CHAPTER 2

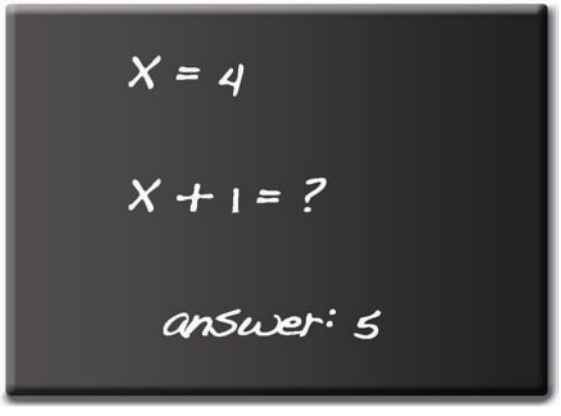
Variables, Comments, and Dimensions

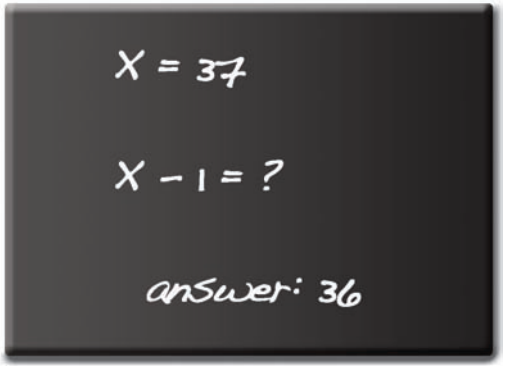
This chapter covers one easy topic (comments) sandwiched between two more complex topics (variables and dimensions). For the complex topics, we'll need to touch on some math, or at least numbers. Don't fret, math-phobic readers. I promise that you'll only dip your toes; you won't get your hair wet. And I also promise to hold your hand through the whole process and to go slowly. I flunked math in high school. I'm not proud of that fact, but I feel safe saying that if I can handle the math in this chapter, you can handle it too. It doesn't go beyond basic arithmetic.

Why bother with all this stuff? Because you need to understand it in order to move beyond simple, Pick Whip-based Expressions. So if you bear with me through this chapter (which I hope you'll also enjoy), you'll pop out the other end much more confident about Expressions. And you'll be able to use this chapter's concepts to craft some really cool effects.

VARIABLES

Perhaps you recall variables from high school math, and perhaps you get a queasy feeling when you think about them. Don't worry: We're not going to use them for anything complicated. But we do need to revisit them. As you may remember—or as you may not—variables are letters that stand in for numbers, as in this problem:

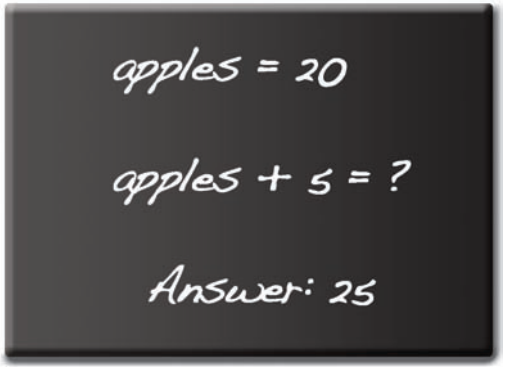




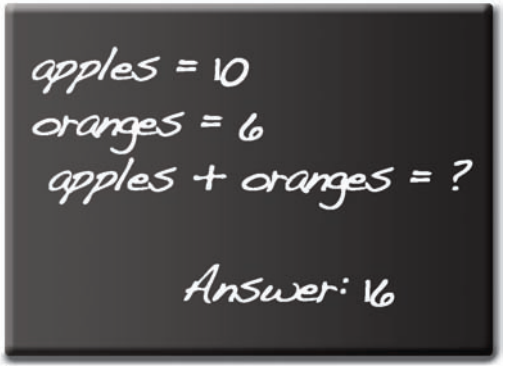
Variables are called variables because they can change what they stand for (they vary). In *that* problem, X means 4. But in this problem, it means 37.

By the way, the opposite of a variable is a constant. The digit 3 is a constant (as are all the other digits). It always means 3. It never means anything else.

