www.timepath.io

Behavior Networks Designer

Behavior Networks Designer is a Unity plugin designed to facilitate the creation of smart interactive characters. Behavior Networks Designer combines Extended Behavior Networks (Dorer 2004) with a simple user interface carefully thought to simplify the design of behavioral artificial intelligence by non-technical enthousiasts.

Introduction

In videogames, Artificial Intelligence (AI) is often created with Behavior Trees, a simple yet robust method to create characters with simple rules, such as the ones needed in first person shooters. Several implementations are available in Unity, some of them remarkably easy to use.

Behavior Networks are an alternative method to solve the same technical problem that Behavior Trees addresses. Arguably, it is a more powerful method, particularly in dynamic and fast-changing environments, which require more reactive and sophisticated behavior.

Behavior Networks are also more intuitive to design. Behavior Networks work by defining Skills and Goals in the interactive character. A goal has a syntax like:

When **x** I want **z**

A Skill is defined in a syntax like:

If **x** and **y** doing **a** has effect **z**

Both for goals and skills, **x**, **y** and **z** are Perceptions that can be quantified by a function returning a value between 0 and 1. Possible examples of Perceptions can be “I am near a fridge”, “I have the thermometer”, “I do not have ammo”, “She is smiling”, etc. In addition, in Skills, **a** is an action, which can be performed with a certain intensity, also between 0 and 1.

Decisions In Behavior Networks are taken through an energy spreading mechanism: a certain skill will perform a certain action if its effect is expected to contribute to a goal. Skills also self-organize according to preconditions (in the previous example, **x** and **y**) and its effects (in the previous example, **z**). For example, if character A wants to kill character B, but it needs a gun to do so, a skill such as “pick the gun” will receive a lot of energy. Later on, once the gun is obtained, then the energy will move to a skill such as “shoot at character B”, now more likely to contribute to achieve the goal.

User Interface

Behavior Networks Designer includes a dedicated user interface, which we have designed to:

1. allow non-technical users to create their own skills and goals, and
2. to visualize during game playout the evolution of the character and the Behavior Network

Using this interface, the AI designer can define the relations between Perceptions and Goals by dragging and dropping. During playout, the AI designer can see how the changes in the perception values affect the relevance of a given goal, or affect whether a given action can be performed. For example, in figure 1, a simple diagram corresponding to "When: not Perception0 and Perception1 I want: Perception0 " is shown. The black and gray rectangles show the extent at which a certain perception is true: in this case, Perception0 is true at 83%, and Perception1 is true at 54%. The green and red lines show the relation between the Goal and the Perceptions. The Goal, shown as a blue box, has a Relevance value corresponding to the smallest condition satisfied (in this case, the negation of Perception0, which is true at 17%). During playout, the evolution of the Perception values will in turn affect the Relevance of that particular Goal.

The Relevance of a given Goal will evolve in real time depending on the different Perceptions. In their turn, Perceptions can be defined through dropdown menus, from a set of simple functions that return a value between 0 and 1, and which can also be customized by the end-user (see figure 2). Skills and Actions are defined in a similar way to Goals and Perceptions.

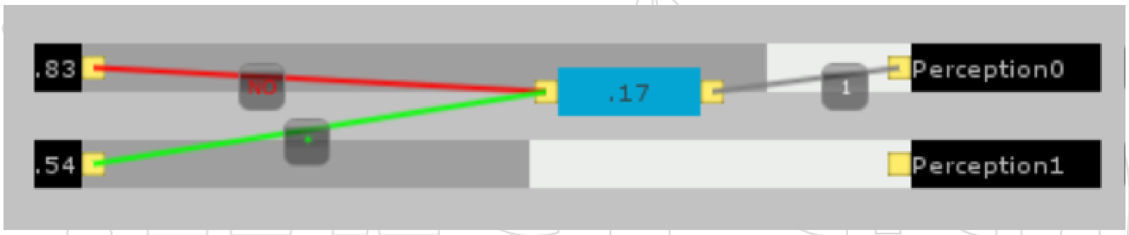
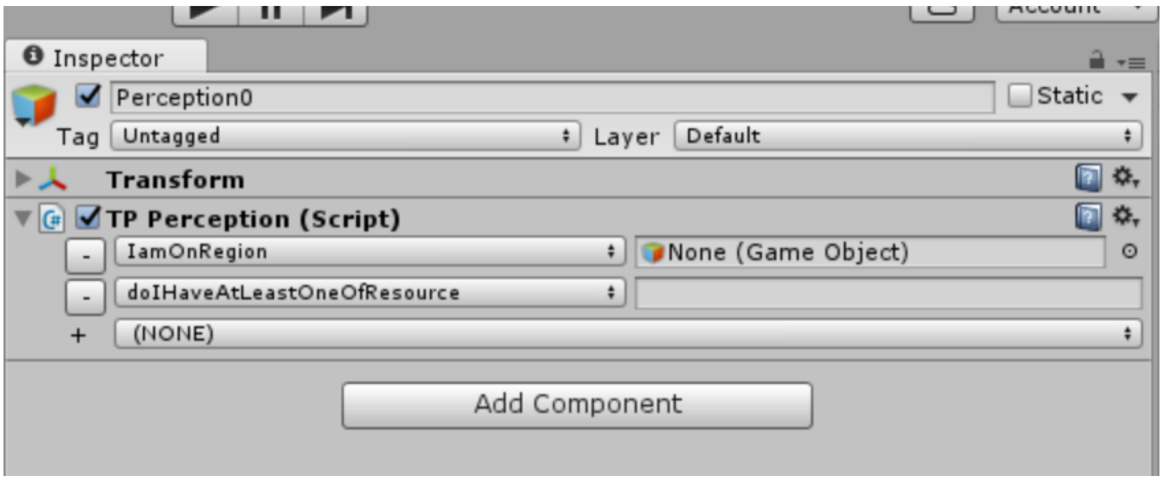


