**A Tutorial for Behavior Networks Designer**

**Joan Llobera 2016**

In this document we will follow a step-by-step approach to the creation of characters which use Behavior Networks to make decisions. The aim is to get you, user of the Unity3D package Behavior Networks Designer, with an acquaintance of the possibilities it offers, and to allow you to use it for your own projects.

Creating characters with interactive behavior requires several additional tasks and decisions to make, before and after each of these steps. These are here outlined in detail in numbers (1,2,3…). However, once you are familiarized with the context of use, the 5 main steps (from a) to e)) are all you need to remember.

If you were to find major difficulties using this package once gone through this tutorial, please send a precise email, if necessary with the necessary prefabs and code chunks, to [support@timepath.io](mailto:support@timepath.io)

**1. Choosing an Animation Controller**

To use an artificial intelligence (AI) library effectively –for example, this Behavior Network Designer—requires having an animation controller that can render the actions chosen by the AI. In this tutorial we will use two examples of animation controllers.

The first, Mecanim, is the default’s animation controller in Unity3D, and to demonstrate its use we will use some assets from a project tutorial demonstrated by Unity3D at GDC 2013, and freely available at the asset store (see link at the end of this tutorial).

The second animation controller will be based on Adapt, a brilliant library for animation control created by Alexander Shoulson (among others) and which is also included in the Behavior Networks Designer in the form of a compiled library. The source code for Adapt is freely available, and it is designed to be easily expandable by the end-user

Once we have done a simple example with each of the animation controllers, we will briefly discuss the advantages and drawback of each, for you to choose the best option for your own project.

**2. Creating your characters and animation assets**

This is the most time consuming part of game creation, the one that requires patient hand-crafted work, artistic skills and a touch of inspiration. We will not address it here.

Please find 2 prefabs in the folder Assets > Prefabs, one called *adapt\_body*, and a second called *mecanim\_body*. Each of these already contains a rigged character linked to some animations, and some simple behavior scripts. They are each adapted from reference tutorial projects (to further explore these, see links at the end of this tutorial).

These 2 prefabs are already loaded within the starting scene of this tutorial, inside the folder *Assets > TutorialScenes*. The scene is called *Tutorial\_ready2start.unity.* Once you have opened this scene, we are ready to start creating your first personality.

**3. Creating a Behavior Network**

In this section, we will create a simple autonomous character that picks balls. The use of Behavior Networks Designer involves 5 steps, which are summarized below from a) to e). To use this tool effectively, these are the 5 steps you will need to remember.

1. **Create a Personality and the Agent that will use it**

The Personality of a character is a set of elements that define its behavior. It is essentially a list of Perceptions, Goals and Skills that, by being related between them, allow the character to make decisions that are appropriate for a given situation.

In Behavior Networks, the most demanding task is defining the Skills, which link the personality with the animation controller. However, once the Skills are defined, simple changes in the goals of the character allow for very different behavior patterns, thus favoring recycling of Skills for different purposes.

To create a new personality, go to Assets > Create > Timepath Personality

This will create an empty game object with a TPPersonality component. Before we explain how to define Perceptions, Goals and Skills within this personality, lets see a personality at work.

For this, you will need a personality example. In the folder *timepath4unity > Prefabs* you will find a personality prefab called meteorite\_picker. Just dragging it in the scene will instantiate it.

You will also need to create at least one Agent. An Agent is a character which makes autonomous decisions. In this context, this means it has a Personality, and a Body, i.e., a 3D rigged character with an Animation Controller (or an ADAPT Body). To create a new Agent, go to Assets > Create > Timepath Agent. This will create an empty game object with 2 components: a TPMentalBag, which you can ignore for a while, and a TPAgent, which should be the focus of your attention. In the component TPAgent, select the personality that you want that Agent to have. In this case, it should be the personality you just created, meteorite\_picker. You also need to select the body you want the agent to have. For this first example, select the gameObject called bodyADAPT, and which is part of the scene.

You can now press play and, if everything went fine, your first interactive character should be going to pick some meteorites to bring them all together.

By selecting the Timepath Agent that is walking around in the scene, it is possible to visualize its inspector, where the different perceptions, goals and skills of the character should evolve dynamically with its behavior. We can now stop the game execution and take a closer look at a personality, and how to create, copy, store or destroy its building blocks

1. **Create Perceptions, Goals and Skills (or destroy them)**

Once we have defined a Personality, we need to create Perceptions, Goals and Skills for it. Select the TPPersonality created. In the Inspector panel:

* To create a Perception, click on the black square with the “+” sign on either side of the Inspector.
* To create a Goal, click on the blue square with the “+” sign.
* To create a Skill, click on the yellow square with the “+” sign.

