connected panels	46
SendSignalToConnected	Sends a signal to all connected panels	46
GetPotential	Returns the slot's potential value	46
SetPotential	Sets the potential value of the slot (and connected slots)	47
SetVariable	Assigns a value to a custom runtime variable	47
GetVariable	Gets the value of a custom runtime variable	47

Slot.GetPanel

function GetPanel () : Panel

Returns the panel the slot belongs to. Useful when "walking through the graph".

```
...  
panel = panel.FindSlot("NextPanel").GetConnectedSlot().GetPanel();  
...
```

Slot.GetNumberOfConnectedSlots

function GetNumberOfConnectedSlots () : int

Returns the number of slots this slot is connected to.

Slot.GetConnectedSlot

function GetConnectedSlot(i : int) : Slot

Returns the i^{th} connected slot, if any. If the index is out of range it simply returns null.

function GetConnectedSlot() : Slot

Same as GetConnectedSlot(0), somewhat more elegant for non-multiplugs. Null if no slot connected.

```
...  
panel = panel.FindSlot("NextPanel").GetConnectedSlot().GetPanel();  
...
```

Slot.GetConnectedSlots

function GetConnectedSlots() : Slot[]

Returns all connected slots as array. The list is ordered according to the vertical position of the connected panels (starting on top). This allows a "menu" panel

Slot.GetConnectedPanel

function GetConnectedPanel(i : int) : Panel

Returns the i^{th} connected panel.

If i is within range, GetConnectedPanel(i) is just a shortcut for GetConnectedSlot(i).GetPanel(). If not, GetConnectedPanel(i) returns zero whereas GetConnectedSlot(i).GetPanel() causes an error.

function GetConnectedPanel () : Panel

Same as GetConnectedPanel (0), somewhat more elegant for non-multiplugs. Null if no panel connected.

```
...
panel = panel.FindSlot("NextPanel").GetConnectedPanel();
if( panel != null )
{
    ...
}
```

Slot.GetLabel

function GetLabel() : String

Returns the slot's label.

Slot.GetPlugType

function GetPlugType() : PlugType

Returns the slot's plug type (possibly PlugType.None)

Slot.GetContentType

function GetContentType() : ContentType

Returns the slot's content type (possibly ContentType.None)

Slot.GetColorName

function GetColorName() : String

Returns the name of the plug's color, e.g. "red".

Slot.GetSlotIndex

function GetSlotIndex() : int

Returns the slot's position in the panel. The uppermost slot has the position 0, the next one 1 etc.

Slot.Below

function Below() : Slot

Returns the slot below this slot (or null if there is none).

Slot.Above

function Above() : Slot

Returns the slot above this slot (or null if there is none).

Slot.GetDataString

function GetDataString() : String

Returns the slot's string content, if any. For content types String and Text.
(For content type Enum you use GetDataInt().)

Slot.GetDataFloat

function GetDataFloat() : float

Returns the slot's float content, if any. For content type Float only.

Slot.GetDataInt

function GetDataInt() : int

Returns the slot's integer content, if any. For content types Int **and** Enum.

Slot.GetDataBool

function GetDataBool() : boolean

Returns the slot's boolean content, if any. For content type Bool only.

Slot.GetDataGameObject

function GetDataGameObject() : GameObject

Returns the slot's GameObject reference, if any. For content type GameObject only.

Slot.ActivateConnected

function ActivateConnected()

Set the "active" flag for all panels this slot is connected to, and send appropriate OnPanelActivated messages.

Slot.SendSignalToConnected

function SendSignalToConnected(strSignal : String)

Sends a signal to all connected strings, which means sending OnPanelReceivesSignal messages to their masters.

Slot.GetPotential

function GetPotential() : float

Returns the slot's potential. Default value is 0.0.

Slot.SetPotential

```
function SetPotential ( fValue : float )
```

Sets the potential of the slot and of all connected slots. The latter receive OnInputPotentialChanged messages.

Note that signals can't be sent "upstream", i.e. from input slots and/or to output slots.

Slot.SetVariable

```
function SetVariable( strName : String, value : object )
```

A variable is a pair name/value which can be attached to the slot at runtime.

