Shape Drivers User Manual

1. Introduction

Shape Drivers is a tool that lets you drive a mesh's blend shapes using various properties.

Blend shapes (also known as morph targets or shapekeys in other programs) can be used for all kinds of purposes, and are typically used to obtain shapes that are hard to get using skeleton-based deformations. Examples of this include complex mouth shapes and other facial features, shapes that deform in many places at once (like an ocean, or a piece of cloth) and corrective shapes to fix rigging or skinning artifacts (as often seen in knee or elbow bends).

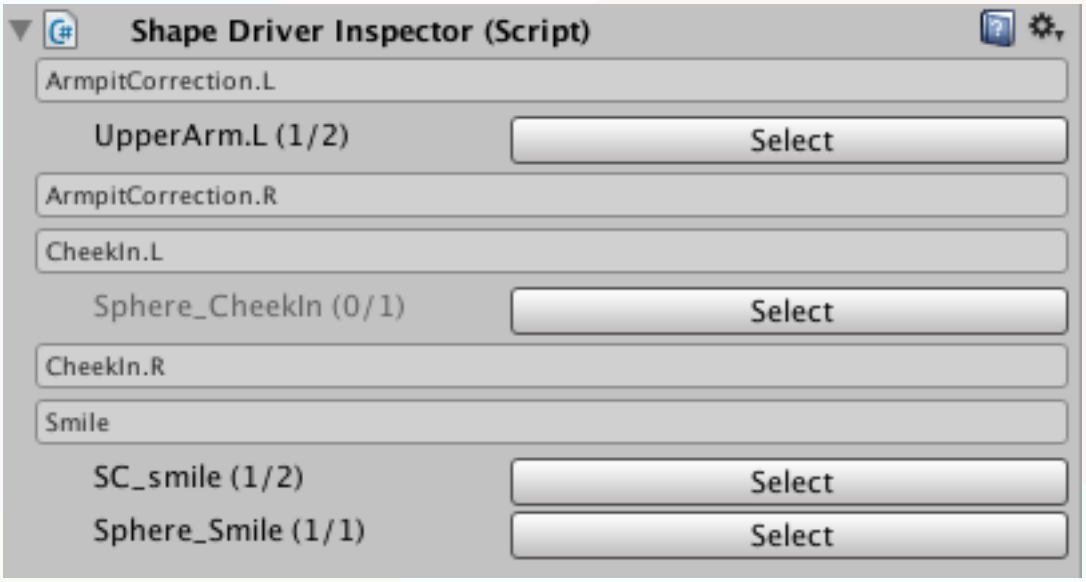
While Unity has supported blend shapes for a while now, there hasn't been a way to easily control them at runtime without rolling your own scripting solution. Here's where *Shape Drivers* comes in. Easy-to-use, yet extremely powerful, *Shape Drivers* lets you set up complex drivers to control your blend shapes at runtime with no scripting required!

2. Structure

Before taking an in-depth at each of the drivers' settings, it's important to understand how *Shape Drivers* works.

First, any object can become a *Shape Driver* by adding the *ShapeDriver* component to it. In most cases you'll likely want to drive shapes from either the shape-holding object itself or a bone from its skeleton, but there are situations where you might want to use a completely separate object. For example, the distance between a "fan" object and a character's head bone could drive a "dog-with-his-head-out-of-the-car" shape.

Because this responsibility can lie with any object, a shape can be driven from many different objects, even from more than one at a time. To see which objects are driving a particular object's shapes you can attach a *ShapeDriverInspector* component. This component won't do anything at runtime, but will allow you to gain a better overview of what's going on in your scene.



ShapeDriverInspector Component

The numbers behind the object names represent how many *Shape Drivers* on that object affecting that particular shape are active and exist in total respectively. Disabled GameObjects do not appear in this list.

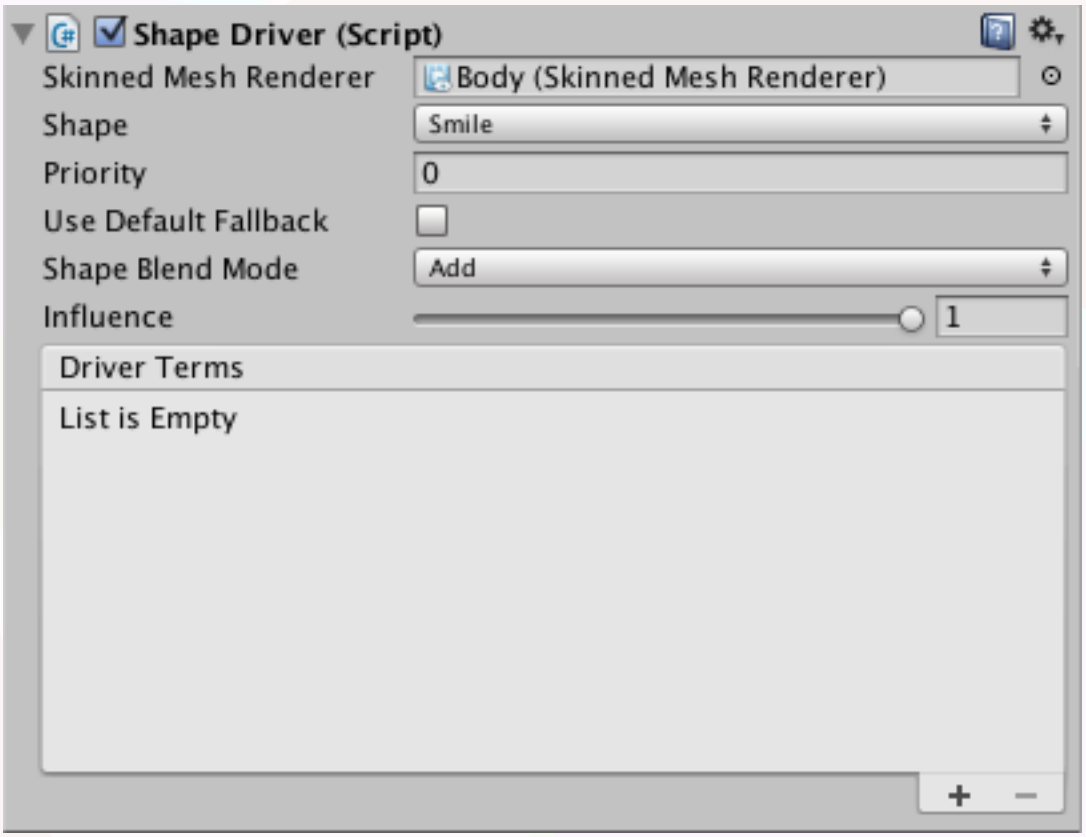
Pressing any of the “Select” buttons will select that object in the scene hierarchy, allowing you to easily locate and edit relevant drivers.