Creating 2 perceptions, one goal and 2 skills should look closely to Figure 3

* To destroy a Perception, click on the “-” button at the right of its name.
* To destroy a Goal, double click on the corresponding blue rectangle with a number.
* To destroy a Skill, double click on the corresponding yellow rectangle with a name.

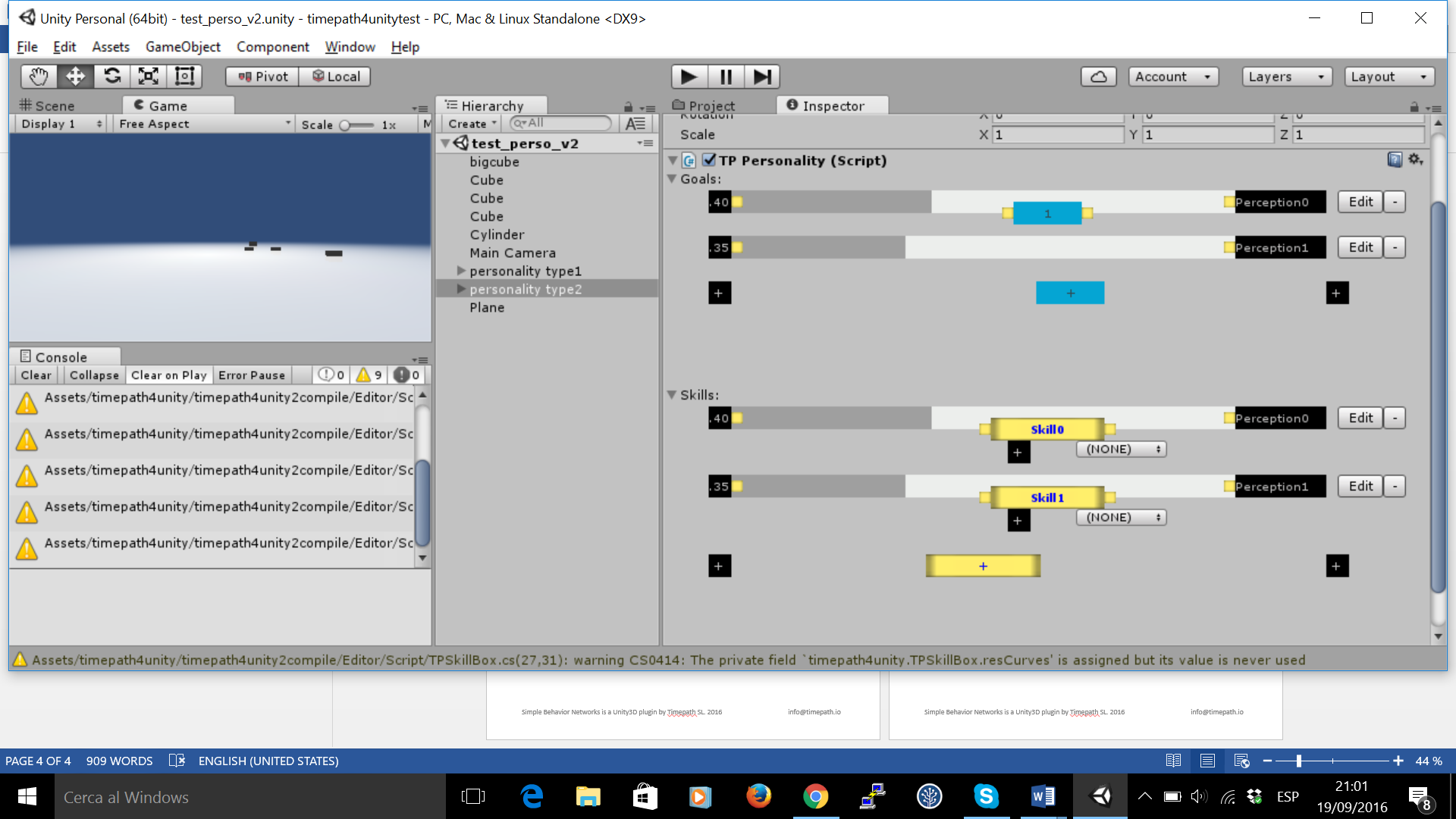


Figure 3 A personality with 2 perceptions, 1 goal and 2 skills

Now you know how to create and destroy perceptions, goals and skills, but you do not know how to use them

1. **Define the Perceptions**

To define a perception in a personality, click on the corresponding black square with a number (on the left), or on the corresponding “Edit” button next to its name (see Figure 3). This will take you to the appropriate game object in your personality. Once there, in the Inspector, an editable menu like in Figure 2 will appear, perceptions can be defined from a simple dropdown callback menu.

