AFTER EFFECTS EXPRESSIONS *BASICS*

INTRODUCTION

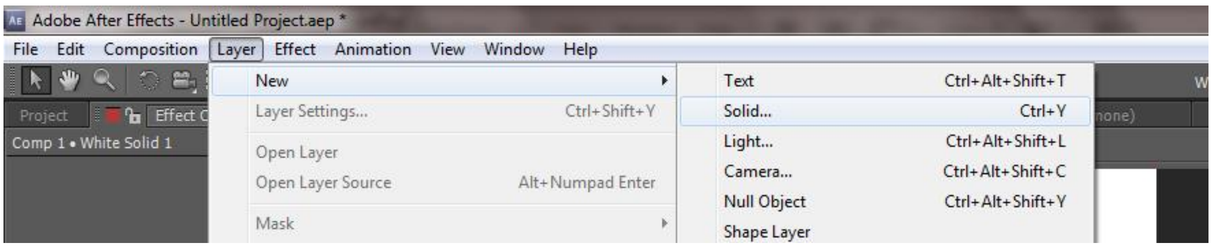
Hello, and welcome to *After Effects Expressions Basics.* I'm David Alex, VFX Artist and big fan of After Effects, and I'm going to walk you through the basic concepts of using expressions in After Effects.

In this document, we'll be looking at what expressions are, how they work and how to use them properly. By the end, you should not only be able to use some of the most common and useful expressions but also have the understanding required to expand your knowledge to an advanced level.

I'll assume that you've never used expressions before to make explanations and ambiguous areas a bit easier to work with. If you've seen a section before, feel free to skip it.

I'll also assume that you have a reasonable amount of knowledge on After Effects. At least know your way around the app, the panels, creating layers, effects and keyframing.

Before we get started, I'd like to clarify some syntax and words I'll be using::

- Rather than filling this document with screenshots (and making it bigger), going through menus like this:

Adobe After Effects - Untitled Project.aep *
File Edit Composition Layer Effect Animation View Window Help
New
Layer Settings... Ctrl+Shift+Y
Open Layer
Open Layer Source Alt+Numpad Enter
Mask
Text Ctrl+Alt+Shift+T
Solid... Ctrl+Y
Light... Ctrl+Alt+Shift+L
Camera... Ctrl+Alt+Shift+C
Null Object Ctrl+Alt+Shift+Y
Shape Layer
Project
Effect C
Comp 1 • White Solid 1
Will be shown like this: *Layer > New > Solid...*
- Shortcuts will be written in brackets ( ).
- **Property** refers to Opacity/Rotation/Scale/Position/Anchor Point... basically the properties of a layer, though sometimes I'll refer to a control with an expression as a property (for ease of understanding).
- **Initial Value** would be the value typed into the box of the property. This is not what the expression produces. It becomes confusing referring to this, but this should make things easier to understand.
- **Final Value** refers to the value produced by the expression, or rather the value that the property has in the end of all calculations.
- **Expressions** will appear in a box like this::

```
If(thisLayer.numKeys>2) {loopOut("cycle",2)} else {value}
```

That's about it.

Also, I'd suggest that you make friends with the Help system (if you haven't already). Go to *Help > After Effects Help* and check out the system (or *Help > Expression Reference).* We can use this document alongside the Help to explain stuff, and it'd be good to refer you to some of the things there. Though some things are hard to understand there, I'll try explain them here.

If you’re working in CS5, you’ve probably noticed that the Help system is online now. If you don’t have a constant connection, you can download the Help from the site.

http://help.adobe.com/en_US/aftereffects/cs/using/after_effects_cs5_help.pdf

It’s a PDF about 30MB. It’s updated quite frequently and contains bookmarks to ease navigating through it.

I’d also like to mention that this book is based off examples, rather than teaching facts, which I find is an easier way to understand something. To that effect, we won’t use project files, we’ll build them. In this way, we can see the whole process and heck might even learn something new other than expressions. I’ll keep things simple and describe the steps as best I can.

All right, having said all that, we’re ready to get started!

WHAT ARE EXPRESSIONS?

An expression is a piece of code that's inserted into a property to calculate its final value. These calculations can range from simple addition or multiplication operations to complex functions that vary with other properties and controls and even with time. The part that makes expressions fun and extremely useful is that they can be used to connect properties and controls to one another, which we'll definitely have to go through.

The Adobe documentation has an interesting way of explaining what expressions are (page 609 in the PDF, *About Expressions* in Help system):

"When you want to create and link complex animations, but would like to avoid creating tens or hundreds of keyframes by hand, try using expressions. An expression is a little piece of software—much like a script—that evaluates to a single value for a single layer property at a specific point in time. Whereas scripts tell an application to do something, an expression says that a property is something."

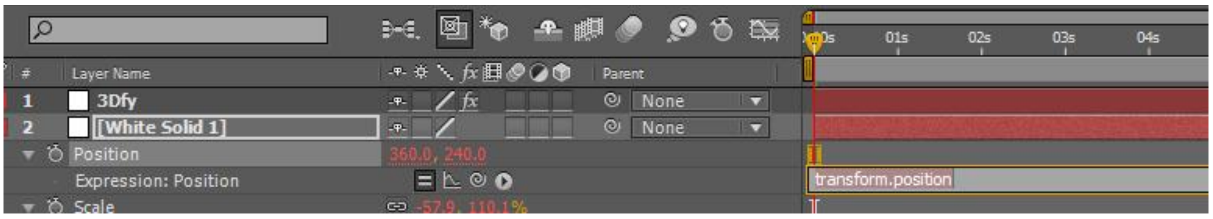
Not as complicated an explanation as one would expect from a Help system. It's nice that they get specific and say it evaluates to a single value for a single property at a specific time. Which we can interpret as meaning it can change over time.

Also, it's important that at this point I mention that expressions should only be used to either do something that keyframes cannot or do something that would require too many keyframes. There's no need to add an expression to make something move across the screen when just 2 keyframes can very easily do that job.

But I'm sure the most important question you're asking is...

How does one apply expressions?

Any property or control with a stopwatch can have an expression, which is pretty much every control in After Effects. In order to add an expression, simply **Alt+Click** on the stopwatch of the property:



Position 360.0, 240.0
Expression: Position
Scale -57.9, 110.1%

```
transform.position
```

A few things happen when you do this:

1. Below the control, you get "*Expression: ~*" referring what it's applying to, as well as a few buttons. We'll look into their uses as we go along.

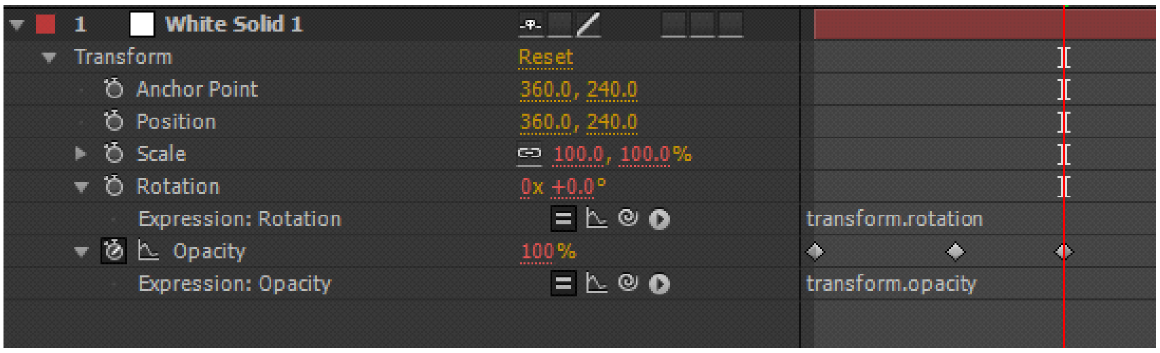
2. In the timeline, a text box appears (which we can refer to as the **Expression Box**) and it immediately has a code that refers to the current control it's applied to.
3. The numbers of the control turn red, which is a good indication of whether an expression is applied to something if you're not in the timeline (like in the Effects panel).
4. A little triangle appears on the far-left, which toggles the appearance of the expression box.

When you alt+click a control, you immediately get a code that refers to that control. If you were to click away at this point, the expression would be applied but nothing would really change because we've just told the control to use itself to produce its value, or to put it better, to use its initial value as its final value.

So if you were to alt+click on Scale, you'd get *transform.scale* and Opacity *transform.opacity.*

This is a good way to remind you what controls are referred to in the expression world. If, say, you wanted your scale to be equal to the opacity, you'd simply alt+click on the Scale's stopwatch and insert "*transform.opacity*" as the expression. What we're telling AE is "take the transform property of opacity and make it the value for this control".

There's something else you should note as well:



1 White Solid 1		
Transform	Reset	
Anchor Point	360.0, 240.0	
Position	360.0, 240.0	
Scale	100.0, 100.0%	
Rotation	0x +0.0°	
Expression: Rotation		transform.rotation
Opacity	100%	
Expression: Opacity		transform.opacity

As you can see above, *Scale, Rotation* and *Opacity* have expressions (the red digits), however the stopwatch for Opacity is active. This is because it has keyframes.

The stopwatch still works as usual. If you click on it (without holding Alt), you'll apply keyframes as usual. We'll go into working with keyframes a bit later, but let's get through the basic basics first.

It's also important to note that **expressions are CASE-SENSITIVE**. If you had *transform.Rotation,* you'd get an error. Keep this in mind (it get annoying sometimes).

Expression vs Initial Value

So let's start with an example that, although very simple, will explain a very important point.

Create a new comp with whatever dimensions you like. I usually go with *720 x 480 25fps Square Pixels 10 seconds* for my experiments. Go ahead and add a Solid with a different colour from the background.

Head down to the opacity (T) and we'll Alt+Click and type in this simple expression:

```
50
```

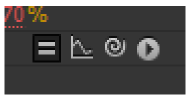
I warned you it'd be simple.