Figure 1: a simple user interface allows visualizing the real time evolution of perceptions and goals during playout



Perception0
Tag Untagged Layer Default
TP Perception (Script)
IamOnRegion None (Game Object)
doIHaveAtLeastOneOfResource
(NONE)

Figure 2: through the Unity Inspector, perceptions can be defined easily from a list of pre-defined and customizable functions.

Animation Synthesis

Behavior Networks provide a modular way to design smart interactive behavior, based on Goals and Skills. To facilitate this task, this package also includes an innovative method to blend different animations together (Shoulson 2014), embodied in a wonderful LGPL library the source code of which can be found at https://github.com/storiesinvr/ADAPT . This library integrates several character animation techniques to create sophisticated behavior, and it is easily extendable.

Timepath is and will remain eternally grateful to Alejandro Beacco for pointing out the existence of this library and to Alexander Shoulson and any other relevant contributor for creating this piece of software and sharing it with the world.

Mecanim integration is also available upon request.

Customization and extension

Using this package, Behavior Networks can be defined as prefabs, stored, shared and modified like any other Unity asset. Perceptions and Actions provided can easily be customized selecting built-in functions directly from the Inspector Panel. New Perception and Action functions can also be created through small chunks of code leveraging the power of the Unity3D API.

Behavior Networks Designer uses a runtime library compiled for Windows. Version for Android, Mac, Linux, iOS and WebGL will follow soon.

For further information on how to use Behavior Networks Designer, please go through the Quick Start Guide that follows.

References

K. Dorer, (2004). Extended behavior networks for behavior selection in dynamic and continuous domains. In *Proceedings of the ECAI workshop Agents in dynamic domains, Valencia, Spain*.

A. Shoulson, N. Marshak, M. Kapadia, and N. I. Badler (2014) ADAPT: The Agent Development and Prototyping Testbed. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*,

Quick Start Guide

The creation of a Behavior Network involves 5 steps, which are summarized below from a) to e). If you were to find major difficulties using this package once gone through this quick start guide, send a precise email, if necessary with the necessary prefabs and code chunks, to support@timepath.io

a) Create a Personality and the Agent that will use it

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

To create a new character with a personality, go to Assets > Create > Timepath Agent. In the component TPAgent, select the personality that you want that Agent to have. Upon pressing play, it will be possible to visualize in real time the dynamic evolution of the behavior network (see example in the sample scene provided).

b) Create Perceptions, Goals and Skills (or destroy them)

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.