SetVariable assigns a value to a variable. The value can be anything derived from System.object. Internally, variables are stored in .Net hashtables.

```
slot.SetVariable( "enabled", true );
```

Slot.GetVariable

```
function GetVariable( strName : String ) : object
```

A variable is a pair name/value which can be attached to the panel at runtime.

GetVariable gets the value of a variable, or null if the variable has never been set.

```
if( slot.GetVariable( "enabled" ) == true )  
{  
    slot.ActivateConnected();  
}
```

AUXILIARY TYPES

Auxiliary enums

PlugType	Indicates the type of plug, if any	48
ContentType	Indicates the type of data content, if any	49

Auxiliary class

Signal	Contains a sender panel, a receiver panel, and a message string	49
--------	---	--------------------

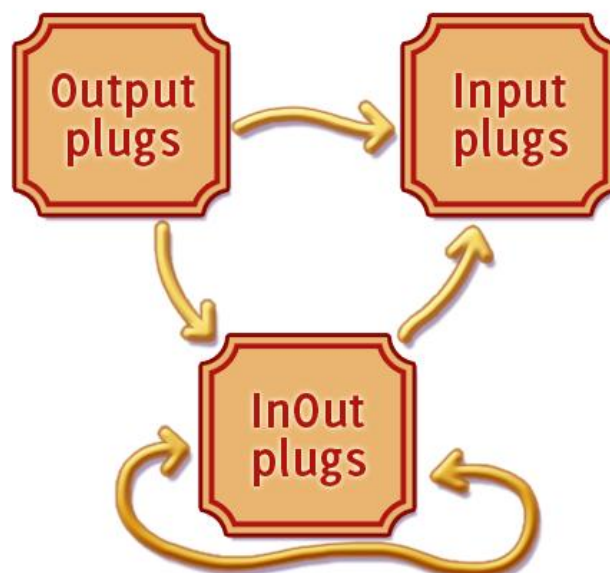
PlugType

enum

Determines what kind of plug the slot has, if any.

Values

PlugType.None	The slot has no plug
PlugType. Output	Output plug. Can be connected to Input and InOut.
PlugType. Input	Input plug. Can be connected to Output and InOut.
PlugType. InOut	Bidirectional plug. Can be connected to all three types, but will usually be plugged to other InOut plugs. This is useful for undirected graphs, for example adventure maps (see Example 4)



Connection of plug types, and propagation direction of signals and potentials.

ContentType

enum

Determines what kind of data the slot contains, if any.

Values

ContentType.None	The slot has no content
ContentType.String	The slot contains a string (used for rather short strings such as identifiers)
ContentType.Text	The slot contains a string (used for rather long strings such as dialog text)
ContentType.Float	The slot contains a floating point number
ContentType.Int	The slot contains an integer number
ContentType.Bool	The slot contains a boolean
ContentType.Master	The slot defines the panel's master,
ContentType.GameObject	The slot contains a reference to a game object
ContentType.Enum	The slot contains an enum
ContentType.More	The slot contains, well, actually nothing – it's just a button to fold/unfold all following slots in the editor.

Signal

class

An auxiliary class for "signal" messages, used in OnPanelReceivesSignal.

Values

mSender	The Panel the signal came from
mReceiver	The Panel the signal is sent to
mstrMessage	The string containing the actual message

MESSAGES SENT TO THE MASTER

Received messages

HelloMaster	Sent to the master after initializing the panel	50
OnPanelActivated	Sent when the panel's "active" flag is set	50
OnPanelDeactivated	Sent when the panel's "active" flag is cleared	50
OnInputPotentialChanged	Sent when the panel's input potentials change	51
OnPanelReceivesSignal	Sent when the panel's "active" flag is cleared	51

HelloMaster

function HelloMaster(panel : Panel)

Sent to the master of each panel after initializing the panels (when the graph has been read from file or URL, before sending OnGraphLoaded). Useful for initialization purposes.

```
import Spaghetti;

function HelloMaster ( panel : Panel )
{
    if( panel.GetPanelType() == "Start" )
    {
        panel.FindSlot("Out").ActivateConnected();
    }
}
```

OnPanelActivated

function OnPanelActivated (panel : Panel)

Sent to the master when the panel's "active" flag is set, for example by Panel.Activate() or by Slot.ActivateConnected().