So click away and see what happens. Our final value (in red) is now 50. You'll notice that we can't directly tell what the initial value was. So click on the red text and you'll notice that the initial value is still 100 (remember initial value is what we type in). Type in something like 20. Click away and you'll notice that the final value is STILL 50!! Try animating the opacity with keyframes... yep, exactly!

So what are we saying here?

Expressions always override the Initial Value applied.

This means that any control with an expression applied to it will use the expression's result rather than the initial value, because otherwise it'd be pointless to have the expression, right?



Initial value, however, isn't completely ignored. If you were to disable the expression by clicking the = sign, or alt+clicking the stopwatch again, you'll notice it doesn't reset to the default value but goes to the value we last typed in as Initial value in the experiment above.

However, expressions can use the initial value as part of its calculation to produce another value that varies with the initial value. This is best explained with, yes, an example.

Using “VALUE”

Expressions have a simple way of referring to the initial value, and that is: *value*

VALUE, in simple terms, refers to the initial value that you type into the control. We can refer to this initial value in two ways:

1. The long way (*transform.opacity* if we’re in opacity)
2. The short way (*value*)

I always take the short way mainly because I don’t have to remember code and also it makes copy-pasting the expression simpler.

Head back into the Opacity and delete any keyframes and set the initial value back to 100. If you alt+click on the stopwatch, you can remove the expression. Alt+click again to apply this expression:

```
value-20
```

The first thing you’ll notice when you click outside the expression box is that our final value (in red) is now 80. If we were to click and scrub the values, as soon as you let go you’ll notice that whatever initial value you’ve set will be taken and have 20 subtracted from it.

Before you start questioning the practical uses of this consider a scenario:

You have opacity animation of 0 > 100 > 50 > 0, but you found these middle values were too high, and would like to reduce the range by 21 percent, it’s as simple as just heading into the expression and typing value-21 *, and what you’ll get is (0-21) > (100-21) > (50-21) > (0-21). Since opacity can’t be less than 0, the 0 keyframes will remain 0, while the others become 79 and 29 respectively.*

If this were Scale, we’d have the 0 keyframes being -21, in which case our object would flip!

So, don’t underestimate the power of value. When it comes to creating animation presets, it’s a very important function.

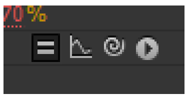
Feel free to mess around a bit with different expressions like *value-(value-20), value/2 etc*

REFERING CONTROLS TO OTHER CONTROLS

This may seem like a big jump from *value*, but it's actually a very simple process.

Expressions can be used to refer one control to another. Like say, to have Scale equal to Opacity at all times. We've covered this example already. But what happens if we wanted the Scale of layer 1 to be equal to the scale of layer 2, or better yet, half!

It's not as simple as *transform.scale* because this would only refer to the layer's own scale. Here's what we'd do::



Remember the four buttons that appear when you apply an expression? The first one activates/deactivates the expression, the second one visualizes it, and the third is the pick whip. The last one there is a menu which acts like a shortcut to applying expression functions.

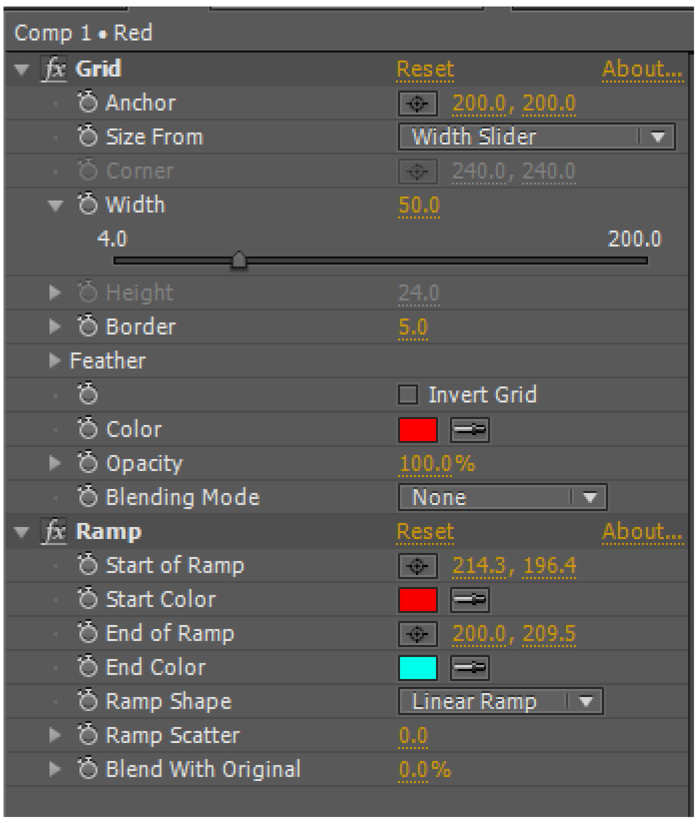
What we're interested in here is the pick whip.

The pick whip is a drag-drop control that we use to refer to another control. It works like the Parenting pick whip. What AE does is it takes whatever has been pick-whipped and turns the control we're picking from into what we're picking to.

An expression is automatically generated which is just one of the best things about it.

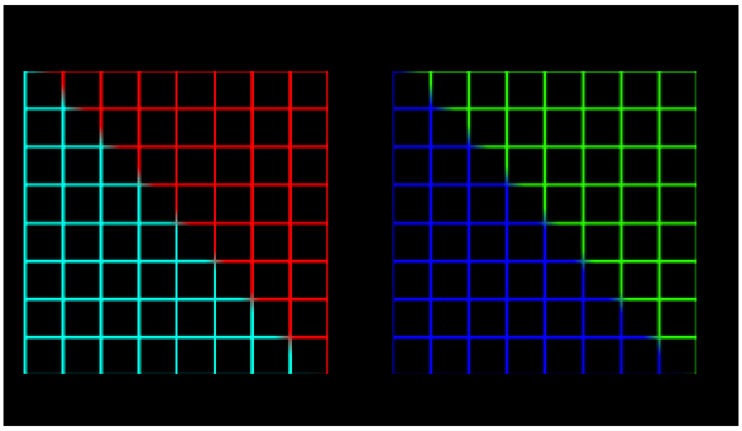
In order to show the power of the pick whip, let's build a Comp for us to experiment in with all this stuff.

What we'll do is create two square solids (400 x 400) and apply Grid + Ramp so we can differentiate the two and see the effects applying:



Comp 1 • Red

Grid	Reset	About...
Anchor	200.0, 200.0	
Size From	Width Slider	
Corner	240.0, 240.0	
Width	50.0	
	4.0	200.0
Height	24.0	
Border	5.0	
Feather		
	Invert Grid	
Color	🟥	
Opacity	100.0%	
Blending Mode	None	
Ramp	Reset	About...
Start of Ramp	214.3, 196.4	
Start Color	🟥	
End of Ramp	200.0, 209.5	
End Color	🟦	
Ramp Shape	Linear Ramp	
Ramp Scatter	0.0	
Blend With Original	0.0%	

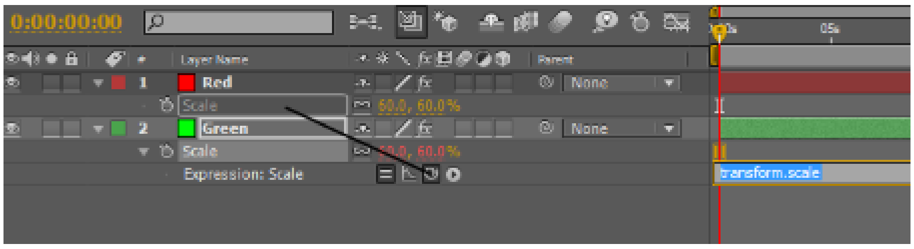


Notice that the Ramps' points are extremely close to one another.

I've set both their scales to 60. I'll call the one on the left "Red", and the one on the right "Green".

So, in order to get the pick whip working, we need to have the controls we want to pick visible. So, I'll select Red and Green and press **S** to show their scales only. I'll then alt+click on Green's scale. I'll get *transform.scale* as expected.

Now, I'll click and hold on the pick whip and drag it to Red's scale::



Ensuring that we get that grey box around Red's scale, when we let go we get this expression in the box:

```
thisComp.layer("Red").transform.scale
```

Now, our Green layer's scale is tied to the Red layer's scale. If we scrub the scale of *Red*, you'll notice that the Green follows suit. This is the most basic use of referral. It'd be fun to go ahead and add */2* at the end of this expression, or even *50, but let's look a bit closely at the expression above.

As you remember from before, we have *transform.scale* involved, however we have a little code before it. This code *thisComp.layer("Red")* is simply telling AE that:

In this Comp, go to the layer called "Red" and take it's scale.

Referrals have a specific syntax. First we refer to the comp we want, then the layer we want, then the property we want. Each step is separated by a full stop:

COMP.LAYER.PROPERTY

You don't necessarily have to remember this because the pick whip does all this work for us. All we have to worry about is whether the control we want to pick is in view. Though you can think of it of leading the expression to the control as if it were starting from the Project panel, moving into the timeline, and looking for the property you want.

So what if you wanted to pick whip to an effect? What if we wanted the other colour of the Green to be the other colour of the Red (which is Cyan)? Well, this is simple enough.

As before, we need to be able to see the control so we can pick whip it. In terms of effects, there's two ways we can view the controls.

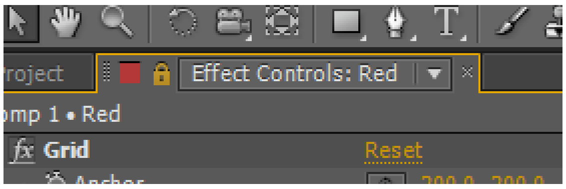
1. Press **E** to view the effects applied to that layer, as well as the effects' controls.
2. LOCK the Effects panel.

In this case, we are pick whipping an effect to an effect, so we can use both of these methods (if we used one, we'd have a lot of things showing in the timeline and have a long way to pick whip, if we used two we'd have to create a new Effects Viewer and that would crowd up our workspace).

It's just the way I prefer it, but do as you see fit.

