LIGHT BEHAVIOR

USER GUIDE

Index

Thank you for choosing to support Onpolyx by buying and downloading this asset pack from the Unity asset store.

If you have any questions/problems regarding this asset pack or suggestions on something you'd like to see added in a future update please email me at Onpolyx@gmail.com.

1. Introduction

Light Behavior is a Unity asset pack containing several C#-scripts that changes the behavior of a light source.

All scripts are designed with a drag-and-drop mentality in mind, to make it as easy as possible to add the desired behavior to your light source, while still providing full control of the behavior via the Unity Inspector.
Contained in the package are also some light-prefabs, these does not only serve as a tool to show what can be done with the behavior scripts but can also be used a quick and easy way to get a desired light-effect added to your scene.

In this manual we'll be looking at what behaviors can be found in the package, how we would go about adding a behavior onto our light source and how to use the light-prefabs.

As the last part we'll also be taking a quick look at how the scripts are designed, to give a general idea of how everything works. This part will require prior programming knowledge; however it is not necessary to read this part unless you're planning on adding/rewriting any code in the scripts.

2. Using the prefabs

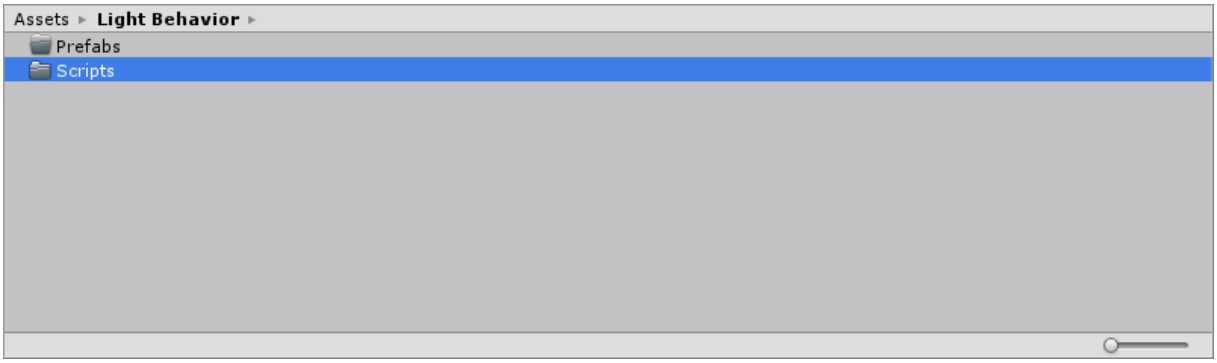
Inside of the folder ***Prefabs*** you will find a bunch of light-prefabs. These are point/spot and directional lights which has already been assigned a behavior corresponding to the name of the prefab and been tweaked to fit that behavior.

This is to make sure you don't have to create a new light source and add a behavior to it every time you want a specific light source in your scene, simply drag the wanted prefabs from the folder and drop it into your scene and you're good to go!

Another function of these prefabs is that they serve as tools to show you what can be achieved with the different behavior scripts and how they are intended to be used.

3. Adding Behavior As Component

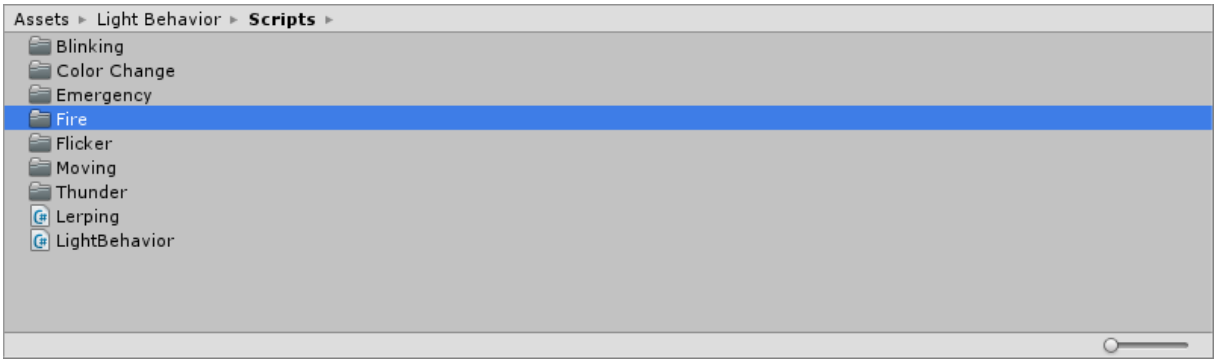
After importing Light Behavior into your Unity project, all of the behavior scripts can be found within the folder ***Light Behavior -> Scripts***