At run-time, once it’s established which drivers drive which shapes, the drivers are sorted by their priority value. Priorities allow for shapes to be driven by multiple drivers, while still producing consistent results. This, in combination with the different shape blending modes (explained below), allows you to create very complex blending setups where needed.

Shape Drivers themselves derive their final value by evaluating one or more *Driver Terms*. *Driver Terms* are the building blocks where all the actual magic happens. Like *Shape Drivers*, they are evaluated sequentially and can have their results either be added to or multiplied with their preceding values.

3. The *ShapeDriver* Component

Now let's take a more in-depth look at the *ShapeDriver* component and its properties.



ShapeDriver Component

First up is the *Skinned Mesh Renderer*. This is the object that you wish to affect a shape of.

The second entry on our list, *Shape*, is the actual shape that you want to drive.

Next up is the *Priority* value. This integer determines the evaluation order of all *Shape Drivers* affecting any one shape. Drivers are sorted low-to-high, with the lowest number being evaluated first. When two drivers targeting the same object and shape also have the same priority their order is undefined and may be inconsistent between play sessions.

In some cases, it might be that the driver doesn't have any value to return. (For example, when the driver doesn't contain any driver terms, or the ones it does don't have any value to return.) When this happens, you can choose to return a default fallback value instead. Toggling the *Use Default Fallback* on will show you the *Default Value* field to enter this value into.

Expanded Fallback Property

When the toggle is off and there is no value to return the driver will not be taken into account at all, and will not modify its targeted shape in any way. When used like this in

combination with a *Driver Term* that only returns a value within a certain range, the shape will not be reset once that term is no longer returning values. This results in a similar effect as Unity's old *Animation* wrap mode *Clamp Forever*.

At the start of each evaluation run (performed in `LateUpdate()`), the starting value is reset to the first active driver (a driver returning a value's) result. Each subsequent driver's result value is then blended into the previous one using one of four *Shape Blend Mode*s:

- **Add:** The new value is added to the previous one.
- **Multiply:** The new value is multiplied with the previous one.
- **Min:** The new value replaces the previous one if it is smaller.
- **Max:** The new value replaces the previous one if it is greater.

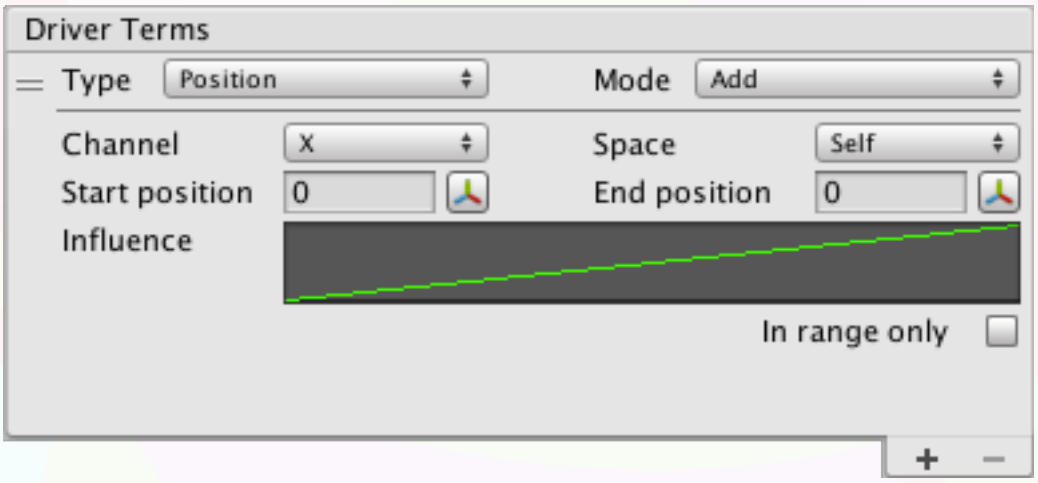
Note though that, while the *Add* operation can lead to, and will operate on, numbers outside of the [0-1] range, after the final driver is evaluated the value is clamped back to within that range by Unity internally, as shapes cannot be extrapolated beyond their default range (as one might be used to from their 3D authoring software).

A final note on blending modes: When the first returned value for a shape has its mode set to *Multiply*, the value will not have any effect. Conceptually, in the case of blending modes you can safely think of "No value" as being the same as the value *0*.