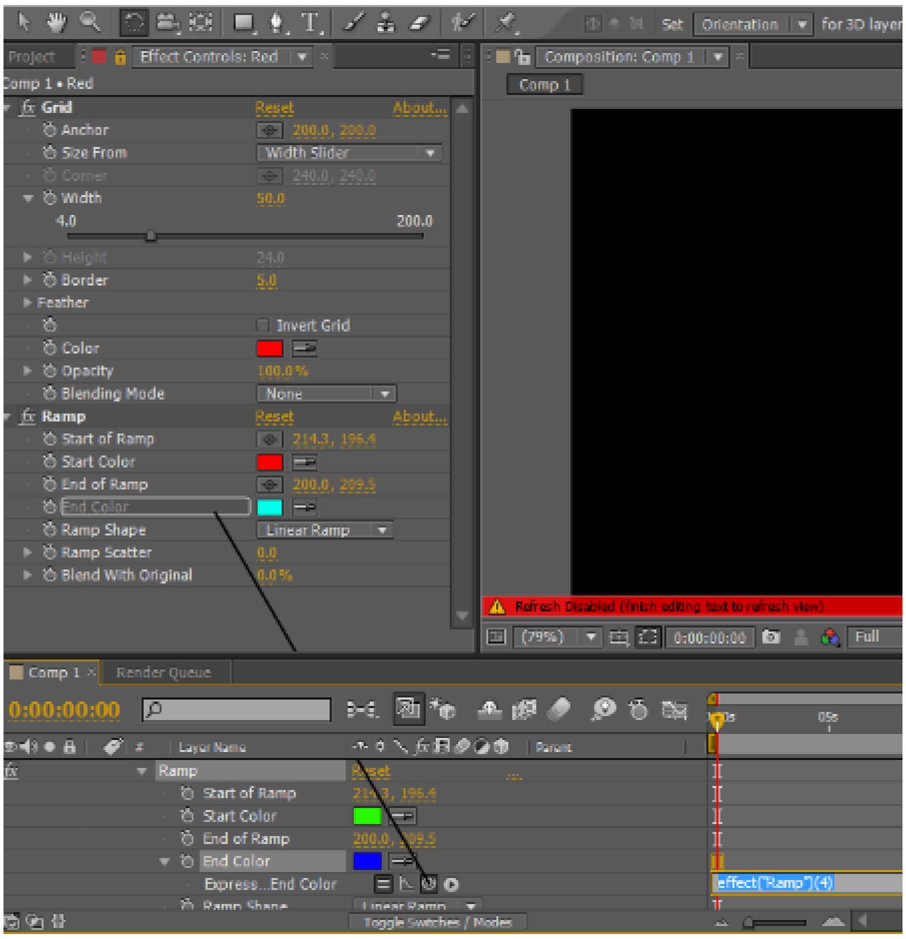
Since I can't pick whip from the Effects panel, I'm going to lock the view with the Red layer's Effects and use **E** on the Green layer's effect in the timeline. So, let's do exactly that.

Click on the Red layer and click on the LOCK at the very top of the Effects panel:



What this does is make it such that if we click on another layer, we'll still be able to see the Red layer's effects. This is because as soon as we click on the Green layer, we'd end up seeing its effects. So this ensures that doesn't happen.

Next, I'll click on the Green layer (confirming that I'm still seeing Red's effects) and press **E**. I'll drop down the Ramp and I can see **End Color**, which is Blue. I can then Alt+click on the stopwatch then pick whip the **End Color** of the Red layer above:



Quite a distance, but the job is done.

What's going to happen is this appears:

thisComp.layer("Red").effect("Ramp")("End Color")

We've had the same referral process happen, but instead of *"transform.scale"* we now have *"effect("Ramp")("End Color")"*

So in this case, we've told it that:

In this Comp, in the layer "Red" there's an effect called "Ramp" and I want the control called "End Color".

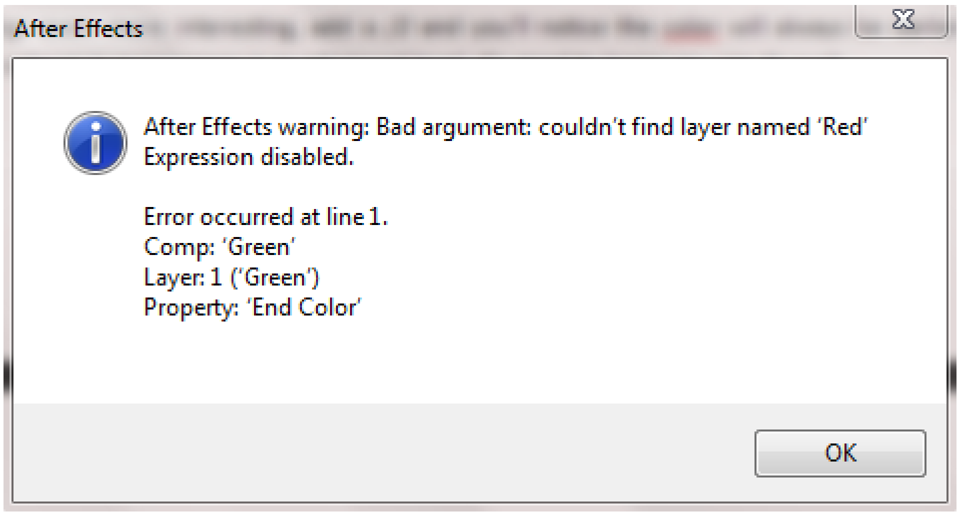
And AE does exactly that. Now we have Cyan for our end colour. Best of all, we can change the color in the Red layer and Green will do exactly the same. Remember that if I try to change the color of the Green layer, it will stick to the Red layer's value because expressions override initial values (which we discussed earlier).

To make things even more interesting, add a */2* and you'll notice the color will always be darker. Though working with colors in expressions is an advanced level, it's good to know you can do this.

So what happens if I want to refer to a layer in a different comp?

If you're asking this question, you probably experienced pre-composing errors. If you haven't, I'll explain.

If you were to take Green and pre-compose it (*Layer > Precompose*), moving all attribute, you'd get two errors from each expression, one of them being:



```
After Effects warning: Bad argument: couldn't find layer named 'Red'
Expression disabled.

Error occurred at line 1.
Comp: 'Green'
Layer: 1 ('Green')
Property: 'End Color'
```

In this case, the **End Color** expression faced the error because it couldn't find the layer named "Red". The popup is also nice enough to show us what line in the expression the error occurred, what comp and which layer.

If we look back, we told AE:

```
thisComp.layer(“Red”).effect(“Ramp”)(“End Color”)
```

and what AE is trying to do is find this “Red” layer **inside** the pre-comp (because we’ve said “thisComp”). So obviously, it’s not going to find Red because it’s in a different comp.

So how would we fix this problem?

Press U to view all our expressions. (Alternatively, right-click and choose *Reveal Expression* Errors)

Now that we understand how referrals work we don’t have to worry about how to pick whip another timeline. Let’s just tweak the expression.

Simple, head over to the **End Color** expression and make the change to this:

```
comp("Comp 1”).layer(“Red”).effect(“Ramp”)(“End Color”)
```

Remember the case sensitivity!

Notice that the “comp” at the beginning starts with a small letter. We also put in brackets, quotes and the name of the comp that holds Red.

So in a comp called “Comp 1” is a layer called “Red” with an effect called “Ramp”. I want the value of “End Color” from there.

We’ve told AE this. Now it will find the control and the expression will be valid again!

After changing scale like this as well, we can head back to Comp 1 and see that if we play with Red’s scale, Green will update immediately even though it’s in a whole other comp.

So we’ve played with referrals and pick whipping, but I have a question...

What happens if I apply an expression to a control with 3 dimensions and pick whip it to a value with 1 dimension... or vice versa...?

Well, that’s when we start looking at arrays! Let’s look into that before we start talking about fun expressions!

ARRAYS

This section gets rather complicated. If you've been reading this continuously I'd suggest you get into AE and mess around with what you've learnt so far before getting into this section.

Before I start explaining, let's view Green's Position (P) and Opacity (T). As you can see, position has two values (X,Y) while Opacity has one. What would happen if we pick whipped them to one another??

Let's pick whip Position to Opacity, we'll get this:

```
temp = transform.opacity;
[temp, temp]
```

And do the opposite, pick whip Opacity to Position, we'll get this:

```
transform.position[0]
```

In the first instance we have *temp = transform.opacity,* which is simply a variable. We're telling AE that wherever it sees *temp*, it should take it as *transform.opacity.*

Right below that is *temp* twice in square brackets.

In the second instance, it's the usual transform expression but there's a zero in square brackets...

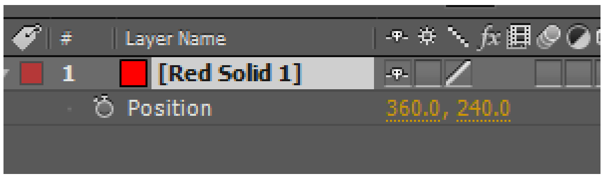
What exactly is happening here?

Although it may seem like things are getting rather complex, I think the sooner you understand what an array is, the easier and more fun the rest of this document is going to be.

What are Arrays?

All controls in After Effects are defined by arrays. An array is held in square brackets and values are separated by commas. This is how individual axes can be referred to.

Best explained as an example, a default solid in a 720 x 480 comp has a position of 360 x 240 (half and half). Which appears like this in the timeline:



Expressed as an Array, this would be [360,240]

If this object was 3D, it would be [360,240,0]

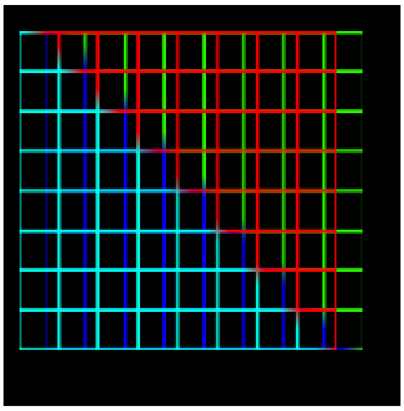
As you can see, we're doing [x,y,z] in the array.