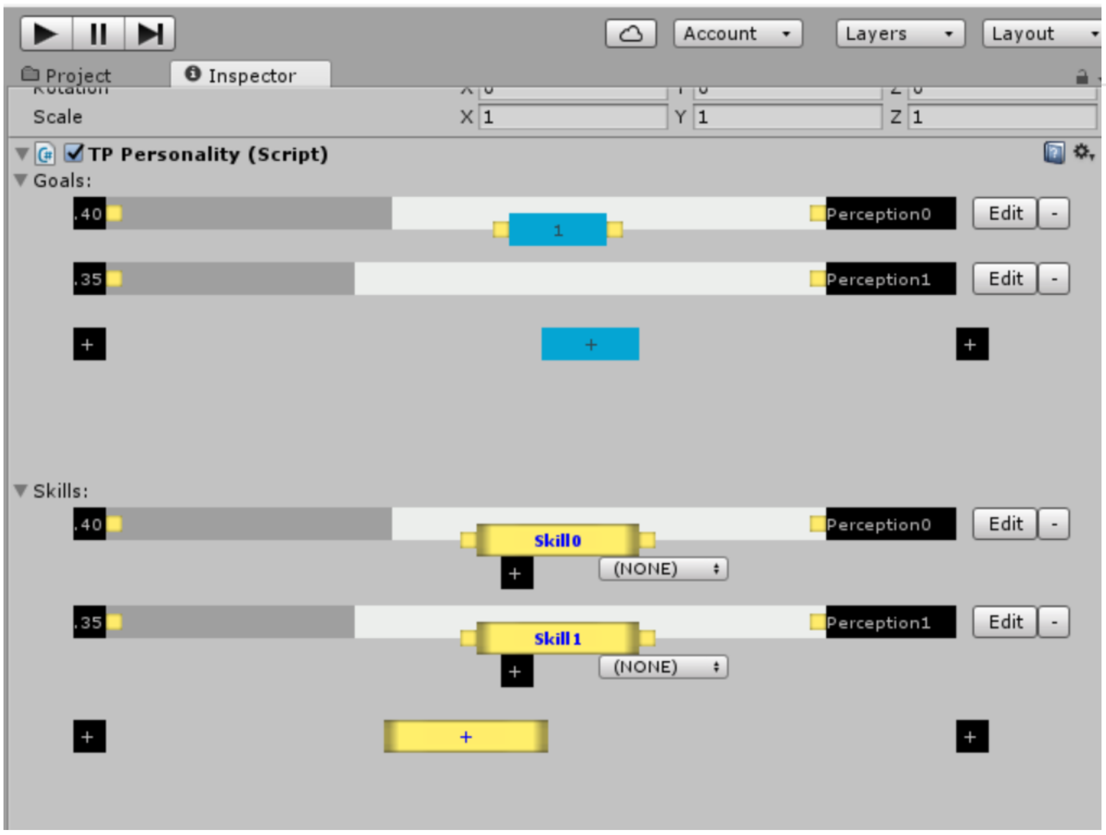


Figure 3 A personality with 2 perceptions, 1 goal and 2 skills

c) 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.

d) 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. To make 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.

e) 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
```

2. 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. Despite this option is not the most recommended, please send an email to support@timepath.io, and they will be provided on the shortest notice possible.

3. 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.

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).

f) Optional: refinement with Resources

The previous combination of building blocks –Perceptions, Goals and Skills- already allows defining quite complex character behavior. However, the user can also extend personalities with Resources, which will appear under the item “4.Using:”

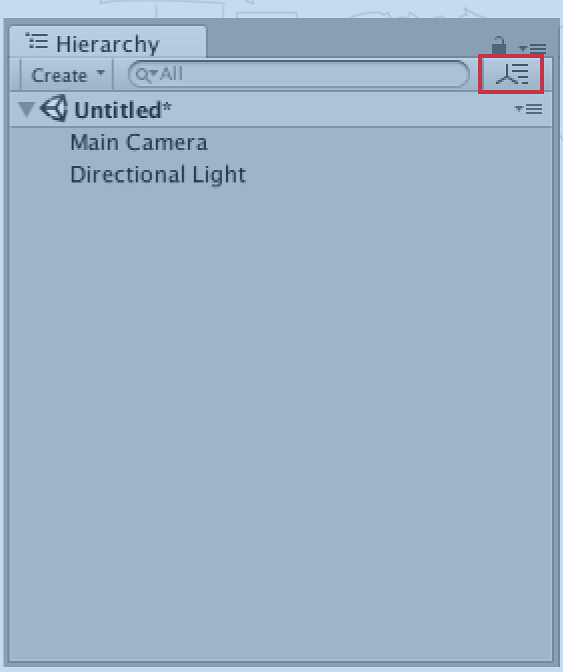
Resources are useful when defining Skills that compete for a limited amount of world, such as typically “Energy”, “Stamina”, but also anything, like “Rocks”, “Spaghetti” or any other resource that in a given environment is scarce enough to not be available at the same time for all the Skills that might want to use it.

To define Resources, select the personality, and under the appropriate skill select an item in the small menu that appears under the yellow rectangle, either an existing resource or by creating a new one. Once selected, clicking on the name will select the appropriate gameObject and show the options available for editing. The prefab “meteorite_picker_resources” contains an example of a personality using Resources.

Annex: how to turn on Alphabetical Sorting. From the Unity3D Manual:

The order of objects in the Hierarchy window can be changed to alphanumeric order. In the menu bar, select **Edit > Preferences** in Windows or **Unity > Preferences** in OS X to launch the **Preferences** window. Check **Enable Alpha Numeric Sorting**.

When you check this, an icon appears in the top-right of the Hierarchy window, allowing you to toggle between **Transform**sorting (the default value) or **Alphabetic** sorting.



Source: https://docs.unity3d.com/Manual/Hierarchy.html

Troubleshooting

If your character does not perform the action that you expect it to, you can try the following options:

1. Write in your action a log message to make sure the action is not being excuted (you can use Debug.Log("my message");)
2. If at a given time a character is not performing an action, use the visualization to clarify why this could be the case. Concretely, this implies:
   a. Check the activation value (the small number in the box), and confirm it is among the ones with high activation
   b. Check if the action contributes to a relevant goal (check the expected effect of performing the skill contributes to a Goal with high relevance value)
   c. Check if the preconditions of the skill are satisfied (i.e., low values for negated items, and high values for non-negated items)
   d. Check the lack of resources needed for the action is not a blocking factor
   e. If it is a Skill that will satisfy a Precondition for another Skill, check whether the relations between Perceptions satisfying successive Preconditions of Skills do end up contributing to a Relevant Goal.
3. Once you have understood why it is not doing the behavior you want it to for that moment, stop the playout, change the Relevance conditions of a Goal, and/or the Preconditions of a Skill, and test it again to check if the behavioral performance of your character to go in your desired direction possible when you want it to.