OnPanelDeactivated

function OnPanelDeactivated(panel : Panel)

Sent to the master when the panel's "active" flag is cleared. Exception: When the spaghetti machine is a trigger machine, this message is never sent.

OnInputPotentialChanged

function OnInputPotentialChanged (panel : Panel)

Sent to the master whenever one of the panel's input potentials is changed.

```
import Spaghetti;

function OnInputPotentialChanged ( panel : Panel )
{
    switch( panel.GetPanelType() )
    {
        case "AndGate":
            panel.FindSlot("Out"). SetPotential( (panel. GetSumOfInputPotentials() == panel.GetNumberOfInputSlots() ) ? 1.0 :
0.0 );
            break;

        case "OrGate":
            panel.FindSlot("Out"). SetPotential( (panel. GetSumOfInputPotentials() > 0.0 ) ? 1.0 : 0.0 );
            break;

        case "NotGate":
            panel.FindSlot("Out"). SetPotential( (panel. GetSumOfInputPotentials() > 0.0 ) ? 0.0 : 1.0 );
            break;

        case "NandGate":
            panel.FindSlot("Out"). SetPotential( (panel. GetSumOfInputPotentials() == panel.GetNumberOfInputSlots() ) ? 0.0 :
1.0 );
            break;

        case "NorGate":
            panel.FindSlot("Out"). SetPotential( (panel. GetSumOfInputPotentials() > 0.0 ) ? 0.0 : 1.0 );
            break;

        case "XorGate":
            panel.FindSlot("Out"). SetPotential( (panel. GetSumOfInputPotentials() == 1.0 ) ? 1.0 : 0.0 );
            break;
    }
}
```

OnPanelReceivesSignal

function OnPanelReceivesSignal (sender : Panel, receiver : panel, signal : Signal)

Sent to the receiver's master whenever a message is sent with Slot.SendSignalToConnected.

```
import Spaghetti;

function OnPanelReceivesSignal ( signal : Signal )
{
    Debug.Log("Signal received.");
    Debug.Log("Sender: " + signal.mSender.GetPanelType() );
    Debug.Log("Receiver: " + signal. mReceiver.GetPanelType() );
    Debug.Log("Message: " + signal. mstrMessage );
}
```

THE EXAMPLES

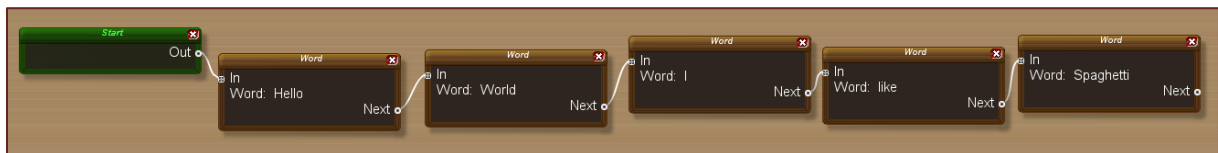
This part describes the examples included in the package (in the SpaghettiMachine/Examples folder). Some of those have been discussed in this manual, others not. The reader is invited to take a closer look at the examples, as they show common techniques and may as well serve as templates for your projects.

However, given the generic nature of the Spaghetti Machine, maybe you will discover ways of doing things even more efficiently or elegantly than we did ?

EXAMPLE 1 - HELLO WORLD

PURPOSE

This example shows a very basic panel set. It allows editing a chain of words.



PANEL SETS

Words: Contains two simple panel types "Start" and "Word".

RUNTIME SCRIPTS

GreetingMachine: A basic *SpaghettiMachine* which prints the words to the console.

DEMONSTRATED CONCEPTS

- **EDITOR**: Basic notions, graph edition,
- **RUNTIME**: Basic notions, Walking On The Graph.