Lastly, the *Influence* property acts as a driver-wide multiplier. This value is applied after all the *Driver Terms* have been evaluated and the driver has produced a final result. The influence does not affect the driver's default value used in case no value could be produced.

4. Driver Terms

A *Shape Driver* without *Driver Terms* is like a car without wheels; it doesn't get you very far. *Driver Terms* are where the actual driver value is calculated. This value can be derived from *Transform* values, distance to a point or object, (*Mecanim* or *Legacy*) *Animation* time, *Mecanim Parameters* or a *script callback*.



Position-type DriverTerm Component

All *Driver Terms* start off with a *Type* property. The full list of available types is:

- **Position:** The value is derived from the driver object's position along any one axis.
- **Rotation:** The value is derived from the driver object's rotation along any one axis.
- **Scale:** The value is derived from the driver object's scale along any one axis.
- **Distance To Point:** The value is derived from the driver object's distance to a point on any combination of axes.
- **Distance To Object:** The value is derived from the driver object's distance to an object on any combination of axes.
- **Mecanim Parameter:** The value is derived from a *Mecanim* Int, Float or Bool parameter.
- **Mecanim Time:** The value is derived from a *Mecanim* state's current time.
- **Legacy Anim. Time:** The value is derived from a *Legacy Animation* clip's current time.
- **Script callback:** The value is derived from the return value of a method available on any of the object's components.

When a *Driver Term* is not the first in the list it also displays a *Mode* property. Like with *Shape Drivers* themselves, the mode determines how the new value should be added to the current one, provided a value exists. The different modes here are:

- **Add:** The new value is added to the previous one.
- **Multiply:** The new value is multiplied with the previous one.

Note that, as with *Shape Drivers*, any *Driver Term* with its mode set to *Multiply* will be ignored if there isn't a value to multiply it with already.

Skipping a row of properties for now, every *Driver Term* contains some form of a *Start* and *End* value. This is the range at which the *Influence* curve maps its [0-1] range on the horizontal axis. This may sound a bit confusing, so let's look at an example.

Let's say you want a shape to be at *20%* strength when the driver object's position is at *4.5* on a specific axis, and *100%* when it is at *10.0*. You would set the *Start position* to *4.5* and the *End position* to *10.0*. This means that, as far as the influence curve is concerned, *4.5* maps to *0.0*, and *10.0* maps to *1.0* on the horizontal axis. Next, you would push the curve's first key to coordinates *(0.0, 0.2)*, and the second key to *(1.0, 1.0)*.

Now you may have noticed that the curve's values on the vertical axis only lie in the range *[0, 1]*, whereas a shape is in reality driven by the range *[0, 100]*. This conversion is handled for you, and is done to provide a normalized curve that is easier to read and maintain.

Getting back to the *Start* and *End* value fields, you may have noticed that next to both of them is a little button. These buttons set their respective fields' values to whatever the current value of the monitored property is. This way you can quickly and precisely set the values you need.

Finally, the last control that all terms have is the *In range only* toggle. When this is checked, the term will not return any value if the monitored value falls outside of the *[Start value, End value]* range. If the value falls outside of this range with the toggle turned off the curve is still evaluated and will return a result in accordance with its *Wrap Mode*. (You can set this wrap mode in the curve's edit window that pops up when you click it in the *Inspector*.)

And that concludes the properties shared between every *Driver Term* type! Next, let's take a step back and take look at that row of properties we skipped earlier. All *Transform*-based *Driver Term*s contain a *Channel* property. This allows you to select the axis to monitor the value of.

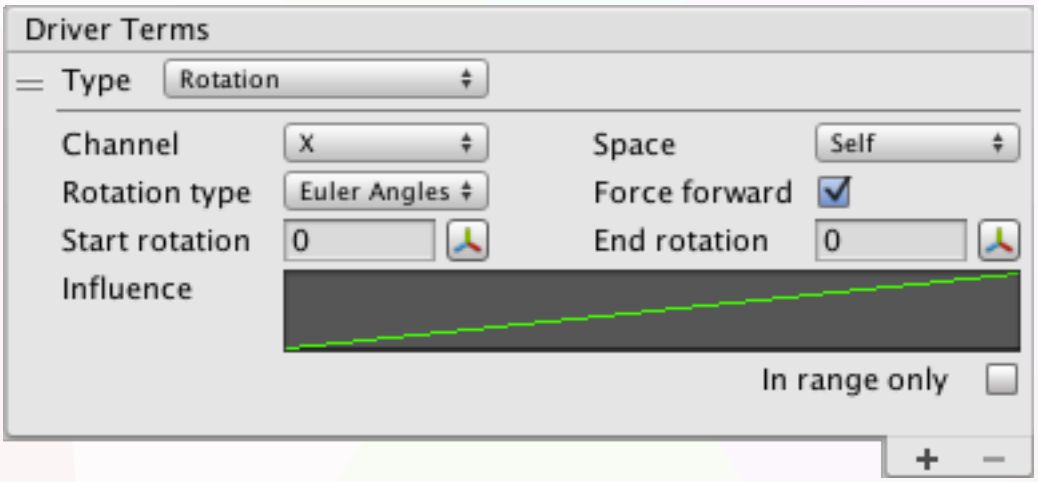
Regarding the final property, both *Transform*-based and distance-based drivers have a *Space* property that let's you monitor the value in either local space (*Self*) or world space (*World*). The world space monitors the value relative to the scene, whereas local space monitors the value as relative to itself and any possible parent objects.

For example, an object rotated +90º along the scene's X-axis will have its up-vector pointing *up* in its local space, but *forward* in world space.