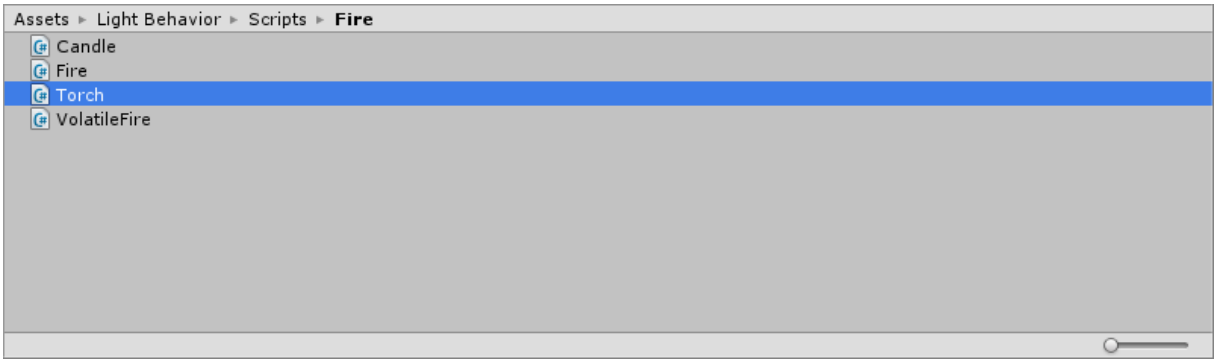
Inside, you will find 7 new folders called ***Color Change, Emergency, Explosion, Fire, Flicker, Moving*** and ***Thunder.***
As you might assume, these folders contains all of the scripts that is in some way linked with that specific type of behavior. So if we're looking for a script that would make our light act like thunder, this script would be found within the ***Thunder*** folder.

As an example, let's assume we have a particle-effect in our scene that simulates a torch and has a point light attached to it. We might want to add a behavior to the point light to make it behave like a torch.

We know that torches are made of fire, so continue by opening the folder ***Fire***



You will now be presented with some C# scripts and some more descriptive names; these are the actual types of light behaviors.
Here you'll be able to find a script called *Torch.cs*, great! Just what we needed! Simply drag the script from the folder and drop it on top of your point-light.



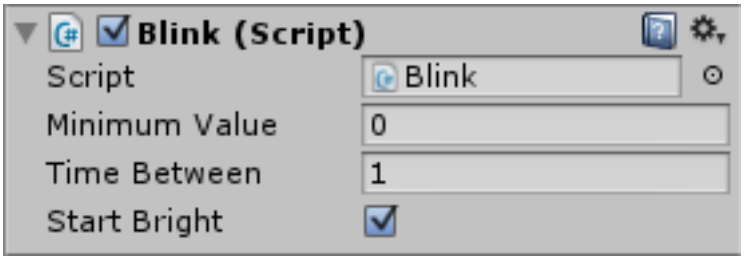
That's all you have to do! When you press play and test your level the point light is now behaving in a way that simulates a torch.

NOTE
Within the folder Scripts there are two scripts called LightBehavior.cs and Lerping.cs, these serve as a skeleton for all behaviors and should ***NOT*** *be added to the light source.*

4. Adding Behavior | *Public values*

Most of the behaviors have some public values contained within them, this is to be able to control the behavior directly via the Unity inspector, without having to open the script itself to tweak some values.

Let's look at the script *Blink.cs* as an example:
Whenever we add this script to a light source we are presented with this component in the inspector:



Leaving these values at default will work just fine, but tweaking some of the values might make the behavior suit your needs more perfectly. Every behavior -script will offer different types of values to be tweaked, but I've done my best to name all of these values in a descriptive way and we'll take a quick look at these values later on.

5. Behaviors| *How they work*

Most of the behaviors function in a very similar way, multiplying the original color of the light with a multiplier which makes the color of our light source become darker/brighter.
For example; let's say we have a white light source and we want to dim this light to a darker color. Instead of having to go through the hassle of changing intensity of the light source which puts a lot of unnecessary stress on the system we would just say: **Light.color = OriginalColor * 0.5;**

Light.Color is the current color of the light source and **OriginalColor** is the original color of that light source.