In traditional math, variables are always letters: x, y, a, b, c, and so on. But most JavaScript programmers frown on using letters; they prefer words. If JavaScript programmers taught math, they would write problems like this on the blackboard:

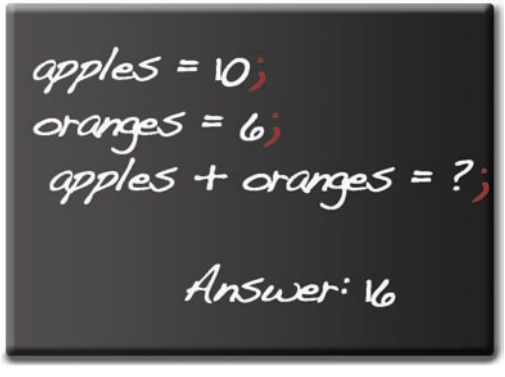


And if they had 10 apples and 6 oranges, and they wanted to know how much total fruit they had, they would write the problem like this:



Boy, I wish my math teacher had used words instead of letters! I might have liked math a little better.

If we'd used JavaScript back in school, we would have written that last problem like this:



Notice the semicolons. In JavaScript, semicolons mean the same thing that periods mean in English. In English, we say that a period marks the end of a sentence. In JavaScript, we say that a semicolon marks the end of a statement. So all this time, when we've been writing Expressions like

```
transform.opacity
```

we could have written them as

```
transform.opacity;
```

But you don't have to include semicolons when your Expression is only one line long. However, you must include them when your Expression is more than one line long. For instance,

```
controller = transform.opacity;
controller
```

Don't worry (yet) about what that Expression means or does, but notice that it contains two statements. I typed each statement on its own line, pressing Return (PC: Enter) after the first statement. As you'll see, you don't have to type each statement on its own line, but doing so makes multiline Expressions easier to read, so I recommend doing it.

The key thing here is that the first statement ends with a semicolon, without which, AE wouldn't understand that the first statement was over and that a new statement was following. This is yet another way in which the Expressions interpreter is literal minded. In English, I could get away with writing, "I'm thirsty Give me some water," without a period between the two sentences. English teachers would complain, but at least everyone would know what I meant. This sort of thing won't work in JavaScript. You must end each statement with a semicolon.

Except for the final one. Notice my second (and final) line doesn't end with a semicolon. Because it's the last line, the interpreter doesn't need a separator to tell it when that line ends and the next one begins. There *is* no next one. However, it's legal to add a semicolon to the last line, if you want to:

```
controller = transform.opacity;
controller;
```

In this book, I'll generally omit semicolons on final lines. Also, I'll type each statement on its own line. But just once, to prove that you don't have to do that, here's a third legal version of the Expression:

```
controller=transform.opacity;controller;
```

This shows that the real function of the semicolon is to separate one statement from another. The separate-line convention does nothing, except to make the Expression easier for humans to read.

Here are a few other legal versions:

- No semicolon after the final statement:

```
controller=transform.opacity;controller
```

- Funky spacing:

```
controller=    transform.opacity;
controller
```

Note that although JavascriptJavaScript allows you to play fast and loose with spacing between words and punctuation symbols, it draws the line at spaces within words. This is *not* legal:

```
controller=transform.opacity;
contr oller;
```

Fine. Now that I've written the same Expression in a zillion ways, what the heck does it mean? Well, the first statement creates a variable called controller. To create a variable, you just make up a word and use it. I could have used


```
hippopotamus = transform.opacity;
```

or

```
empireStateBuilding = transform.opacity;
```

but I chose controller, because I thought it better described the point of making a variable in the first place. This variable is going to control the Expression:

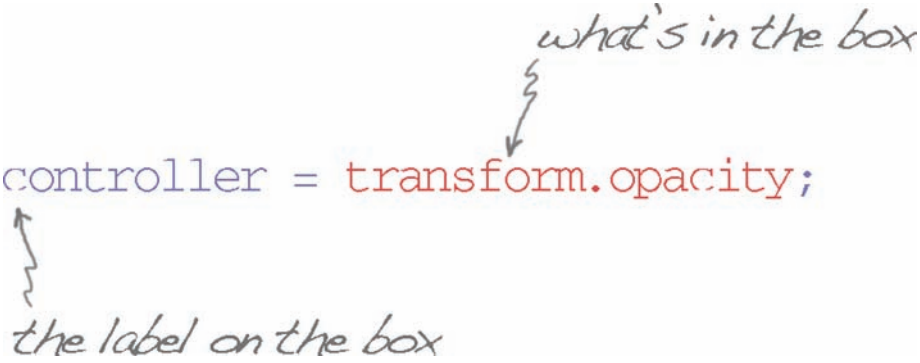
```
controller=transform.opacity;
```

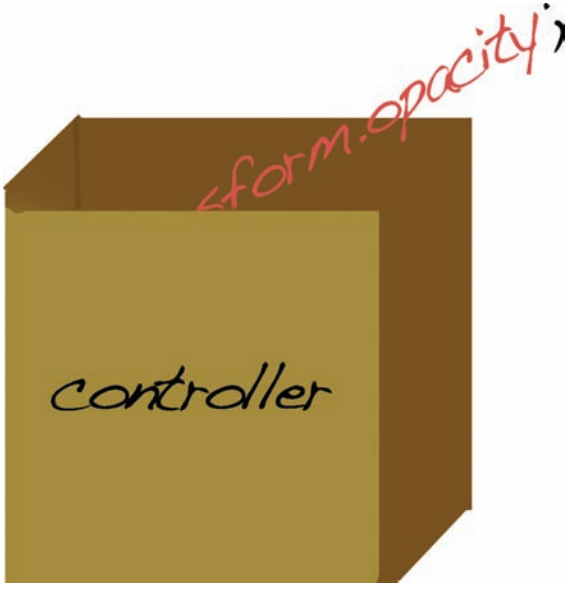
Next, I gave the variable a meaning: in *this* Expression, controller means `transform.opacity`, the value of the Opacity property in the Transform group. So if Opacity happens to be 50 right now, controller is 50 right now. If I change Opacity to 100, controller will be 100:

```
controller is equal to whatever Opacity is equal to.
```

I like to think of variables as cardboard boxes in the attic. By typing `controller = transform.opacity;`, I'm putting a box in the attic and writing "controller" on it in magic marker. Then I'm sticking the value of Opacity

in the box. If I later want the value of Opacity, as I do on the second line of the Expression, I can go back to the box and get it, just as if I later want some socks that I've put in a box, I'll find them in the box labeled socks:





In reality, variables are little chucks of RAM. RAM is your computer's temporary memory (as opposed to its permanent memory, its hard drive). RAM is temporary, because it is erased when you turn off your computer or stop running a program.

Creating a variable is like saying, "Hey, computer! Store the value of `transform.opacity` in a little chunk of RAM, and label that chunk 'controller.' And when I want to know the value of `tranform.opacity`, I'll just say 'controller.' When I say that, tell me what's stored in the chunk of RAM with that name."

(RAM stands for random access memory. The "random access" part means that the computer can access any value stored in memory without starting with the first item, moving to the second, then the third, then the fourth, and so on, until it finds the item it's looking for. Instead, it can jump right to what it wants and access it. The opposite of random access is sequential access. If you're looking for a Stephen King novel on your bookshelf, you can randomly access it, meaning you can jump right to that novel. If you had to sequentially access it, you'd have to start with the first book, check its title, move onto the next one, then

the next, and so on, until you found the book you were looking for. At which point, you'd be so worn out, you'd probably just turn on the TV.)

The last statement of an Expression is the most important one. It's the line that AE will use to control the property to which you apply the Expression. So if you apply this Expression to Rotation. . .

```
controller = transform.opacity;
controller
```

. . . only the second statement, controller, will control Rotation. Let's say I omit it:

```
controller = transform.opacity;
```

That's just telling the computer to store a value in a variable. Okay, it will. But so what? It's like asking an overly literal friend to pick a number between 1 and 10. She says, "okay" and then just stares at you. Why didn't she do anything? She did do something: She picked a number between 1 and 10 *in her mind*. You didn't ask her to tell you the number, so she didn't.