(Tip: Use the editor's *[Local | Global]* toggle at the top-left corner, next to the transform tools, to visualize an object's axes in the desired space.)

Editor Space Toggle

While this covers all the properties for the *Position* and *Scale* type *Driver Terms*, the *Rotation* one is a stubborn little thing, and requires a few more options.



Rotation-type DriverTerm Component

The first property added is the *Rotation type*. This can be one of three values:

- **Euler Angles:** The transform's standard *eulerAngles* and *localEulerAngles* values are used as-is. Due to the way Unity stores and handles rotations internally axis values may change or "flip" at seemingly arbitrary times. Use this mode only if you are sure the object's rotation values will remain consistent enough for this driver's execution. Values lie in the range [-90, 90].
- **Pitch Yaw Roll:** The transform's standard *eulerAngles* and *localEulerAngles* values are converted to a *Pitch Yaw Roll* equivalent. While less intuitive in some ways, this rotation type is more stable than its *Euler Angles* counterpart, and should prevent unexpected axis changes or flipping in most cases. However, in some cases where the object is rotated over multiple axes it can also lead to inconsistent or unexpected results. Values lie in the range [-180, 180].

- **Single Axis:** The object's rotation is derived using a local coordinate system for this object. While highly stable and consistent, this method only works for bones that rotate along a single axis (or very little along any others, in some cases). This is the recommended type for things like elbow correction shapes, where the arm bone only or mostly rotates along a single axis. Values lie in the range [-360, 360].

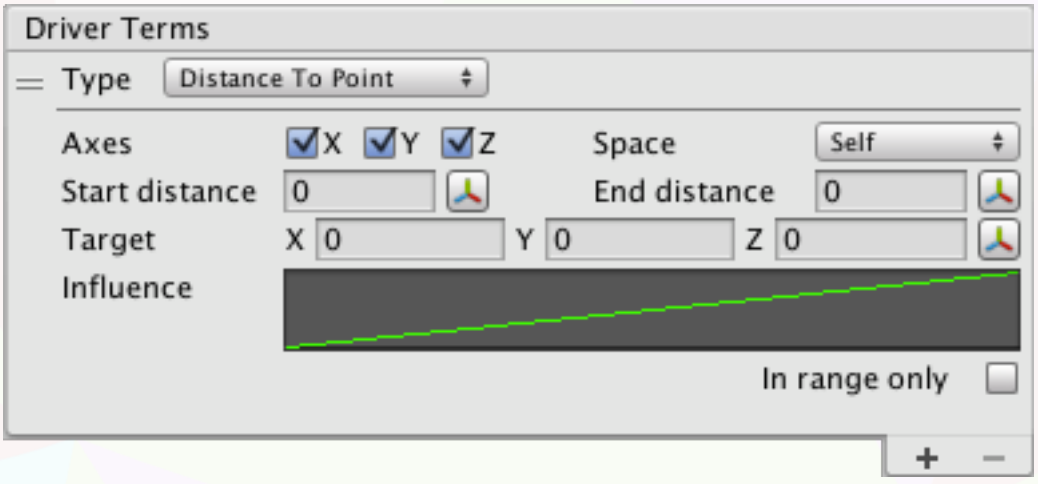
The second property unique to the *Rotation Driver Term* is the *Force forward* toggle. When a *Driver Term's Start value* is higher than the *End value* the value is typically interpolated as usual, to maintain consistency. However, with rotations it may be the case that you need to pass the "turning point" where 360º reverts back to 0º (or -180º to 180º or vice versa).

For example, going from 350º to 15º would normally interpolate the long way round, passing 340º, 330º, ..., all the way to 15º. However, you may need the rotation to go past 360º instead, effectively first moving to 0º, then moving on to 15º. *Force forward* forces this by converting to *Start value* to a negative amount in its `Init()` method. This is done by subtracting 360º. In our example of using 350º, this would thus be converted to -10º instead.

You can also enter in a negative *Start value* yourself; the *Force forward* toggle is mostly there to keep things clear, as well as allowing you to use the positive values created by the "set from current value" buttons.

Now a quick word about the *Scale Driver Term*. Unity doesn't have any internal representation for a "real" world-space scale; everything is based on the scale along an object's local axes, combined with the same for all its parents. Any scale value requested in world space will therefore be an approximation, arrived at by multiplying the *Transform's lossyscale* by its rotation.

Next up, let's take a look at the *Distance*-type *Driver Terms*.



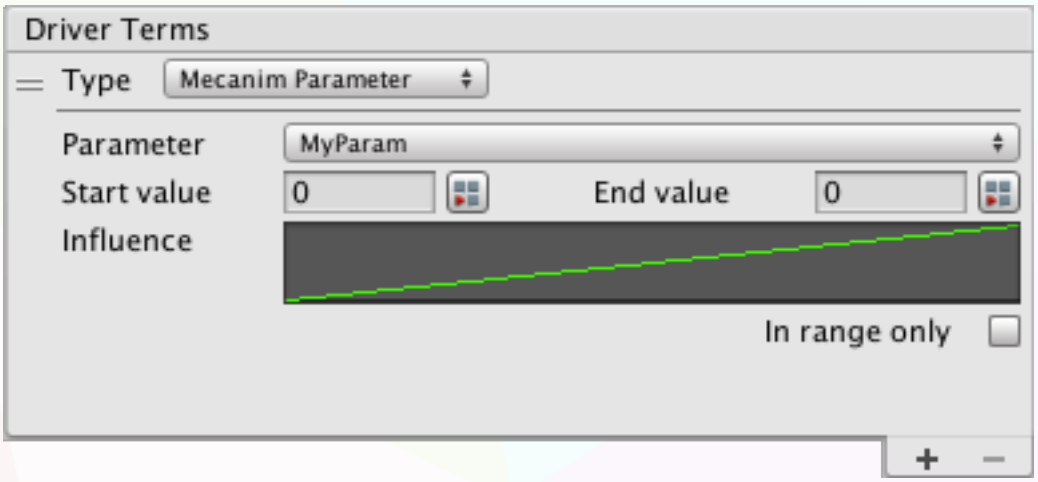
Distance to Point-type Driver Term

First, you may notice that the *Channel* property has been replaced by an *Axes* one. The distance-based *Driver Terms* may use any combination of axes, allowing you to ignore certain dimensions and only measure the distance within a single line or 2D plane if you so desire.