Basically what we're saying here is that the color of our light source should be our original color multiplied by 0.5, in the case of our white light source this would result with a grayish color.

This is because:
white * 1 = white
white * 0.5 = gray
white * 0 = black

6. Behaviors – In Depth Look

Since the package contains so many different behaviors and all of these have different values to be tweaked, let's take a more in-depth look at each behavior; what they do and how to control them.

6.1 Blink

Makes the light blink, changing from minimum allowed value to original value at a fixed rate.

- **Minimum value**
  Minimum allowed strength of the light source, 1 being completely bright and 0 being black

- **Time Between**
  Seconds between each change made to the light source

- **Start bright**
  If the light source should be turned on when starting

6.2 Dual Color

Light will start off with its original color, fading to the second color and then fade back to its original color.

- **Second Color**
  New color that the light should fade into

- **Time Between**
  Seconds between each full color change

6.3 Dual Color Simple

Light will start off with its original color, fade to the second color. It will then jump straight back to its original color.

- **Second Color**
  New color that the light should fade into

- **Time Between**
  Seconds between each full color change

6.4 Calm Emergency

Light starts of completely bright and fades out, never going completely black, before fading back to its original brightness.

6.5 Intense Emergency

Light starts of completely bright and fades to black, before fading back to its original brightness.

6.6 Candle

Light fades in and out in a fixed rate with very small changes, simulating a candle.

6.7 Fire

Light simulates a fire, fading in and out with. On a fixed rate it goes into “intense mode” which makes the light fade in and out at a more intense rate, giving the light a more living and unexpected feel to it.

- **Calm Time**
  How long, in seconds, the fire should remain in “calm mode”

- **Intense Time**
  How long, in seconds, the fire should remain in “intense mode”

6.8 Torch

The light fades in and out on a fixed rate, simulating a torch.

6.9 Volatile Fire

Simulates a fire that acts a bit more volatile, fading in and out in a fixed rate with bigger changes than Fire. Like Fire, Volatile Fire also goes into “intense mode” which makes the light fade in and out in a more intense way.

- **Calm Time**
  How long, in seconds, the fire should remain in “calm mode”

- **Intense Time**
  How long, in seconds, the fire should remain in “intense mode”

6.10 Flicker

Light flickers, going between minimum value and completely bright in a random fashion.

- **Minimum value**
  Minimum allowed strength of the light source, 1 being completely bright and 0 being black

- **Flicker Frequency**
  Controls the frequency of the flickering. 0 being always turned off , 100 being always turned on.

6.11 Horror Flicker

Light flickers in a more fixed rate, simulating a broken lamp used for horror games.

- **Minimum value**
  Minimum allowed strength of the light source, 1 being completely bright and 0 being black

6.12 Move Between

The light will move from its original position towards another location, when reaching the new location the light source will start moving back towards its original position.

- **Target Location**
  Used to get a position from another game object inside of the level. This can be any kind of game object, but I suggest using an empty game object that is static to prevent potential errors.

- **Seconds Between**
  Travel speed for the light source between each destination.

6.13 Move One Way

The light source will move from its original position towards another location, when reaching the new location the light source will jump back to its original position.

- **Target Location**
  Used to get a position from another game object inside of the level. This can be any kind of game object, but I suggest using an empty game object that is static to prevent potential errors.

- **Seconds Between**
  Travel speed for the light source between each destination.

6.14 Thunder

Used to simulate lightning strikes over a large area.
The light source will start of black, when reaching a fixed time it will flicker on and off in a very short time, simulating a thunder strike.

- **Time Between**
  Seconds between each thunder strike

- **Sound (optional)**
  Sound effect that should play whenever a lightning strike occurs

6.15 Thunder Window

Used to simulate lightning strikes that's flashing through a window.
The light source will start of black, when reaching a fixed time it will flicker on and off in a very short time, simulating a thunder strike.

- **Time Between**
  Seconds between each thunder strike

7. Scripts – In Depth Look

To implement Light behavior into your games you should never be required to open the scripts and make any changes in the actual code.

However, should you decide to edit any code or just want to know what these scripts are actually doing when compiled, this section will help you understand how these scripts are designed and structured.