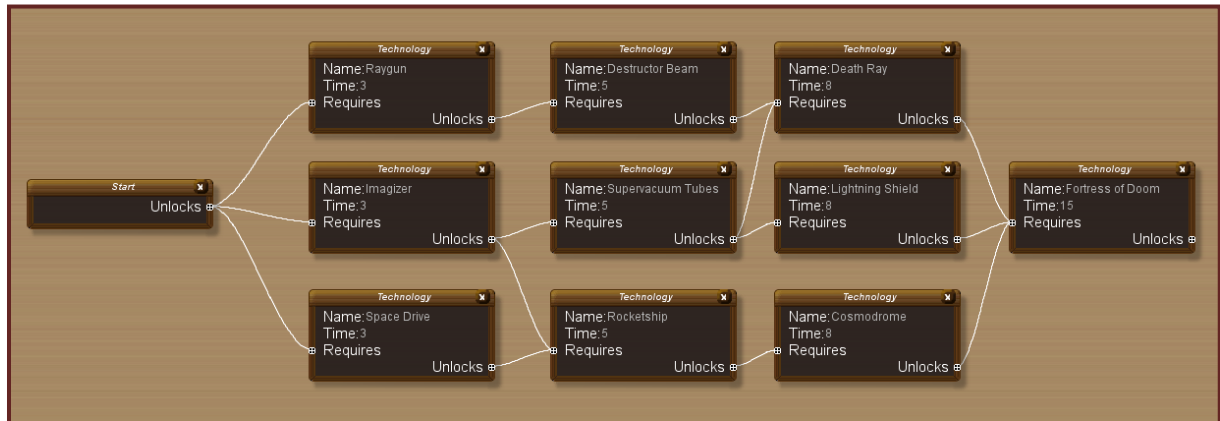
PROJECTS THAT MAY REQUIRE A SIMILAR APPROACH

Linear campaign

EXAMPLE 2 - TECH TREE

PURPOSE

This example shows how to implement a tech tree system.



PANEL SETS

TechTreeSet: Contains two simple panel types "Start" and "Technology".

RUNTIME SCRIPTS

TechTreeMachine: A *SpaghettiMachine* which shows how to use the tech tree graph in a game.

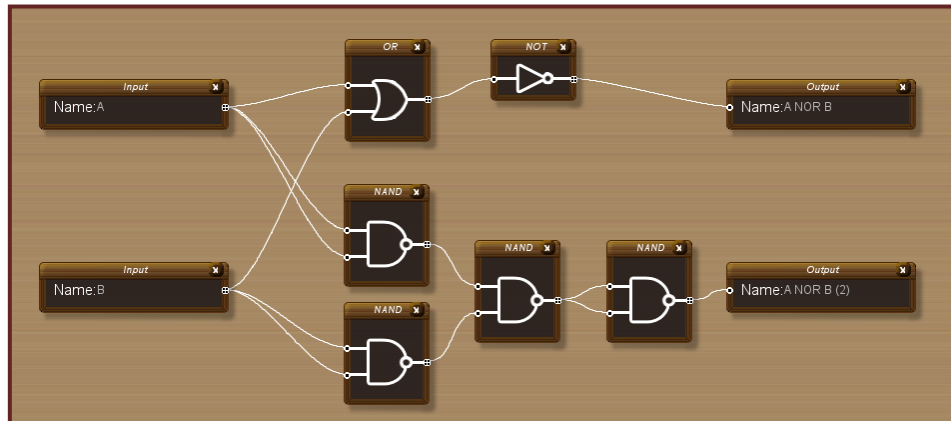
DEMONSTRATED CONCEPTS

- **EDITOR**: Basic notions, graph edition,
- **RUNTIME**: Custom variables, Walking On The Graph

EXAMPLE 3 - LOGIC GATES

PURPOSE

This example shows how to implement logic gates and test them.



PANEL SETS

LogicGates: Contains input and output panels, as well as four types of logic gates.

RUNTIME SCRIPTS

LogicMachine: Lets you set the input values and observe the outputs in real-time.