Lastly, the *Distance To Point* has its *Target* as a fixed *Vector3* point, whereas *Distance To Object* measures the distance to a target object's *Transform*'s position.

Note though that when a *Distance To Object*'s *Target* object is used in local space (*Self*), the term will compare both objects' local coordinates, each within their own local space. This is undesirable when you need to measure the absolute distance between objects, and can lead to some very unexpected results. However, you can use this to compare objects' local positions. For example, perhaps you want to drive a shape based on how similar two characters' hands are placed. You could then use local space to measure the relative distance between *character 1*'s *Hand.L* bone and *character 2*'s.

Let's move on to the animation-based *Driver Terms*.

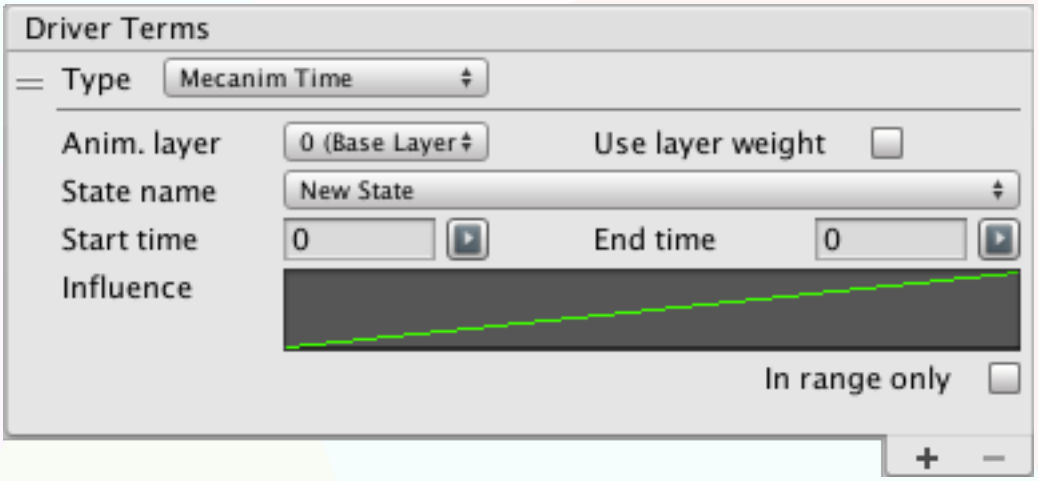


Mecanim Parameter-type Driver Term

Mecanim allows you to set special parameters that can be used to drive animations and transitions within its state machines. The *Mecanim Parameter Driver Term* allows you to read out these variables and use them to drive your curve. Unfortunately *Trigger* parameters are not supported due to an internal limitation, leaving:

- **Float:** A floating point number.
- **Integer:** A whole number.
- **Bool:** A true or false value.

Like the Sith, *Bools* only deal in absolutes, and therefore cannot smoothly transition from one value to another. With relation to the curve, *false* always maps to *0.0* on the horizontal axis, while *true* always maps to *1.0*. The *Start* and *End* value fields are therefore hidden when a *Bool* parameter is selected.



Mecanim Time-type Driver Term

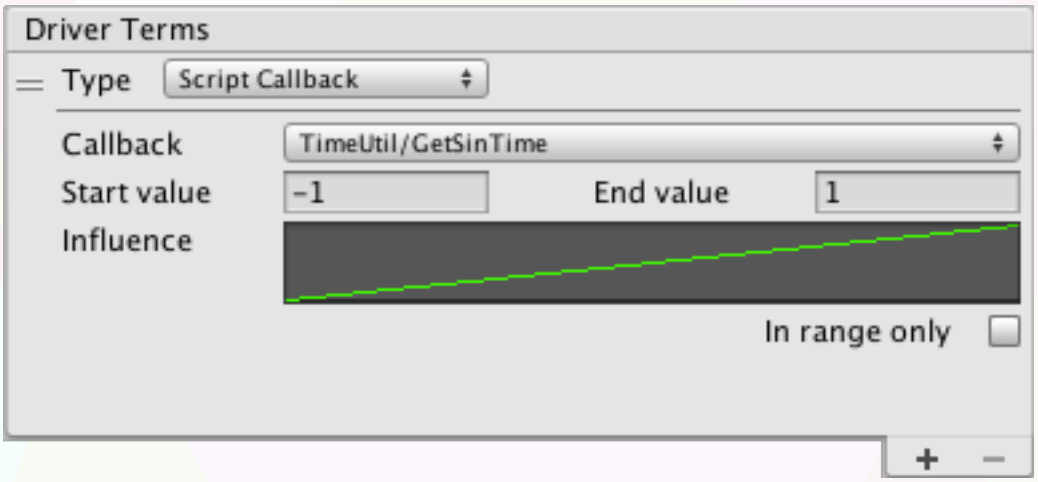
The *Mecanim Time Driver Term* allows you to drive a shape based on a *Mecanim* state's current time, for any one-off shapes that you may have.