In the case of opacity, which has 1 dimension, we don't need to use an array since [50] and just plain 50 will result in the same thing. However, arrays come in handy when we're working with, say, two objects we'd like to have relative position values.

For example, in the Boxes comp, we can select RED and hit Ctrl+Home to center it. Then we can head over to the GREEN pre-comp and apply an expression to its position:

```
thisComp.layer("Red").transform.position +20
```

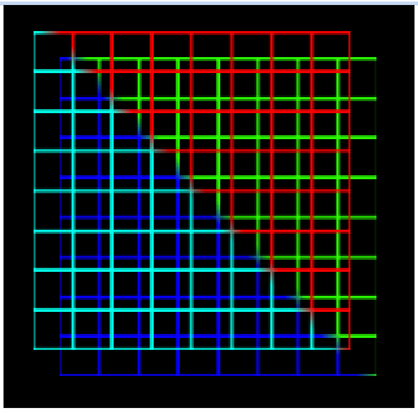
What you'd probably expect is the layer to be 20 pixels down and 20 pixels to the right, but only the latter is happening. Why is that?



This is because, for some very strange reason, AE is assuming that you're adding that 20 to ONLY the first dimension, which in this case is the X axis.

So in order to have 20 added to BOTH the X and Y axis, we'd do this:

```
thisComp.layer("Red").transform.position + [20,20]
```



What's happening now is we're having 20 added to the X, and 20 added to the Y. This is also great because we can go ahead and add different values for each dimension.

Array Dimensions

If you remember, I mentioned that arrays have from 1 to 4 dimensions. AE's Help has a great table that illustrates them (Page 620 in the PDF, Arrays chapter in the Help system)

DIMENSIONS	PROPERTY
1	Rotation (degrees) Opacity (%)
2	Scale (X, Y) Position (X, Y) Anchor Point (X,Y) Audio Levels (left,right)
3	3D Scale (X, Y, Z) 3D Position (X, Y, Z) Anchor Point (X, Y, Z) Orientation (X, Y, Z)

4	Colours (Red, Green, Blue, Alpha)

The colour one is pretty interesting. Keep in mind this doesn't work in the normal range (0-255) but rather a decimal scale (0-1). It's a bit tricky to co-ordinate but also a handy control.

We can now use arrays and do something like:

```
transform.scale+[20,0]
```

So in the example at the beginning of the chapter, we can see that since we have *temp* defined as a variable, our opacity will immediately be transferred to the position for both X and Y, coz AE assumes that's what you were trying to do.

```
[transform.opacity,transform.opacity]
```

Was it?

But what about the second instance when we pick whipped the Opacity and got another different expression?

```
transform.position[0]
```

In this case, we're referring to *part* of an array.

Referring to Dimensions

An array is split up in a weirdly offset manner. For this explanation, I'll use a 3D position array of [360, 240, 0].

Array referrals start from 0 and count in order of the dimensions. In the case of my example array, *[0] is 360, [1] is 240, [2] is 0.*

When we're writing an expression and we want to refer to a specific dimension, we type it out like this:

```
transform.position[0] or transform.scale[1] etc
```

In the above transform.position is referring to the X axis position, while transform.scale is referring to the Y axis scale. It's important to get used to the fact that this starts from 0, and the sooner the better.

So, in an interesting case, we can create a new null or solid and do this:

```
RedX=thisComp.layer("Red").transform.position[0];
GreenY=thisComp.layer("Green").transform.position[1];
[RedX,GreenY]
```

Here I've used variables and defined RedX as the X position of the layer RED, and GreenY as the Y position of the GREEN layer. At the bottom, we've told it to make X axis to be RedX, and Y to be GreenY, the variables we set earlier. Putting that array at the bottom works like when we just typed 50 into the expression box. We're telling it to ignore the initial value and make this array the values of the control.

I hope that makes sense...

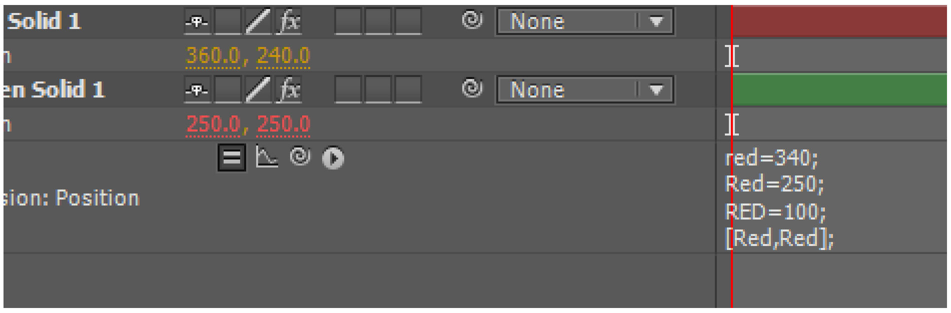
So, these, variables....

VARIABLES

Variables are words that you give a value and can later relate to within the expression. Like in math the way we'd use x and y as variables while solving stuff like angles, variables are used in the inverse way.

They're fun to use and pretty simple to understand. There are some rules though:

- Variables only work within the same expression, which means if you were to use one elsewhere, you'd have to define it again there. They are mostly used for easy editing later and to keep things nice and neat.
- You can use any word as a variable, except those reserved by After Effects like comp, Comp, time, position etc. To avoid conflict, use random words or words that are unlikely to have been used already. I also tend to name them after what they're doing so I don't come back and wonder "what the heck is this?"
- Also note that variables are case sensitive. "Red", "red" and "RED" are all considered very different things.



```
red=340;
Red=250;
RED=100;
[Red,Red];
```

Not only didn't I get an error above, but it's very easily taken Red which is 250.

You'll also notice that you may define a variable but you don't have to use it!

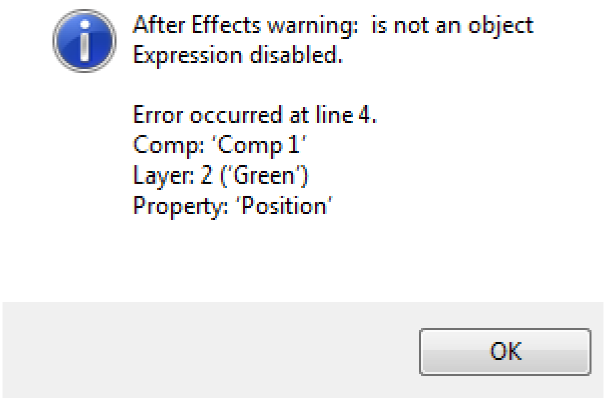
Also notice the ';' at the end of each line.

This is there to separate the lines. In AE, you'll find that it does not take paragraphing.

If I did the RED/red/Red example:

```
red=340;
Red=250;
RED=100;
[Red,Red];
```

Without the colons I'd get this error:



```
After Effects warning:  is not an object
Expression disabled.

Error occurred at line 4.
Comp: 'Comp 1'
Layer: 2 ('Green')
Property: 'Position'
```

AE gets really confused, because this is what it sees:

```
red=340Red=250RED=100[Red,Red]
```

And this won't make any sense, so you'll get an error. Though some of the time it usually works without colons, it's best to play it safe and add them.

Another interesting thing about variables is that you can define an array as a variable. So you can have something like this applied to position/scale:

```
Purple=[20,10,45]
value+purple
```

Shortcuts for Defining Variables

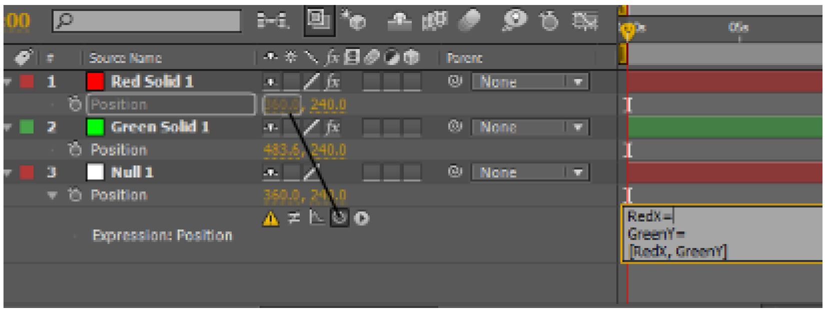
Yes, there's some shortcuts when it comes to applying variables.

Remember a few pages ago I mentioned that the pick whip had more features. Well...

In the case of the RedX, GreenY example, what I can do is this:

```
RedX=
GreenY=
[RedX,GreenY]
```

Then, I can put the cursor at the end of the first line, then with Red's position showing I can pick whip the X VALUE, not the whole control, just the VALUE:



```
RedX=
GreenY=
[RedX, GreenY]
```

What's going to happen is that the referral code for Red Solid will be added to **where the cursor is!** Which is awesome! So all I have to do now is add the ; and then move down to GreenY and pick whip that as well (not forgetting the ;).

This is the other great thing about the pick whip. Not only can you select properties, but you can also pick specific values/dimensions and have them added to the expression. Please note that if you're not editing in the expression box and you pick whip, it will replace your whole expression!

So, we're done with all the explanations, let's get in and have some fun!

SOME EXPRESSIONS

wiggle

One of the most common and most used expression is WIGGLE.

The syntax is as follows:

wiggle(frequency,amplitude)

Wiggle works a bit like the Wiggler, except it doesn't use keyframes and it is active throughout the duration of the layer.

Wiggle takes the *initial value* (what you typed) and randomizes it by a defined amount at a defined frequency. This means that it can wiggle around a given area, and works with keyframes!

In order to apply wiggle, you just write wiggle and type in a frequency and amplitude (or amount) in brackets. That's it.

The frequency is in amount per second, and amount depends on the control it's applied to (position would be pixels, rotation would be degrees etc). There is some confusion, however, on how wiggle uses these values.