DEMONSTRATED CONCEPTS

- **EDITOR**: Panels with images
- **RUNTIME**: Potentials

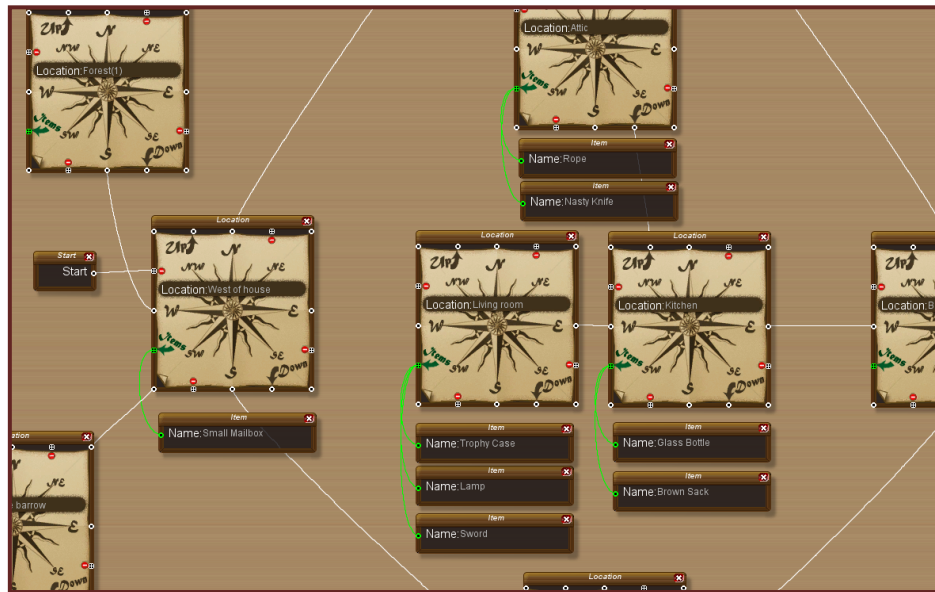
PROJECTS THAT MAY REQUIRE A SIMILAR APPROACH

Electric/electronic circuits, neural networks

EXAMPLE 4 - TEXT ADVENTURE MAP

PURPOSE

This example shows how to implement a text adventure map, or, more generally, a map composed of separate locations linked by gates.



PANEL SETS

TextAdventureMap: Contains panels for locations and items.

RUNTIME SCRIPTS

AdventureMachine: Lets you "explore" the map using the commands W, NW, N, NE, E, SE, S, SW, Up, Down and Look.

DEMONSTRATED CONCEPTS

- **EDITOR**: Panels with images, colored links
- **RUNTIME**: Panel activation, State machine

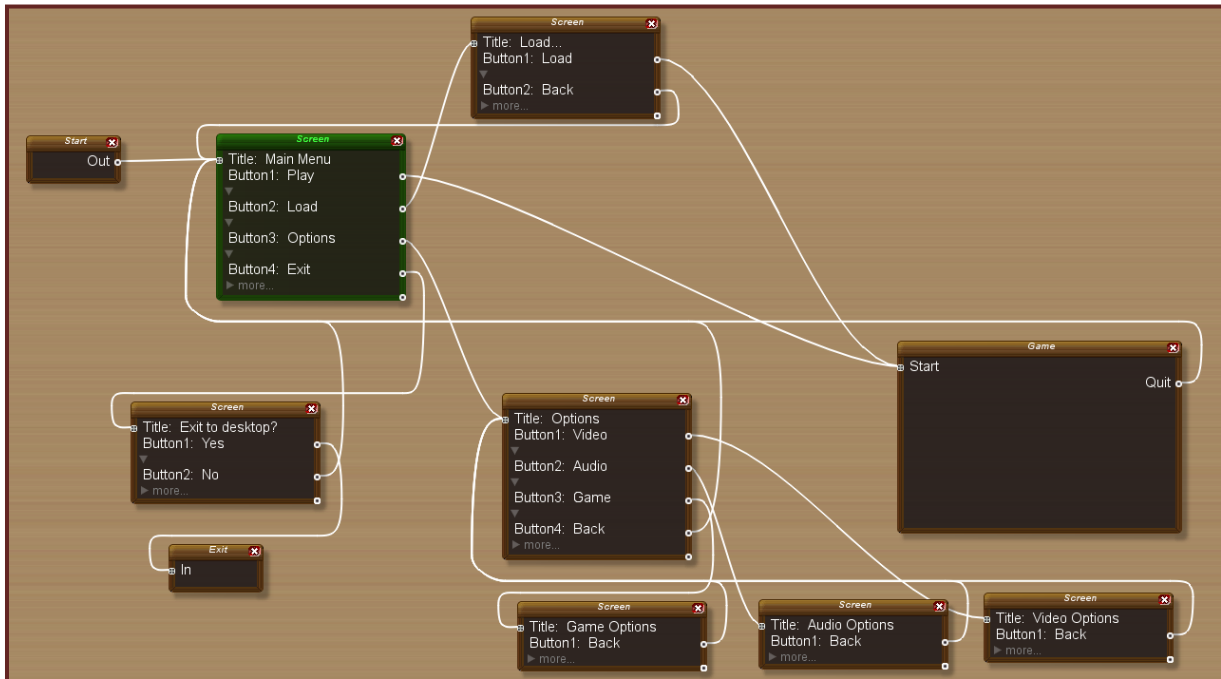
PROJECTS THAT MAY REQUIRE A SIMILAR APPROACH