The *Anim. Layer* refers to the *Mecanim* animation layer that the targeted state resides on. It's important to note that this value is stored by index rather than by name. This means that when you rearrange your layers' order you may need to update this value.
Every *Mecanim* animation layer has a *weight* assigned to it. The *Use layer weight* toggle let's you either keep this weight into account and use it as a value multiplier, or ignore it. When using the weight, the value is multiplied by the parameter value before the influence curve is evaluated, not the final term's result.

The *State name* refers to the name of the state you wish to target. Note that this is not necessarily the same as the name of the motion that state represents.

As *Mecanim* animations cannot be played outside of play mode their value at edit-time will always be *0*. The "set from current value" buttons will therefore set their respective fields' values to the selected state's motion's duration instead.

The *Legacy Anim. Time Driver Term* is very similar, but lacks the *Anim. layer* property, and allows you to refer to animations by their clips' names.



Script Callback-type Driver Term

Last up is the *Script Callback Driver Term*. This lets you select a method from any of the components attached to this object, and use its return value to drive your shape. For a method to be a valid callback target it must:

- **Be a public method:** Private and protected methods are excluded.
- **Be an instance method:** Static methods are excluded.
- **Return a float (System.Single) value:** Methods returning any other type, including those that could be implicitly cast (like ints and doubles), are excluded.

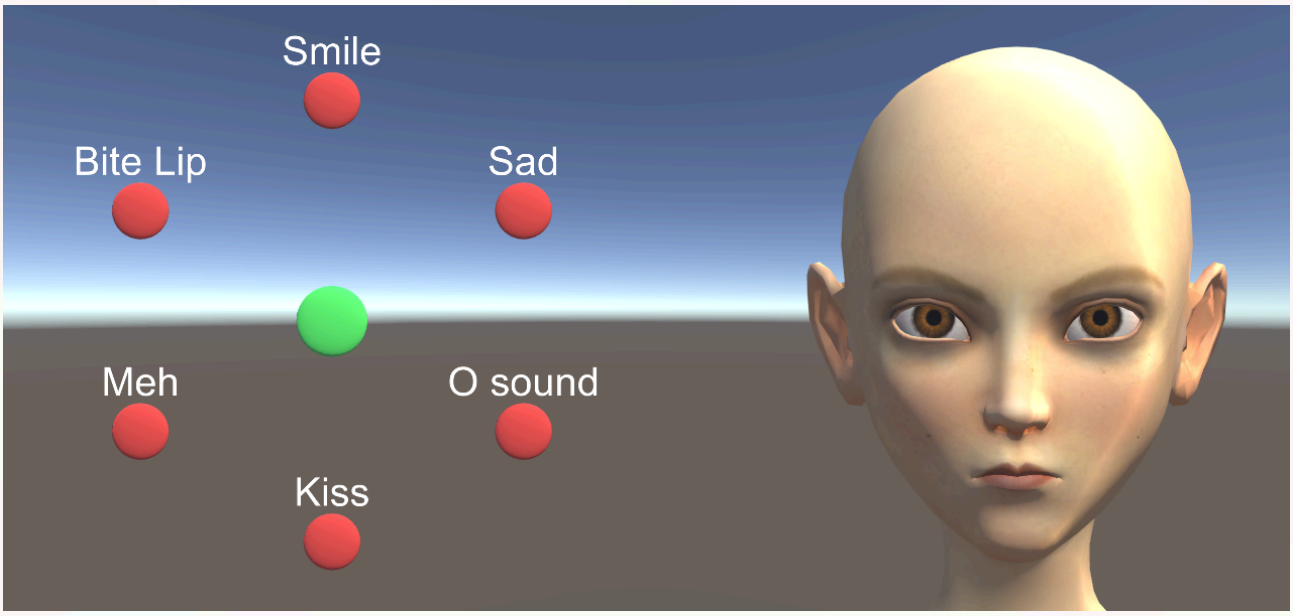
(Tip: The callback method is called every frame. Just like with `Update()` and friends, try to avoid any expensive calls like `GetComponent()` or reflection methods inside of it.)

```
1 using UnityEngine;
2
3 public class TimeUtil : MonoBehaviour {
4     public float GetSinTime(){
5         return Mathf.Sin(Time.time);
6     }
7 }
```

A Simple Example of a Valid Callback Method

5. Practical Examples

To close, I would like to show a couple of practical examples to illustrate some uses or functionality that may not be immediately obvious. All scenes mentioned here can be found in the asset's *Examples* folder.