The perceptions that appear in the editor are defined as methods in the class TPPerception, whose source code is included in the package at the folder:

Assets > timepath4unity > TPPerception.cs

A Perception is any method of the class TPPerception which does not take more than one input parameter (a tag, a game object, another element) and which changes the double “value” with a quantity that is between 0 and 1. Combinations of methods can also be selected in the dropdown menu. Several examples are provided in the class TPPerception and in the personality prefab called “meteorite\_picker”.

These perception methods are fairly easy to define by a computer scientist, but do not hesitate asking a colleague with further programming experience to help you if you do not understand what is required.

1. **Define the Goals**

The Goals can be very simply defined by connecting the yellow nodes next to the blue boxes with the yellow nodes next to the black boxes (see Figure 1). The logical relation between a Perception and a Goal can be changed by clicking on the small box in the middle of the lines.

In the case of Relevance Conditions, at the right of the blue box, the lines will show either a green “\*” when affirmative or a red “NOT” when negated. The number in the blue box will show the relevance values resulting from combining these Perceptions.

In the case of Goal Conditions, at the left of the blue box, the lines will always show a number, but clicking on them will also allow negating the perception corresponding to the goal condition.

The result will correspond to a simple goal definition. For example, in figure 1, the goal defined corresponds to:

When:

not Perception0 and Perception1

I want:

Perception0 0.89

The reason to define these elements visually instead of doing it through direct textual rule writing is twofold:

1. It makes sure the formal syntax required is always preserved, and
2. It makes sure the user understands the visual layout. Indeed, at execution time, when the personality is embodied inside an agent, this visual layout will show how the changes in the perceptions will change the relevance value of the goal in real time, thus helping to debug the overall personality.

Finally, clicking on the blue box will select the game object, corresponding to that goal, and the inspector panel will show the rule created as well as manually allow adjusting the importance of the goal (the small number shown in the Personality Inspector, in the Goal Condition, as shown in Figure 1). This allows defining completely a Goal in a Simple Behavior Network.

1. **Define the Skills and the Actions**

The definition of the Skills and the Actions is the most difficult part of using Simple Behavior Networks. It involves 3 steps:

1. First, you need to state the logical relations with existing Perceptions, in a way similar to how we did it with Goals. For example, connecting a skill called “pickM” in the same way than the previou goals will generate a structure that corresponds to:

If:

Not Perception0 and Perception1

doing:

pickM

has effect:

Perception0 0.89

1. The second step to define a Skill corresponds to defining its associated Action, which is done in a similar way than Perceptions, but this time using methods defined in the class TPAction, which can be found in the folder:

Assets > timepath4unity > TPAction.cs

A method defined in TPAction can be any kind of method that has 1 input field, or combinations of them. The example provided uses the wonderful LGPL library for character animation the source code of which can be found at <https://github.com/storiesinvr/ADAPT> , and which integrates several character animation techniques to easily create sophisticated behavior. The tutorial to learn its ins and outs is also included in the package.

Alternatively, the freedom to call any class or method within a TPAction method implies that any other animation system can be used, and we wish different users are able to integrate this tool easily with their preferred animation frameworks (please let us know!).

For Mecanim enthousiasts, Timepath can also provide Inspector editable menus to rapidly prototype Simple Behavior Networks using the API that allows changing Mecanim state machines.

1. The third step to define a skill is the likeliness of the effects. For this, click on the corresponding Skill, unfold its corresponding game object. A small hierarchical structure will appear with the following items:
   * 1. When:
     2. DoAction:
     3. Effects:
     4. Using:

Under “3.Effects”, the effects created will also have an “effect likeliness” that will be editable, to adjust how likely performing a certain action under certain conditions is likely to have a desired effect.

SCREENSHOT OF EFFECTS

Depending on your editor settings, the numbered list of items in a Skill might appear disorganized, with item “2. DoAction:” appearing higher than “1.When:” . To correct for this fact, turn on Alphabetical sorting (see Annex in next page).