Graphical adventure map, star systems linked by jump gates, nonlinear campaign

EXAMPLE 5 - MENU SYSTEM

PURPOSE

This example shows how to implement a menu system.



PANEL SETS

MenuSystem: Contains the Screen panel, as well as Start, Exit and Game panels.

RUNTIME SCRIPTS

MenuMachine: Simulates the navigation in the menus.

DEMONSTRATED CONCEPTS

- **EDITOR**: Foldable panel parts ("> more")
- **RUNTIME**: Panel activation, State machine

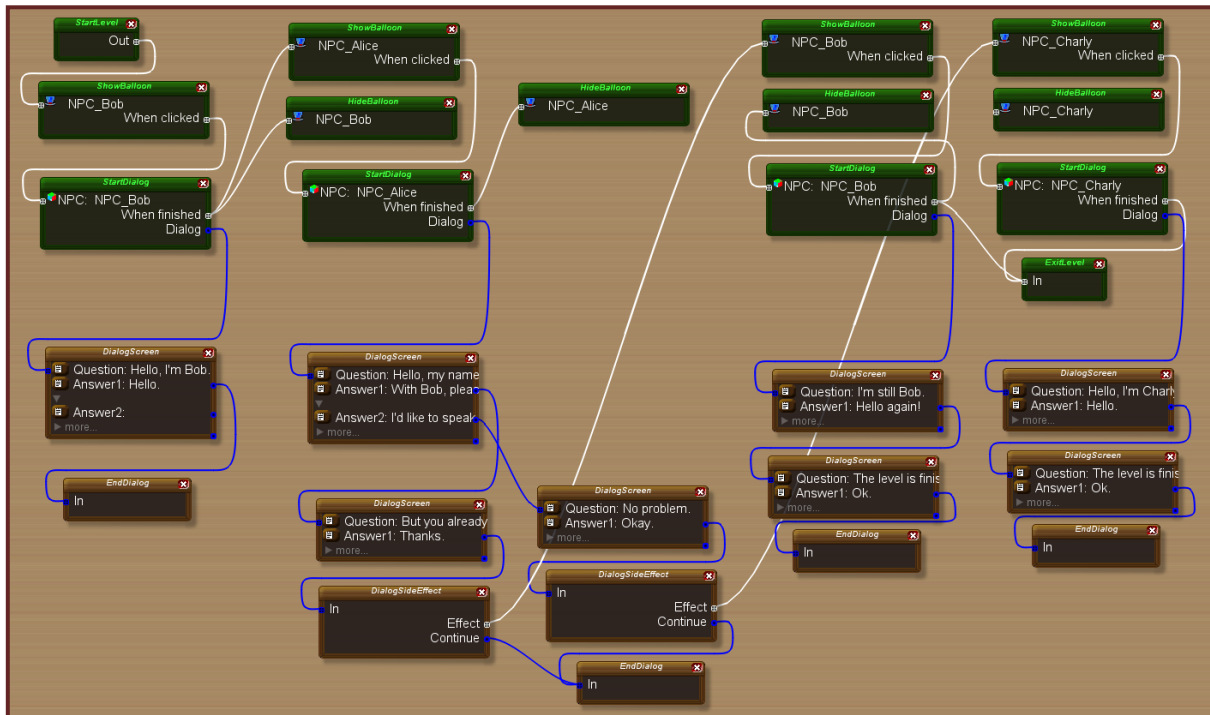
PROJECTS THAT MAY REQUIRE A SIMILAR APPROACH

Dialog systems (cf. example 6)

EXAMPLE 6 - DIALOG AND INTERACTION SYSTEM

PURPOSE

This example shows a system to handle dialog trees and interactions. It contains a simple visual scripting system which can easily be extended. Details see page 25.