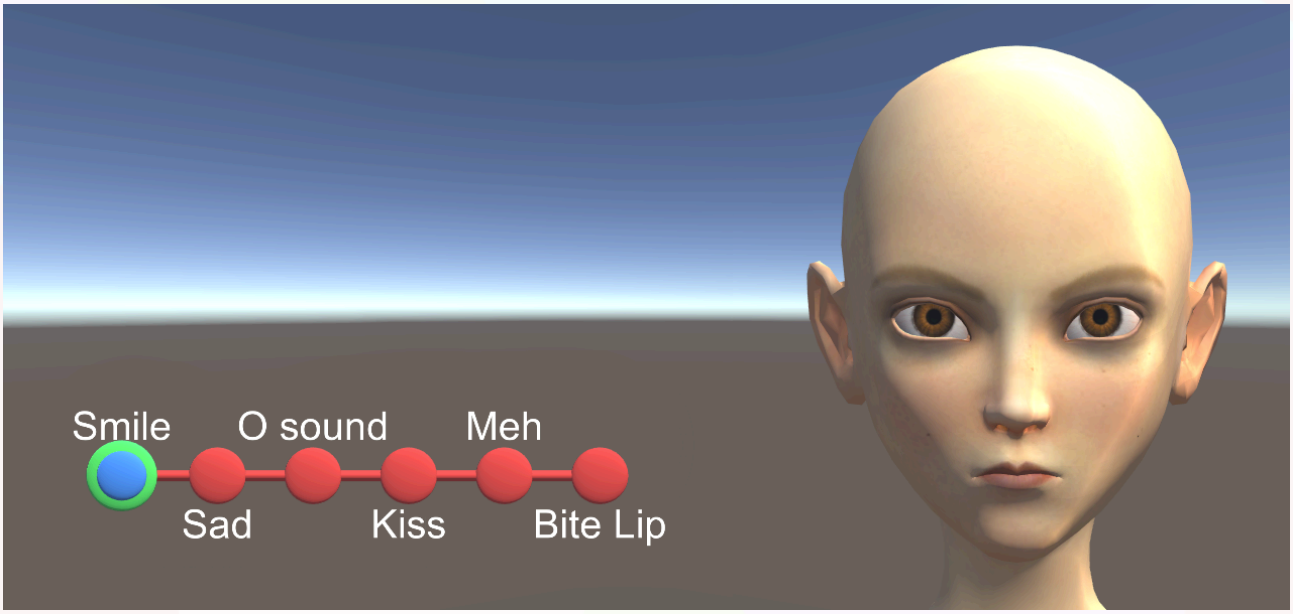
Circular Expression Controller

The *CircularController* scene shows an example of a circular expression controller. The proximity of the green disc to any of the red ones determines that respective shape's influence. This kind of layout can be particularly useful where smooth blending between shapes is required. A common example for this is mouth shapes with regards to sounds (i.e.: lip syncing). You could set up a control circle similar to this and simply tween the

controller object using a script based on the current text or other input, and the mouth will automatically blend nicely between all the different shapes and sounds.

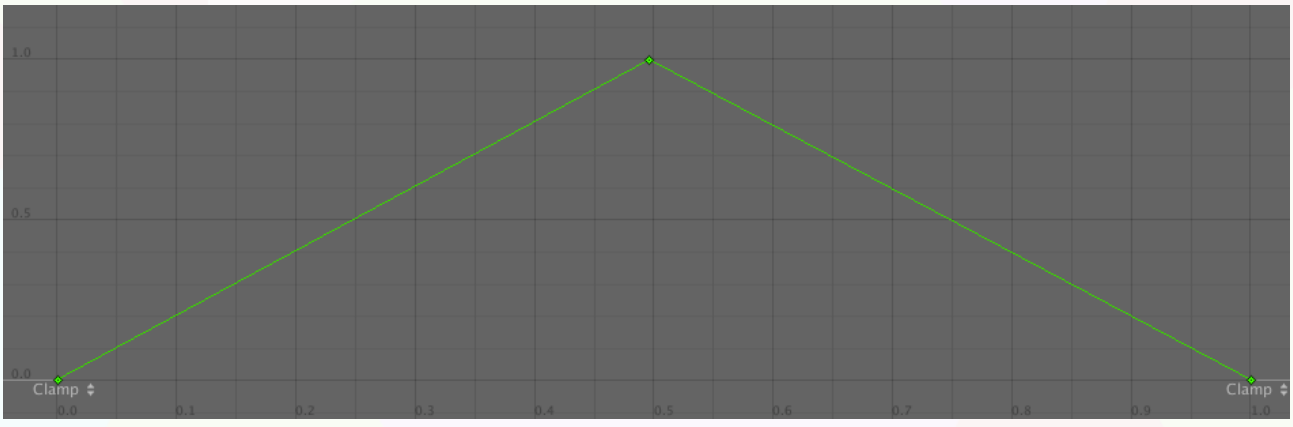
Note that for this kind of setup multiple *ShapeDriver* components are required; one for each shape. It is entirely up to you whether to attach them to the controller object or the target points, and both may make sense in different scenarios.

There's not much hidden magic here except for the realization that all targets are equidistant from the center, and it is that distance that is used as the *Distance Shape Drivers' start value*. Because of this, shapes on opposite ends never get affected at the same time, and smooth blending can occur in every direction. The setup could be easily expanded by adding a single "neutral" target in the center of the ring, though in some more expansive cases even more center targets may also make sense.