Take *wiggle(5,200)* applied to a position property. What we'd expect is that 5 times a second it'd move 200 pixels in one direction and then 200 pixels in another direction for the next 5 frames, which is not what actually happens. Wiggle doesn't have a set direction, but you can rest assured that if you took all the number of pixels moved in a certain time and did the necessary math, I'd add up to that 200 pixels. Just know that 5 times a second it will have moved 200 pixels in random directions.

So if you're looking to have a layer move 200 pixels in a certain direction, I'd suggest typing in 400 or 500 as the amount. That extra amount should compensate.

Once you get the hang of wiggle it's very fun. It's especially great for adding camera shake to footage. Also remember that wiggle happens around the original value. So if you had opacity of 0 and wiggle(2,40), you'd get values ranging around 10-20% and just reaching 35 on some occasions. But if you clicked on the red text and changed the initial 0 to say 40, you'd get values ranging 20-70 percent, touching the tens and mid 70s every now and then.

Wiggle is very useful for adding camera shake. Adding *wiggle(.5,200)* adds a nice slow camera shake while *wiggle(1,50)* would really shake things up. Visit expression reference, you'll find a nice expression on how to wiggle a layer for a specific time.

Also in the PDF at page 641, it's explained that wiggle randomises each dimension on its own, meaning it produces an array. This makes sense because if equal values were applied to both X and Y, the layer would just move diagonally rather than randomly.

In order to have a certain control such as Scale to have equal wiggle for both X and Y, or position, we can use a very nice array and variable combination:

```
X=wiggle(.25,40)
[X[0],X[0]]
```

If we wanted to have them have different rates of shake?

```
[wiggle(2,200)[0],wiggle(.5,200)[1]]
```

Or to just keep things neat:

```
w1=wiggle(2,200);
w2=wiggle(.5,200);
[w1[0],w2[1]]
```

Which is why I found it important to show arrays and variables early!

loopOut

One of the most useful expressions to me, and shockingly enough I've never seen it mentioned in any tutorial I've seen.

loopOut is a rare expression that not only uses keyframes but also depend on keyframes to produce values.

Head over to "Red" Rotation (R) and create two keyframes of 0 to 135 about 1 second apart.

Next, add this:

```
loopOut("pingpong"2)
```

What happens is the two keyframes will now bounce between one another as time goes by. So it'll go 0 -135 then 135-0 then back again. It'll continue back and forth until the end of the layer.

The syntax for loopOut is:

```
loopOut("type" , number of keyframes)
```

Type refers to the type of loop, and must be in quotes (otherwise it starts looking for a variable). Number of keyframes is how many keyframes counting from the last that will be looped. If you set it to 3, it'll loop the last FOUR keyframes. It's better to think that it's repeating the last 3 *movements*.

There's also various types you can use:

CYCLE will play to the last keyframe, then snap back to the first keyframe and start the animation all over again.

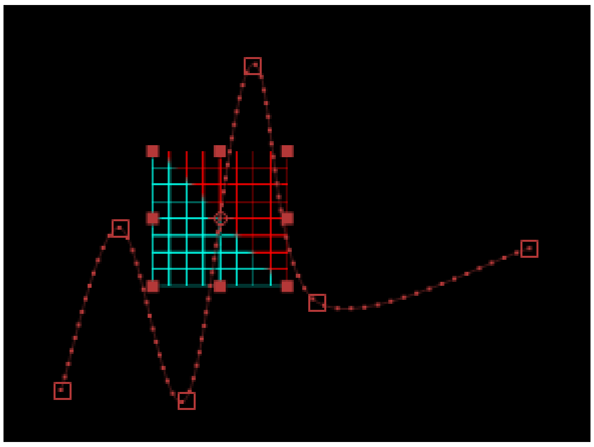
PINGPONG will play to the last keyframe, then play the keyframes backwards, then forwards again... as the name suggests.

OFFSET extends the animation. For example, if you had keyframes on position from point A to B then C, it'd play that animation then continue moving the object as though those same keyframes specified have been offset such that point A is now at point C and it starts to work like cycle.

This is useful for cases when you'd like to make an object move in a zigzag formation. All you do is move it the first 7 shape and watch the rest get added by offset.

CONTINUE is used without defining the number of keyframes (ie *loopOut("continue")*) It is used to make the object continue moving in its last direction in its last speed. It ignores all other keyframes.

Let's create another example.



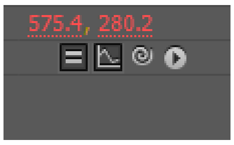
I've created six position keyframes for Red and set the keyframes to Rove Across Time so that they're evenly spaced (Select the keyframes, *Right-Click > Rove Across Time*)

I'm going to first apply this:

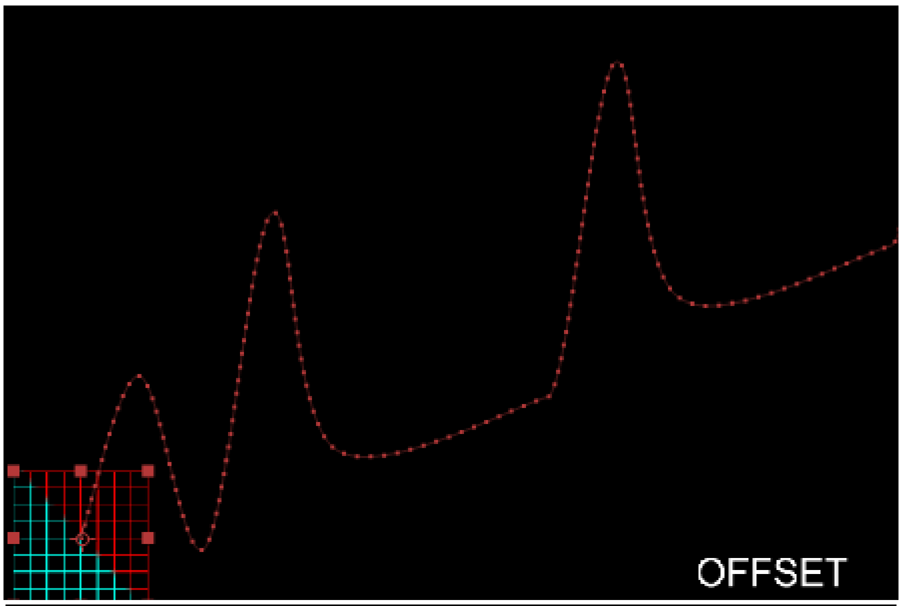
```
loopOut("offset",3)
```

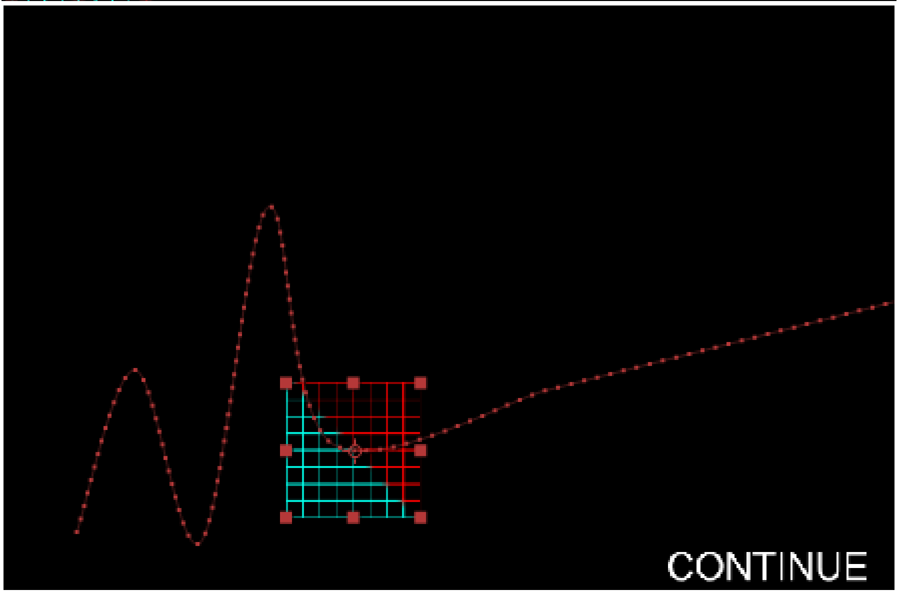
If you look at the image above, 3 would mean the last 3 *movements* (as mentioned earlier) so it's going to be the last horizontal move and the up and down movement.

In order to see what the animation is going to look like, we can use the 2nd button:



This will visualize the paths in the Comp viewer, drawing the lines like the ones used by keyframes to show the path the object is going to take.





I figured not to include **pingpong** and **cycle** because they'd look exactly the same. However, notice the difference between offset and continue.

This is great for making continuous animations like stuff bouncing around, throbbing opacity or scale, radar screens and so on. Try making a button that breaths with a few keyframes and the appropriate loopOut.

time

A handy tool used to animate an object is *time.* Time is represented by the current time in seconds, and also drops down to the decimal points.

Time is handy when it comes to animating stuff like rotation, evolution of the *Fractal Noise* effect and anything else for that matter.

Time works like a handy multiplier. If we went into RED's rotation and typed in this:

```
time*200
```

The object would rotate 200 degrees every second. So if you were looking for 1 revolution per second it's as simple as *time*360.* Since at 1 second, time is going to be 1, at 2 seconds it's going to be 2, and it'll go through all the values in between to the decimal point (1.1,1.2,1.3 etc) which all depends on the frame rate, but that's complicating things.

Head over and apply *Fractal Noise* to a layer and apply *time*200* to the Evolution. Simply alt+click on it in the Effects panel and the expression box will appear below in the timeline. Then add the expression. You'll notice that it'll animate for you continuously.

Now if you're thinking "*why don't I just put a 0 keyframe at the start and a 200 keyframe at the end?"*

The problem with this is that tweaking this range will mean you have to go to the end of the animation and change that last keyframe. The expression is also handy because you don't have to worry about re-calculating the value of the last keyframe should you want to extend the composition/layer.