I will not be going in-depth for every single behavior or explain all variables used. Would you like to know more about a specific script I would suggest opening that script and reading the code, I have done my best to describe everything as best I can through comments.

7.1 Base Classes

If you remember from our folder-structure, there are two C# scripts located within the Scripts folder, named *LightBehavior.CS* and *Lerping.CS*, that we never seem to use when we're adding behaviors to our light sources.
This is because these scripts function as base-classes for our behaviors and so they are actually "added" by default whenever we add a behavior to the light source.

7.2 LightBehavior.CS

LightBehavior.CS function as the main base-class, and so all different types of behaviors should inherit from this class. It contains all general variables such as thisLight, timePassed and changeValue.

All of these variables are set to private, so whenever we want to use any of these variables we are instead using accessors such as Get/Set .
This is to limit access of these variables, making sure that our child-class can't change something important like what our target light is.
Another thing that's great with private variables is that they won't show up in the inspector, which would only serve to confuse the User.

Light Behavior initializes all its values within its void Awake(), this is because we want to make sure that all base values contained within LightBehavior.cs have been initialized **BEFORE** we try to initialize any values inside of our specific behavior scripts, which is done within Start() of that behavior-script.

7.3 Lerping.CS

Lerping.CS serve as the base-class for all different types of behaviors that uses .Lerp in any kind of way.
Lerping.CS inherits from LightBehavior.CS which gives us access to all base values from our behavior scripts. Initliazation within Lerping.CS looks a bit different though, and is contained within a separate method; void Initialize().
This is to make sure we're not relying on Awake() being ran after LightBehaviors void Awake() and before the Start() method within out behavior script.
Instead we just call Initialize() from the behavior scripts void Start() before initializing any values linked to our specific behavior script.

7.4 Mathf.Sin()

Most of the time when we're changing the color of our light source we do so by using some variation of Mathf.Sin().
This is because, while we might want to make the light behave in a seemingly random way, we still want to make sure that we know exactly what our light source is doing.

In the case of Candle.CS the method to change the actual color of the light source looks like this

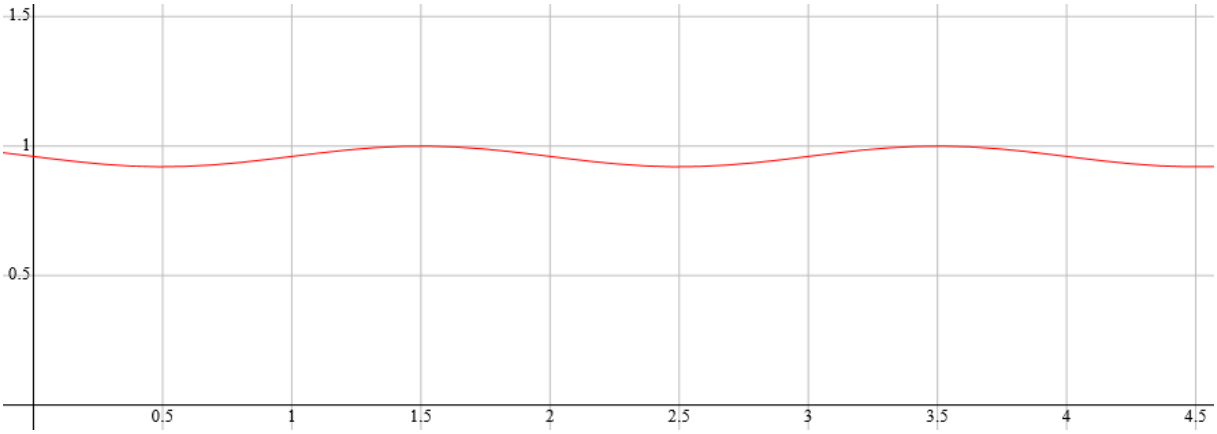
```
ChangeValue = -Mathf.Sin (TimePassed * Mathf.PI) * 0.04f + 0.96f;
```

Just looking at this method might make you want to scratch your head but all we're actually saying right here is:

Y = -Sin (X * pi) * 0.04 + 0.96

changeValue will act as **Y** because it is depended on timePassed which acts as **X**, moving us further along the graph.

Would we take this calculus and enter it into a graph-calculator we wound end up with a graph looking like this:



This allows us to make sure that the light behaves in a predicted and pre-set way, while still giving us great flexibility in **HOW** the light should act.