PANEL SETS

GameSystem_Actions: Panels that correspond to actions.

GameSystem_Dialogs: Panels that belong to the dialog system.

RUNTIME SCRIPTS

NPC: Attached to a character (or object), manages the speech balloon, user clicks on the character etc.

DialogHandler: Singleton that manages dialogs, pretty much like the *MenuMachine*

GameMachine: Very basic visual scripting system, easily expandible.

DEMONSTRATED CONCEPTS

- **EDITOR**: Multiple colors
- **RUNTIME**: Trigger machine, visual scripting

PROJECTS THAT MAY REQUIRE A SIMILAR APPROACH

Anything that requires some sort of visual scripting

❧ INDEX ❧


activation.....	20	Order of links	14, 33
Align bottom	32	Panel (class)	39
Align left	32	panel set	7
Align right	32	definition in XML.....	29
Align top.....	32	Panels	6
attaching panels to game objects	34	plugs.....	6
Auxiliary types.....	50	PlugType (enum)	50
colors	7	Potentials.....	22
content.....	6	Replace	32
ContentType (enum)	51	Save.....	32
First Steps	4	Save as.....	32
graph	2	Search	32
Graph Editor	32	Signal (class).....	51
Group.....	32	Signals.....	22
Installation	4	Slot (class)	44
label.....	6	slots.....	6
Load.....	32	Spaghetti (namespace)	35
Load panelset.....	32	SpaghettiMachine	20
master.....	34	SpaghettiMachine (class).....	35
Messages sent to the Master	52	type	6
multiplug.....	7	Ungroup.....	32
order of links	33	Visual Scripting	25
namespace Spaghetti	35	workflow	8
New	32		

❧ TABLE OF CONTENT ❧

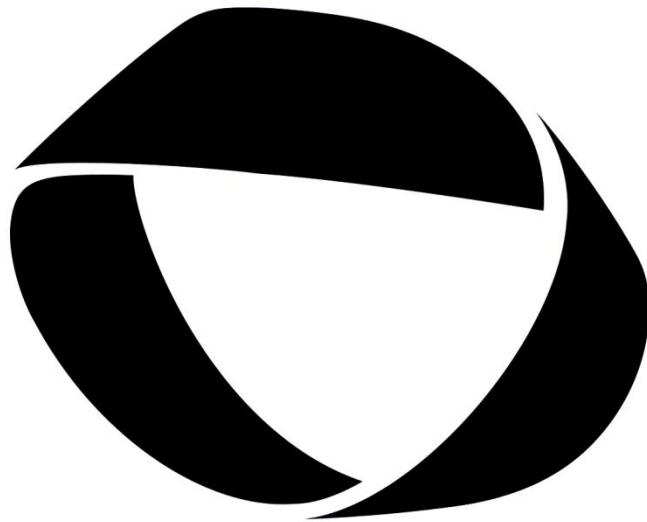
Introduction ❧.....	2
What is the Spaghetti machine?.....	2
What do those graphs look like?	3
Getting Started ❧.....	4
Installation	4
Example 1: First Steps.....	4
Basic Notions ❧	6
Panels, slots and plugs.....	6
Type.....	6
Slots.....	6
Label	6
Plugs.....	6
Content.....	6
Colors.....	7
Multiplugs.....	7
Panel Sets.....	7
The Workflow	8
An Example ❧.....	9
Example 2: Tech Tree Management.....	9
Creating the panel set.....	9
Editing the graph	11
Importing the graph	11
Using the graph.....	12
The Debugging Window	13
Tip: Walking on the Graph.....	14
Advanced custom panels ❧	15
Example 3: Logic Gates	15
Example 4: Text Adventure Map	17
The Runtime Module ❧.....	20
SpaghettiMachine.....	20