Speaking of keyframes, it's pretty obvious that time is not affected by keyframes because it's absolute. If, say, you wanted to add some sort of offset to a rotation control, simply add that value in the expression:

```
(time*200)+45
```

Notice that it's a small 't', and that initial value is completely ignored.

Interestingly enough, we can also add 0-200 keyframes and apply *loopOut("continue").* Varying the space between them would affect the speed.

valueAtTime

This is another very interesting expression.

valueAtTime refers the expression to a specific point in time within an animation. The syntax for valueAtTime is:

```
valueAtTime(t)
PROPERTY.valueAtTime(t)
```

As you can see, we're using a dot there to separate the two, because now we're still using the referral method and referring it to a particular point in time.

For example, if we wrote:

```
valueAtTime(1)
```

In the above case, the expression would return the value of the property at the 1 second mark. Now this doesn't sound very interesting, but it becomes very fun when we involve *time* as discussed earlier. If we had keyframes, it'd show the position at 1 second and have it fixed there. So it's *kinda* using the initial value.

Ordinarily if we had Green moving from Point A to Point B and pick whipped Red's position to this control, we'd find that both are moving exactly the same (which only makes sense).

So what if we wanted to make Red have a slight timelag?

If we think about it for a sec, if we pick whipped this is essentially what is happening:

```
thisComp.layer("Green").transform.position.valueAtTime(time)
```

AE is simply using the value at the **current** time!

So what if we offset that time? For the sake of clarity and neatness, I've added a variable *GreenPos.*

```
GreenPos=thisComp.layer("Green").transform.position;
GreenPos.valueAtTime(time-2)
```

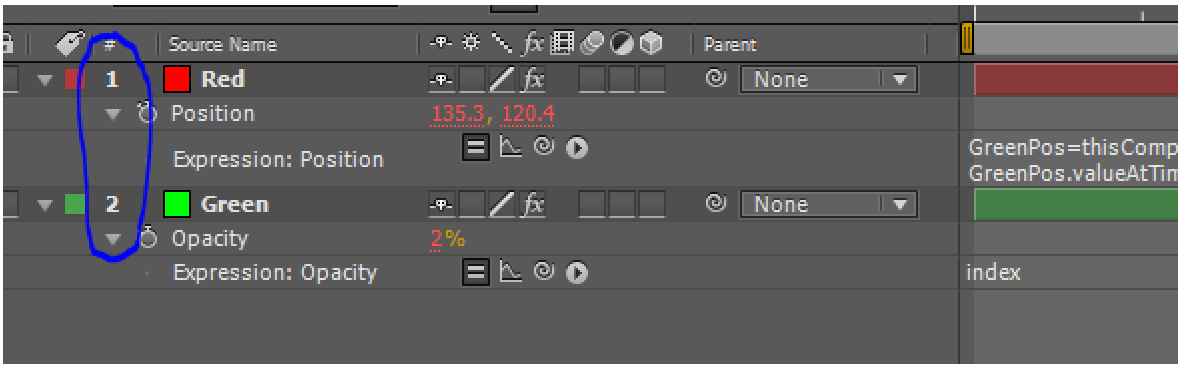
What's happening now is that it's going to take the value of *GreenPos* but using what it was 2 seconds before the current time. So, Red is going to be 2 seconds *behind* Green! We could also do +2 to make it ahead.

We'll use this later on to make some very fun animations, this along with the below expression.

index

Index is the number of the layer in order from the top. The top most layer in the timeline has an index of 1, then following is 2 and so on.

In order to prove this, create a bunch of solids (Ctrl+D if you like) and head over to the bottom layer's Opacity and simply have "*index*" as the expression. You'll see the opacity will be equal to the layer's number. Moving the layers up and down will show a responsive change:



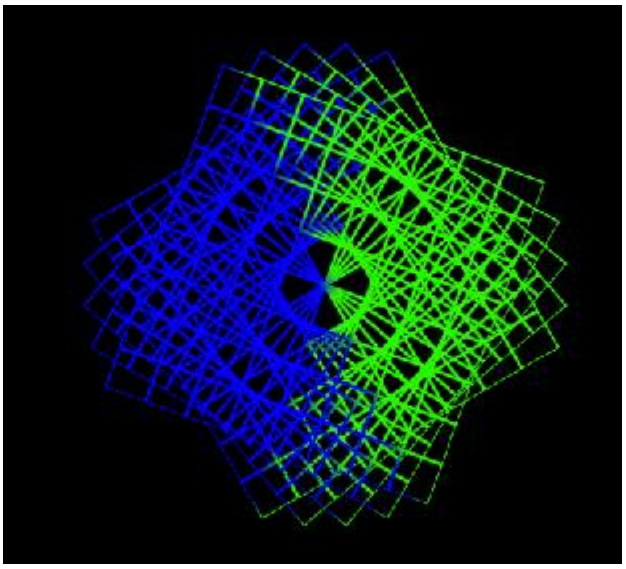
As you can see, Green is layer 2, so it has an opacity of 2.

So how can we practically use this?

Take the Green layer's rotation and type in:

```
Index*10
```

Now duplicate it about six times:



What's happening here?

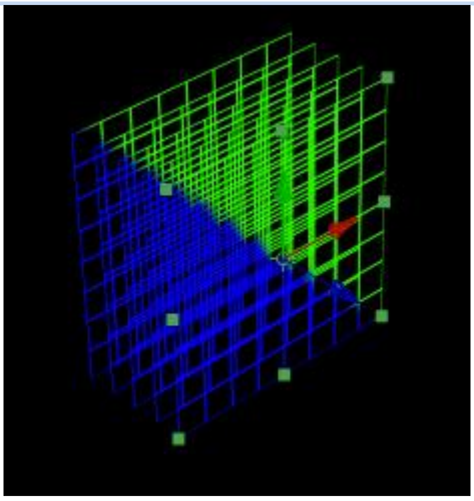
Layer 2 has a rotation of 20 (index of 2 times 10), layer 3 has a rotation of 30 etc. As the layers add up, their position in the timeline determines the rotation. How cool is that?

This can work with anything. Delete the duplicates and apply this to the Position. Except this time we want it to affect the Z axis only. So, do this:

```
value+[0,0,index*5]
```

Or to keep things neat we can do this:

```
offsetter=index*30;
value+[0,0,offsetter]
```



(Viewed from **Custom View 1**)

Sometimes if you have a layer who's index is high like 10, you'll find this offset starts off far. In order to fix this, we can just subtract index by the value of the very first layer:

```
offsetter=(index-10)*30;
value+[0,0,offsetter]
```

Another fun step can be to do this to opacity:

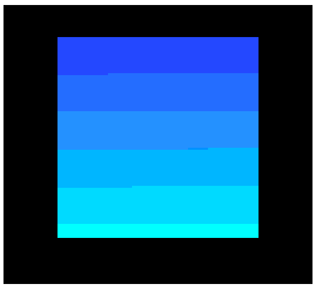
```
value-(index*10)
```

Fun times with expression indeed. Now, let's look into building a fun expression!

EXAMPLES OF EXPRESSIONS

I: SQUARE SHAKE AND TURN

Create a new square solid in a new comp. 200 x 200 pixels. Add a colourful Ramp and Posterize effect to add some perspective:



Now, create 3 rotation keyframes: 0 > 45 > 15.

Add an expression to this:

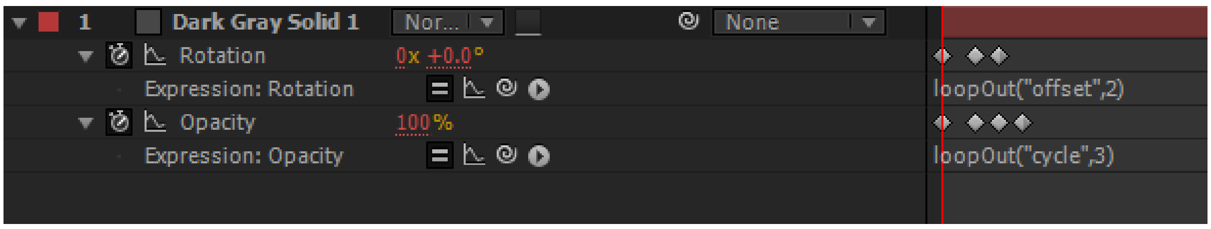
loopOut("offset",2)

If you remember, offset will repeat the keyframes in an offset manner from the last. This means the final values will go 0 > 45 > 15 > 60 > 90 etc

It will continue indefinitely.

As a tweak, let's add opacity keyframes of 100 > 40 > 100. I want this effect to throb, but I want it to hold 100 for a while before it fades out again. In order to do this, I'll add another keyframe for 100 a few seconds later (so we have 100 > 40 > 100) then apply:

loopOut("offset",3)



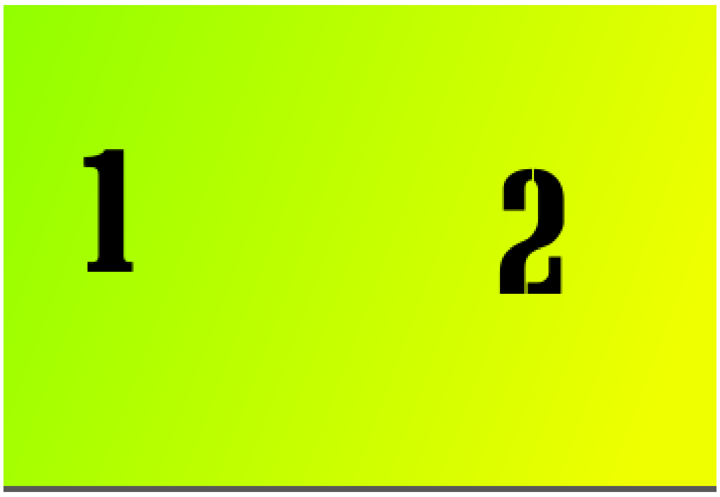
```
Rotation                0x +0.0°
Expression: Rotation                 loopOut("offset",2)
Opacity                 100 %
Expression: Opacity                  loopOut("cycle",3)
```

Pretty cool!