The last statement of an Expression tells the computer to *do* something—something you can actually see, as opposed to something "in its head." And the something it will do will always be the same something: set the property (to which you applied the Expression) equal to the value of the last statement. Because, in this case, the last statement is "controller," the property will be set equal to controller.[1]

Assuming this Expression is applied to Rotation, here it is again with an English translation:

```
controller = transform.opacity;
```

Translation: Store the value of opacity in a box called labeled "controller."

```
controller
```

Translation: Set the value of this property (Rotation) to whatever is stored in the box labeled "controller."

Now, I'll be the first to admit that this is a silly Expression. We could have eliminated the variable and just typed (or Pick Whipped)

```
tranform.opacity
```

[1]In reality, `controller = transform.opacity;` is a legal Expression. If After Effects encounters a one-line Expression that's a variable assignment, it will use the value of the variable as the value of the Expression. So in this case, the value of the Expression would be `transform.opacity`.

as we did in Chapter 1. My point wasn't to show you anything useful; it was to explain a bit about how variables work. But we'll get to useful shortly enough. Before we do, here are a few more grammatical rules about variables.

They can't contain spaces. So this is *wrong*:

```
my age in 2008 = 42;
```

If I want to simulate spaces, I can use

```
my_age_in_2008 = 42;
```

or

```
myAgeIn2008 = 42;
```

In that last example, I used what programmers call camel case, because supposedly the uppercase letters look like humps on a camel. If you're using camel case, as I will in this book, it's traditional to start the first word with a lowercase letter, as I did with "my" and all subsequent words with uppercase letters, as I did with "AgeIn2008."

You may use numbers in variables (I just used 2008), but variables can't start with a number:

```
Good:  hippo1 = 10;
Bad:  1hippo = 10;
```


And the only symbols you can use besides letters and numbers are underscores and dollar signs:

```
Good:  friends_who_live_near_me = 5;
Good:  $$$_in_my_bank_account = 150;
Bad:  myAgeThisYear!!! = 41;
```

Finally, if you want to be formal, you can announce the fact that you're creating a new variable by preceding it with the word "var":

```
var controller = transform.opacity;
controller
```

Notice that you only type "var" the *first* time you use the variable:

```
Bad:
var controller = transform.opacity;
var controller
Good:
var controller = transform.opacity;
controller
```

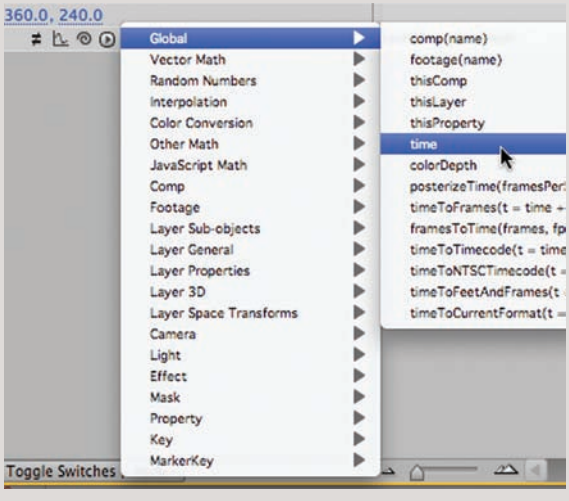
I like "var" because it makes it clear that controller is a variable that I'm defining (and not some already defined word in the JavaScript language), so I'm going to continue using it throughout this book. But you'll see Expressions made by other people besides me, and each programmer has his or her own habits. So get used to sometimes seeing "var" and sometimes not seeing it.

SIDEBAR
Reserved Words

Words that are part of the Expressions language are called reserved words, and you should never use them as variable names. If you do, After Effects won't necessarily display an error (I wish it would). Instead, your Expression will just behave in some odd, unpredictable way.

Reserved words include the following: abstract, as, boolean, break, byte, case, catch, char, class, continue, const, debugger, \default, delete, do, double, else, enum, export, extends, false, final, finally, float, for, function, goto, if, implements, import, in, instanceof, int, interface, is, long, namespace, native, new, null, package, private, protected, public, return, short, static, super, switch, synchronized, this, throw, throws, transient, true, try, typeof, use, var, void, volatile, while, with.