Panel & Slot	20
Useful runtime mechanisms.....	20
Panel activation	20
Signals.....	22
Potentials.....	22
Visual Scripting with Spaghetti	25
A Dialog/Interaction system.....	25
The runtime part.....	26
Editor Reference ☞	28
Panel set definition.....	28
The Graph Editor	30
Basic buttons.....	30
Panelsets	30
Editing a graph.....	31
Tip: Order of Connections.....	31
Attaching Panels to Game Objects.....	32
Runtime Class Reference ☞	33
The Class SpaghettiMachine and its Members.....	33
SpaghettiMachine.....	33
SpaghettiMachine. mMachineType.....	34
SpaghettiMachine. LoadFromFile	34
SpaghettiMachine. LoadFromResources.....	34
SpaghettiMachine. LoadFromURL.....	34
SpaghettiMachine. LoadFromString	35
SpaghettiMachine. GetNumberOfPanels.....	35
SpaghettiMachine. GetPanel	35
SpaghettiMachine. GetPanels.....	36
SpaghettiMachine. FindPanelByType	36
SpaghettiMachine. OnGraphLoaded	36
The Class Panel and its Members	37
Panel	37
Panel. GetPanelType	37

Panel. GetSlots	37
Panel. GetInputSlots.....	38
Panel. GetOutputSlots.....	38
Panel. GetNumberOfSlots.....	38
Panel. GetNumberOfInputSlots	38
Panel. GetNumberOfOutputSlots	38
Panel. GetSlot	38
Panel. FindSlot.....	39
Panel. FindInputSlot.....	39
Panel. FindOutputSlot	39
Panel. GetMaster	39
Panel. Activate.....	39
Panel. Deactivate.....	40
Panel. IsActive.....	40
Panel. GetInputPotentials.....	40
Panel. GetSumOfInputPotentials.....	40
Panel. SetVariable	41
Panel. GetVariable.....	41
The Class Slot and its Members	42
Slot.....	42
Slot.GetPanel.....	42
Slot.GetNumberOfConnectedSlots	43
Slot.GetConnectedSlot.....	43
Slot.GetConnectedSlots.....	43
Slot.GetConnectedPanel.....	44
Slot.GetLabel.....	44
Slot.GetPlugType.....	44
Slot.GetContentType.....	44
Slot.GetColorName.....	45
Slot.GetSlotIndex.....	45
Slot.Below.....	45
Slot.Above.....	45

Slot.GetDataString	45
Slot.GetDataFloat	45
Slot.GetDataInt.....	46
Slot.GetDataBool	46
Slot.GetDataGameObject.....	46
Slot.ActivateConnected.....	46
Slot.SendSignalToConnected	46
Slot.GetPotential.....	46
Slot.SetPotential	47
Slot.SetVariable.....	47
Slot.GetVariable	47
Auxiliary types	48
PlugType.....	48
ContentType	49
Signal	49
Messages sent to the Master	50
HelloMaster.....	50
OnPanelActivated	50
OnPanelDeactivated	50
OnInputPotentialChanged.....	51
OnPanelReceivesSignal.....	51
The Examples 	52
Example 1 - Hello World	52
Purpose	52
Panel Sets	52
Runtime Scripts	52
Demonstrated Concepts.....	52
Projects that may require a similar approach.....	52
Example 2 - Tech Tree.....	53
Purpose	53
Panel Sets	53
Runtime Scripts	53

Demonstrated Concepts.....	53
Example 3 - Logic Gates	54
Purpose	54
Panel Sets	54
Runtime Scripts	54
Demonstrated Concepts.....	54
Projects that may require a similar approach.....	54
Example 4 - Text Adventure Map	55
Purpose	55
Panel Sets	55
Runtime Scripts	55
Demonstrated Concepts.....	55
Projects that may require a similar approach.....	55
Example 5 - Menu System	56
Purpose	56
Panel Sets	56
Runtime Scripts	56
Demonstrated Concepts.....	56
Projects that may require a similar approach.....	56
Example 6 - Dialog and Interaction System.....	57
Purpose	57
Panel Sets	57
Runtime Scripts	57
Demonstrated Concepts.....	57
Projects that may require a similar approach.....	57
Index ❸.....	58
Table of Content ❸.....	59



METHOD
in the
WADNESS