II : JOIN TWO POINTS

In this example, we're going to use referencing, loopOut and wiggle.

To start, create a new Composition, create a background and add two text layers: 1 and 2. Place them in random positions.



Okay, we're going to join these two guys together.

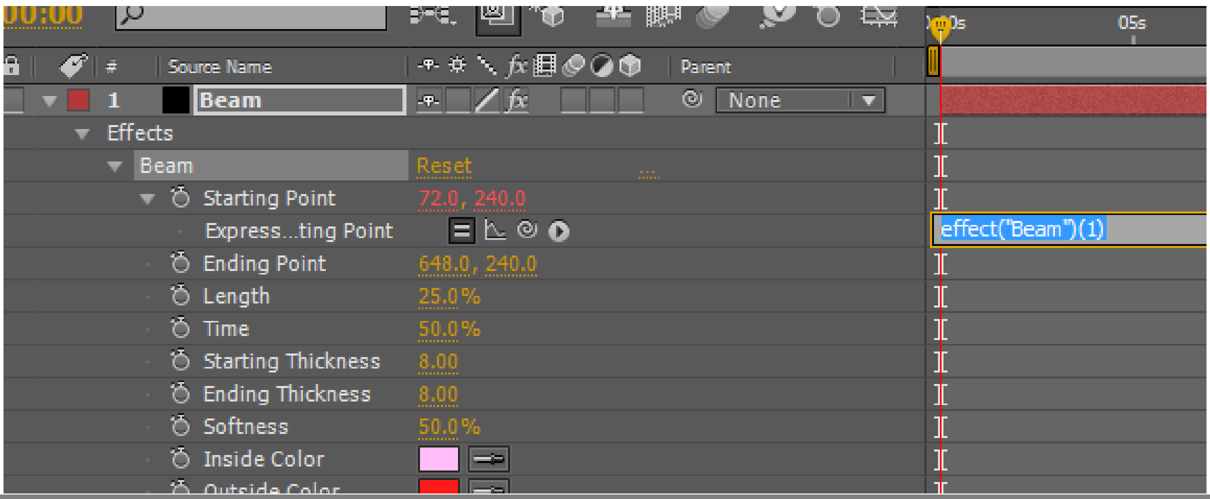
Create a new Solid and add the "Beam" effect (*Effect > Generate > Beam)*

We'll get a red line in the center of the comp.

What we want to do is link the anchor Points to the Start and End points of the Beam effect. For starters, let's set it up by setting both Inside and Outside Color to white and the length to 100%.

We already know that we're going to be using the anchor points. However the anchor point in the timeline is a relative value, so we actually want to use the position (since position is defined by the position of the anchor point in the Comp, while anchor point is relative to the layer and its position).

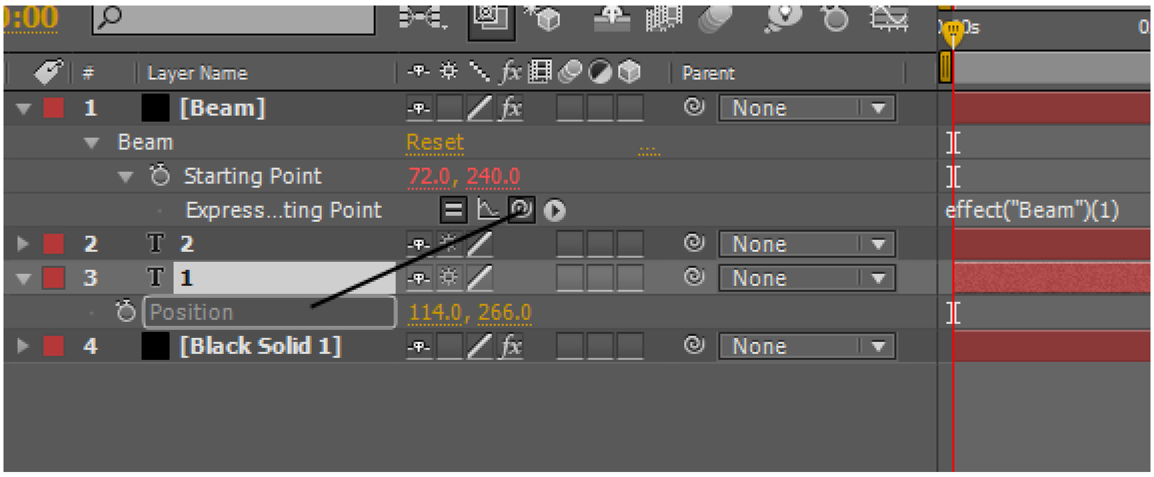
So let's press P to show their positions. Then Alt+Click on the stopwatch of Start point:



Beam	Reset
Starting Point	72.0, 240.0
Express...ting Point	
Ending Point	648.0, 240.0
Length	25.0%
Time	50.0%
Starting Thickness	8.00
Ending Thickness	8.00
Softness	50.0%
Inside Color	
Outside Color	

```
effect("Beam")(1)
```

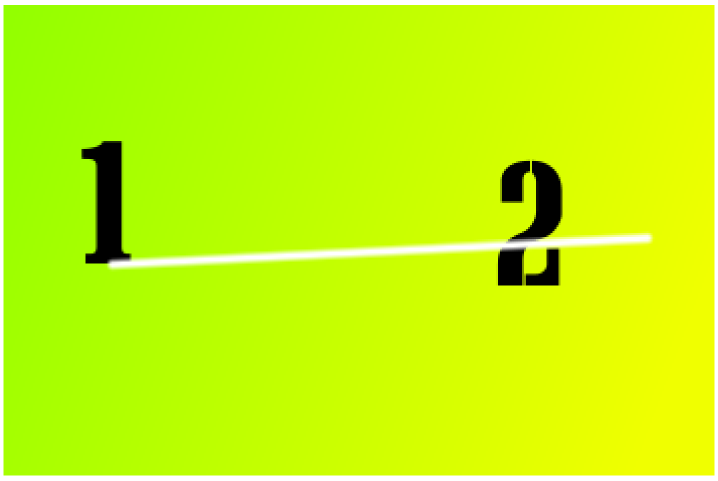
If you aren't on a high resolution screen, you've probably lost sight of the 1 and 2 layers because the effect has completely expanded. In order to fix this, we can just collapse everything then press U, which will show all keyframed and expressioned properties of the selected layer(s). So we can now directly pick whip the position of 1:



We'll get:

thisComp.layer("1").transform.position

At this point we can say we understand this referencing and can immediately tell that this is correct. The view will become like this:

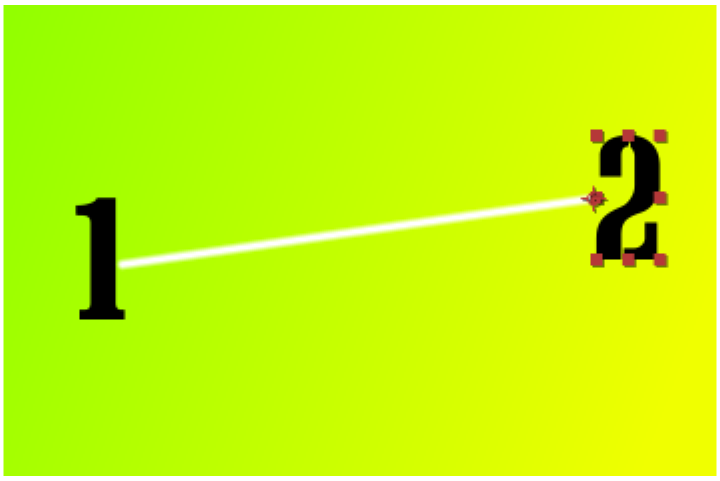


As mentioned before, the anchor point defines the position. So at this point we can move around the anchor Point using the Pan Behind Tool (Y) or move the actual layer. The line follows suit immediately.

So for the two, we can try our luck and type out the referencing, rather than worrying about whether the layer is visible. So, try it. This is how it should look:

thisComp.layer("2").transform.position

Which is essentially the first expression with the layer name changed. So the comp looks like this:



In this image I've moved their anchor points so they look cleaner using the Pan Behind Tool.

The best part is now we can keyframe the positions of layers 1 and 2 and the line will follow them. So we don't have to worry about keyframing those start/end points.

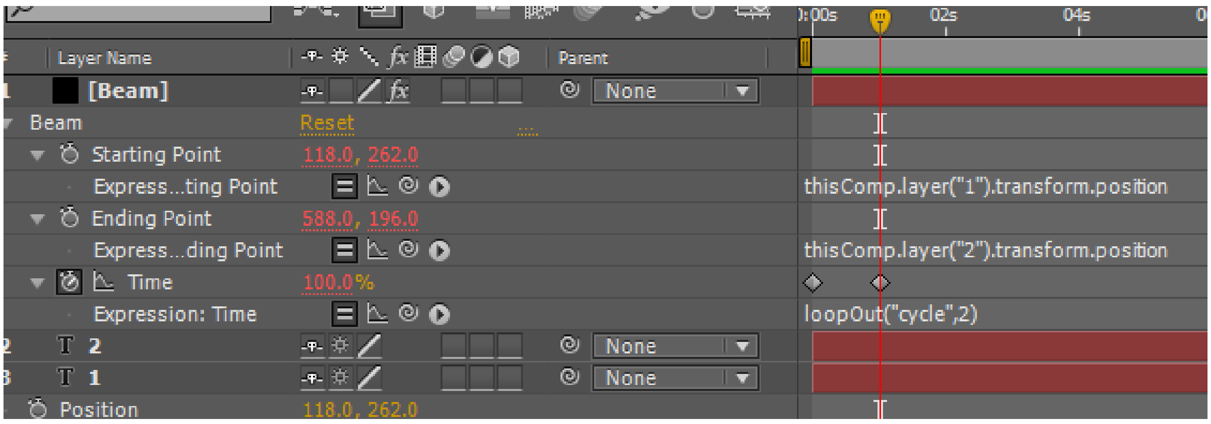
Now let's make this a bit more interesting. Select the Beam layer and set the length to 30%.