Those are standard JavaScript reserved words, meaning you shouldn't use them as variable names in any system that uses the language (web browsers, Flash, etc.). In addition to those words, After Effects users should avoid naming variables after words that stand for Comp and layer properties. For instance, you should never name a variable position. You can see a complete list of special AE reserved words in the Expressions language menu.



As you can see here, time is a reserved word. So you should never write a statement like this:

```
time = 23;
```

For instance, if you see code that looks like this,

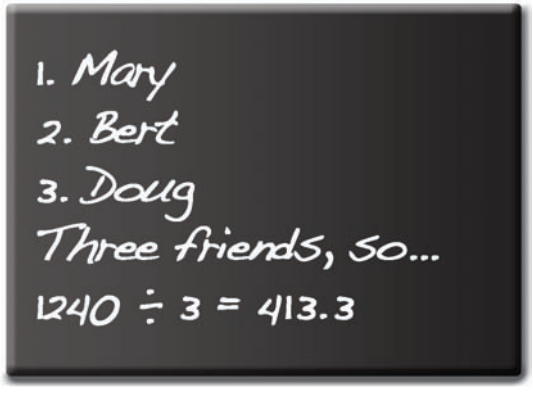
```
var apples = 100;
oranges = 200;
```

understand that both apples and oranges are variables, even though the programmer only preceded apples with the optional "var." Why would a programmer do this? Because, like most humans, programmers are often sloppy. When programmers are paying attention, most of them try to be consistent.

Okay, let's put variables to real-world use. In the following example, the goal is to position four layers at various places in the Comp window and to make a fifth layer automatically move to the average location of the other four. So if the four are stationed at the four corners of the Comp, the fifth will be in the center.

To complete this task, we'll have to use a tiny bit of math. We'll use math for averaging. For instance, if my friends Mary, Bert, and Doug are playing Tetris and I want to know their average score, I'd first need to list their individual scores:

Next, I'd add the three scores together:



Finally, I'd count the number of Tetris-playing friends and divide the total by that number:

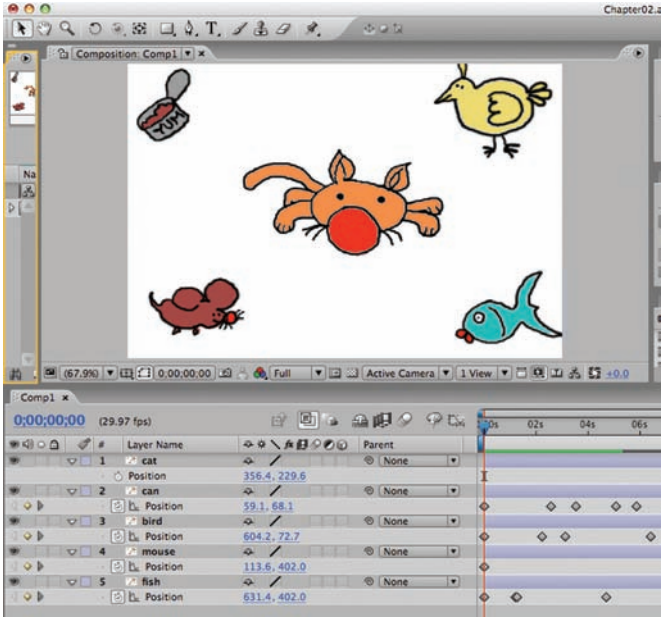
(I used a calculator for that last part.)

That's it. Their average score is 413.3. If you want an average, add up all the individual values and then divide by the number of values.

So to get the average position of four layers, we'll have to add all their positions together and divide the result by 4. Or rather, *we* won't have to do that; AE will have to do it. But we'll have to use an Expression to tell AE to do it. Let's get started:

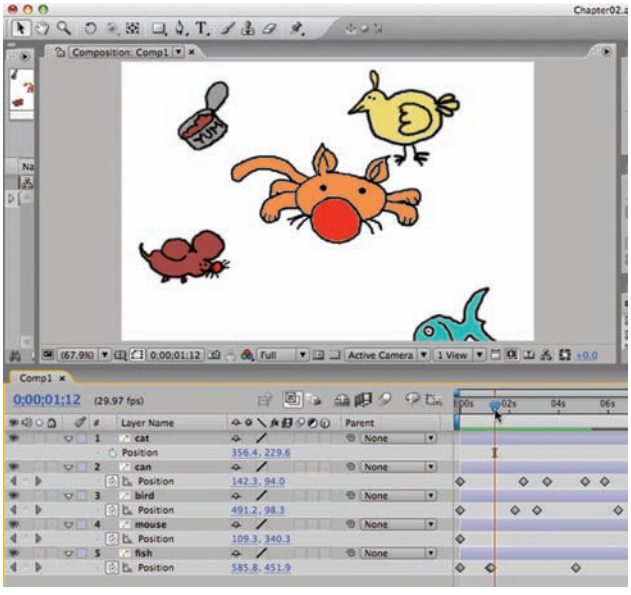
1. Open Chapter02.aep, Comp1 (or use a similar Comp of your own, one that contains five small graphics).

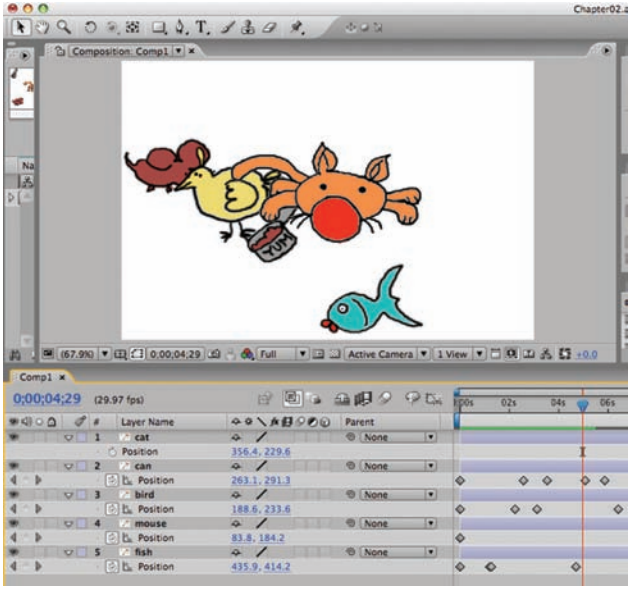
If you preview the Comp, you'll see that I've animated the food layers to move about. But the kitty just stays put.



The kitty is hungry, but he doesn't want to play favorites. He's equally fond of the mouse, the bird, the goldfish, and the can of cat food. Our goal is always to keep the kitty an average distance from his prey. (If each prey layer is at one of the Comp's corners, the kitty should be in the center.)

To calculate the average, we'll have to add mousePosition + birdPosition + goldfishPosition + canPosition. Then we'll have to divide the result by 4 (because there are four food items).





We could write the entire Expression as one long line:

```
(thisComp.layer("mouse").transform.position + thisComp.
layer("bird").transform.position + thisComp.layer("fish").
transform.position + thisComp.layer("can").transform.
position)/4
```

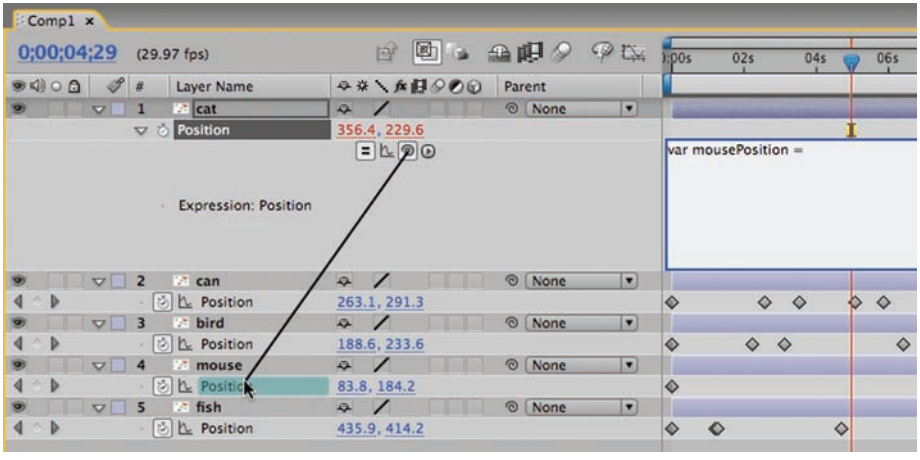
But that would be ugly and confusing. We'll use variables to make the Expression easier to read, edit, and understand.