Linear Expression Controller

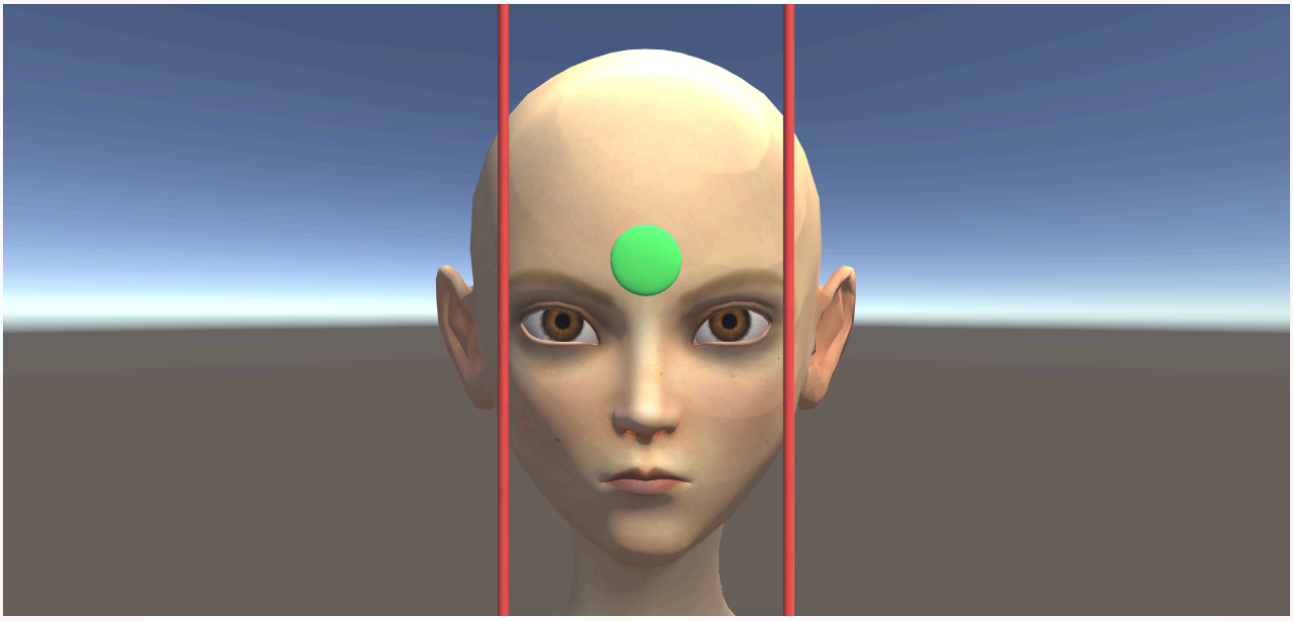
The *LinearController* scene shows an example of a linear expression controller. Rather than having a two-dimensional control rely on the distance between the controllers and targets, here we simply sample the controller's position on the X-axis. We do this once for every shape, with every shape having a range from the one before it to the one after (*n* has a range of *[n-1, n+1]*). For example, "O sound"'s range starts at "Sad"'s position and ends at "Kiss"'s. The influences are then blended using a custom curve that looks like this:



Linear Expression Controller

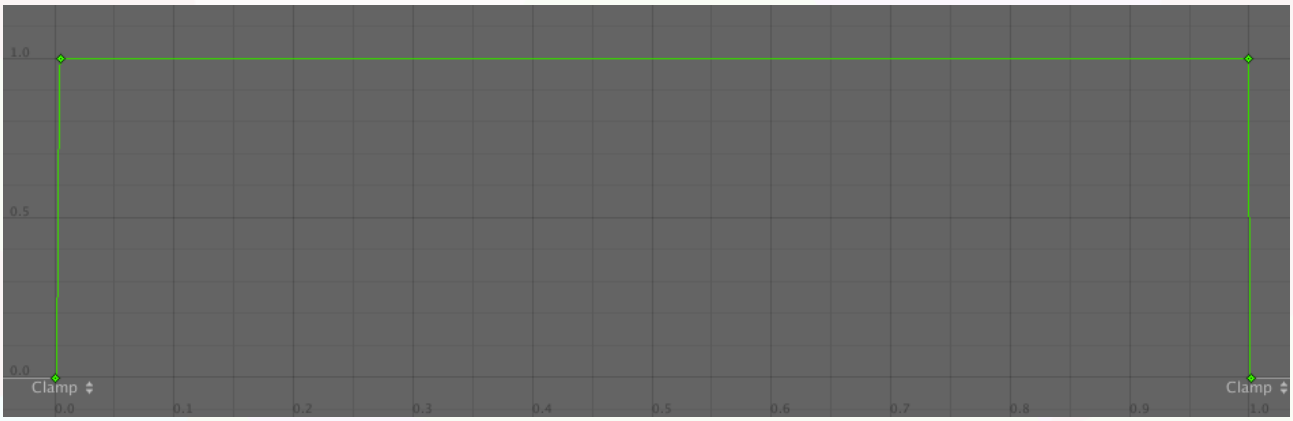
Because the range starts at the previous target's position and ends at the next's, the center of the curve aligns with the selected target. This essentially creates a simple crossfade effect between all adjacent shapes.

Example purposes of a linear controller setup like this can include character customization sliders and easy-to-use static posing tools. Note that this setup wouldn't really work for a full real-time expression controller, as blending from "Smile" to "Meh" for example would cause three intermediate shapes to appear.



Bounded Vertical Controller

Lastly, the *BoundedVerticalControl* scene shows an example of a *Shape Driver* that only functions within some bounds. The gist of it is quite simple: We drive the "Smile" shapes with a simple *Position Driver Term* on the Y-axis. However, we then multiply that term's result with a second *Position Driver Term*, this one sampling the value on the X-axis. The curve we use for the second term looks like this:



A Sharp Falloff Curve

If the second term's value is outside the specified range we want the resulting value to be *0*. Now you may be inclined to want to use the *In range* option, bit this simply ignores the second value instead of returning *0*. Instead, we therefore use a curve that plateaus at *1* pretty much all the way, but then very sharply declines to *0* at the edges. With a curve

wrap mode of *Clamp*, this both gives us the result we want, as well as the option of defining how strong we want our falloff to be.

6. Acknowledgements

The model used in the example scenes is a modified version of “Sintel Lite” (http://www.blendswap.com/blends/view/7093) by Ben Dansie (http://www.bendansie.com/), released under the CC-BY license (http://creativecommons.org/licenses/by/3.0/).

A custom rig, weighting and blend shapes were added for exemplary purposes.

Many thanks to Ben and the Blender/ Sintel team (http://www.blender.org, http://sintel.org) for creating and sharing this character!