Now at the beginning of the timeline, set a keyframe to "Time" and set its value to 0, then go down about 1 second and set another keyframe to 100%.

Now alt+click on the stopwatch and type:

loopOut("cycle",2)

If you remember, this will loop the keyframes, which means we'll have the beam moving from 1 to 2 continuously!



```
Layer Name                          Parent
1  [Beam]                           None
   Beam                  Reset
     Starting Point      118.0, 262.0
       Express...ting Point         thisComp.layer("1").transform.position
     Ending Point        588.0, 196.0
       Express...ding Point         thisComp.layer("2").transform.position
     Time                100.0%
       Expression: Time             loopOut("cycle",2)
2  T 2                              None
3  T 1                              None
   Position              118.0, 262.0
```

Hit RAM Preview and see! Cool!

Try changing the loop type to pingpong and expanding/contracting the space between the keyframes to change the rate at which this occurs.

We can make this look somewhat like a game of Pong. Head over to 1's position and type:

```
[value[0],wiggle(2,100)[1]]
```

What we've done is told it to take the initial value for X (using the square brackets to define that it's only X) then applied wiggle to the Y side.

Remember, wiggle produces an array that has the same dimensions as the control its applied to. So in order to break up this 2 dimensional array into 1 dimension (since it's on the Y side only), we simply add the [1] so we take the wiggle's Y axis.

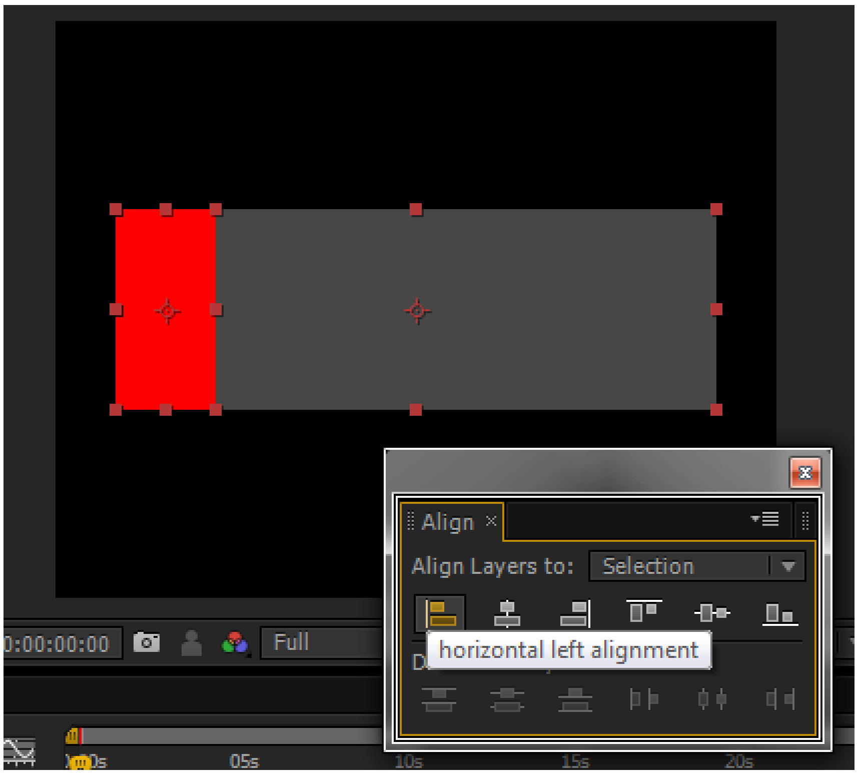
Cool, now we can bring back the 2nd keyframe and RAM preview :-D PONG! (well, pretty close at least).

III: BOUNCING COLOR BAR

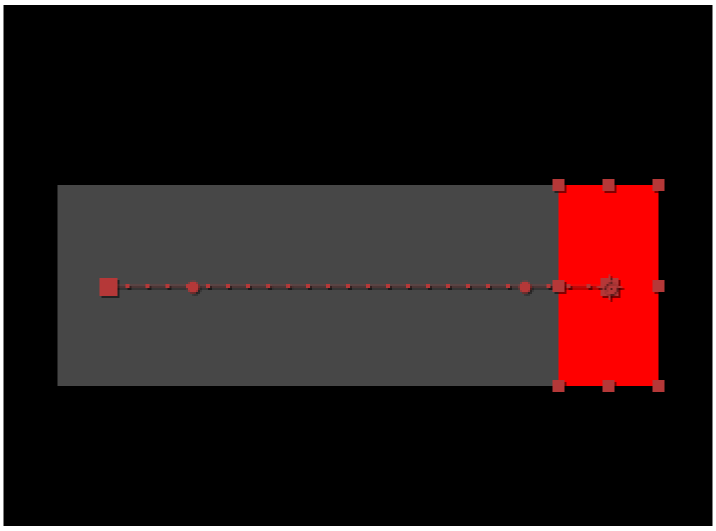
In this example, we're going to use *valueAtTime, time* and *loopOut.*

Create another new comp. This time we'll create a new Grey solid at 650 x 200 named "Back", then another Red Solid 100 x 200 named "Colour".

Using the Align panel (*Window > Align)* we can select Colour and Back and align left:



Create a new keyframe for RED's position, go down .5 seconds then click the "Horizontal right alignment" button in the Align pallet.

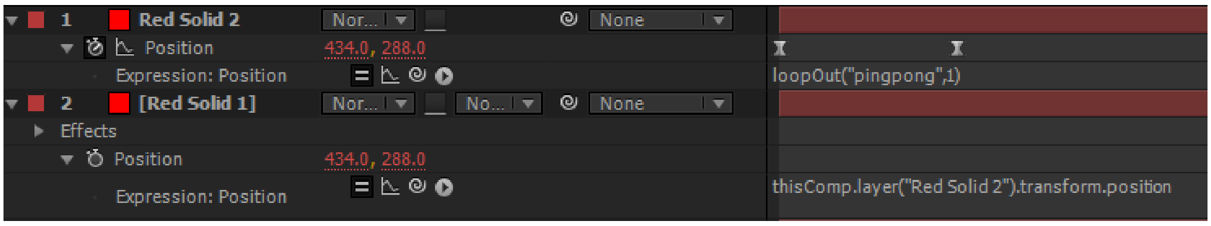


Now we can Easy Ease the keyframes by selecting the two keyframes and hitting F9.

Now we can duplicate this layer. Let’s add a Fill effect (*Effect > Generate > Fill)* and make it a dull orange (#FF961B):

Move this layer down below the Red.

Now, head into its Position property and remove the keyframes, then pick whip the control to the Red’s position:



```
loopOut("pingpong",1)
```

```
thisComp.layer("Red Solid 2").transform.position
```

Now they’re both moving together, which sucks, but we can make it move at a different time. Now, I also want a bunch of colors all having their own color and their own offset. So we can Alt+click on Fill color and add:

```
wiggle(0,.7)
```

A wiggle with frequency of 0 will assign the control a random value.

Okay, next part gets a bit complicated. We know we’ll be using *valueAtTime* and *time,* so let’s look at it:

```
original=thisComp.layer(“Red Solid 2”).transform.position;
original.valueAtTime(time-(index/2))
```

Divided by two because an offset of 2 seconds seems too much. This makes our two objects move opposite one another! Pretty cool, but not what we want. I want to squeeze the layers together, so I’m going to divide by 20:

```
original=thisComp.layer(“Red Solid 2”).transform.position;
original.valueAtTime(time-(index/20))
```

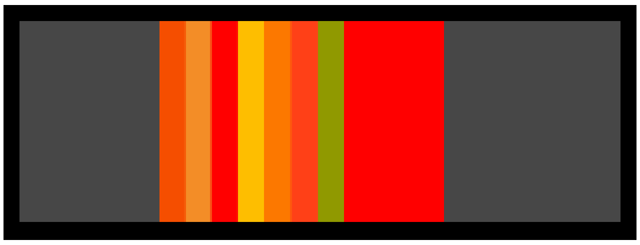
Now duplicate it a whole bunch of times and RAM Preview. Pretty cool, huh?

The problem you'll immediately notice is the big gap between the original and the next layer. This is because of the index offset. In order to remove this we'll add *index-1*. This might add too many brackets (since index-1/20 is index minus one twentieth which isn't what we want).

So delete all the duplicates and do this:

```
original=thisComp.layer("Red Solid 2").transform.position;
place=index-1;
original.valueAtTime(place/20)
```

This will fix our problem and we now have a nice smooth animation:



Nice. We can also select all the layers, squeeze in their vertical scale so we have a nice thin line. Add some Glow in an adjustment layer and we've got ourselves a cool preloader!

Also notice that we can play around with the keyframes of the original layer. Even add keyframes or maybe change the loopOut type, and they will *all* follow this layer.

Pretty awesome. Play around with this and see what you can achieve.

CONCLUSION

So in this document we've covered how expressions work, how to use the pick whip, how to use some interesting expressions like loopOut and wiggle, and how to offset expressions using time and valueAtTime.

But this isn't the end of using expressions. Head over to *Help > Expression Reference...*, check out the little menu with the expressions shortcuts, learn new expressions like *if {} else { }*, *linear*, and more references like *CompWidth, CompHeight, numKeys* etc.

In the After Effects Help system, there's tons of links to awesome tutorials all over the web with some intermediate to advanced stuff. Also check out my blog (http://ideastocreations.blogspot.com). I've got posts on interesting expressions (which covers some stuff that isn't here) and how to use Expression Controls. Don't forget to drop a comment!

Any questions/comments/complaints/suggestions:

- mailthevoid [at] gmail [dot] com
- "Ideas to Creations" page on Facebook
- The blog post where you got this eBook!

It's been fun working with examples and experimenting with them. Thank you again for reading this ebook, share with a friend and share what you create with this.

See you on the next eBook!
davdalx

davdalx presents IDEAS TO CREATIONS

http://ideastocreations.blogspot.com