2. Select all five layers via Command + A (PC: Control + A); then press P to reveal their Position properties.
3. Add an Expression to kitty's Position property.
4. In the text-entry area, type

```
var mousePosition =
```

You might want to add a space after the equals sign, by hitting the spacebar after typing " = ." You don't have to—the Expression will work fine with or without it—but I think it will make the Expression easier to read.

5. Pick Whip the mouse layer's Position property.



The Expression now reads

```
var mousePosition = thisComp.layer("mouse").transform.position
```

Remember, I'm printing Pick Whip–generated text in red and text that you type in blue.

6. Because this is only the first line of a multiline Expression, type a semicolon.

The Expression now reads

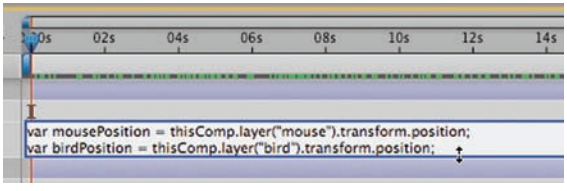
```
var mousePosition = thisComp.layer("mouse").transform.position;
```

7. Press Return (PC: Enter), and on the next line, type

```
var birdPosition =
```

TIP

If you need more space to type a multiline Expression, point your mouse at the lower lip of the text-entry area. When your cursor turns into a double-headed arrow, drag downward:



8. Pick Whip the bird layer's Position property. Then type a semicolon:

```
var birdPosition = thisComp.layer("bird").transform.
position;
```

9. On the next line, type "var fishPosition =" (without the quotation marks), Pick Whip the fish's Position property, and type a semicolon.

```
var fishPosition = thisComp.layer("fish").transform.
position;
```

10. On the next line, type "var canPosition =" (without the quotation marks), Pick Whip the can's Position property and type a semicolon.

```
var canPosition = thisComp.layer("can").transform.
position;
```

So far, your Expression should read

```
var mousePosition = thisComp.layer("mouse").transform.position;
var birdPosition = thisComp.layer("bird").transform.position;
var fishPosition = thisComp.layer("fish").transform.position;
var canPosition = thisComp.layer("can").transform.position;
```

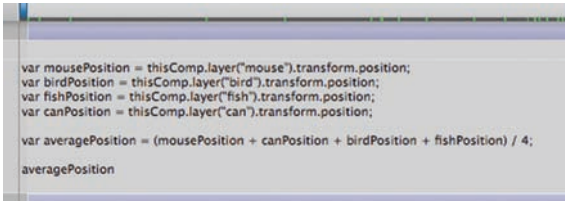
11. On the next line, type

```
var averagePosition = (mousePosition + birdPosition +
fishPosition + canPosition)/4;
```

12. On the final line, type "averagePosition" (no quotation marks). Remember, it's the final line that actually controls the property. So kitty's Position will be controlled by the averagePosition of the other layers.

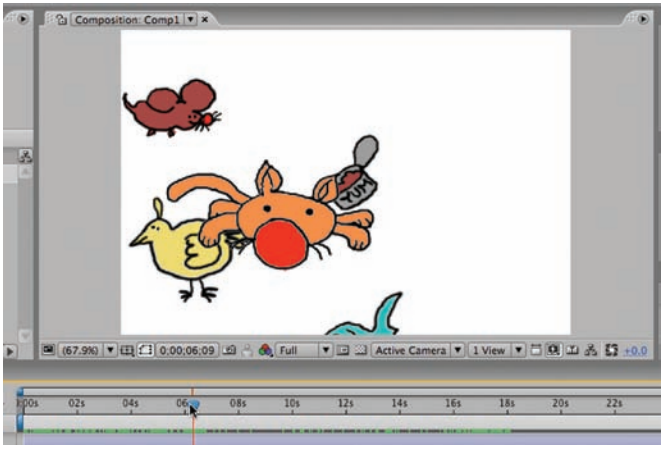
The final Expression reads

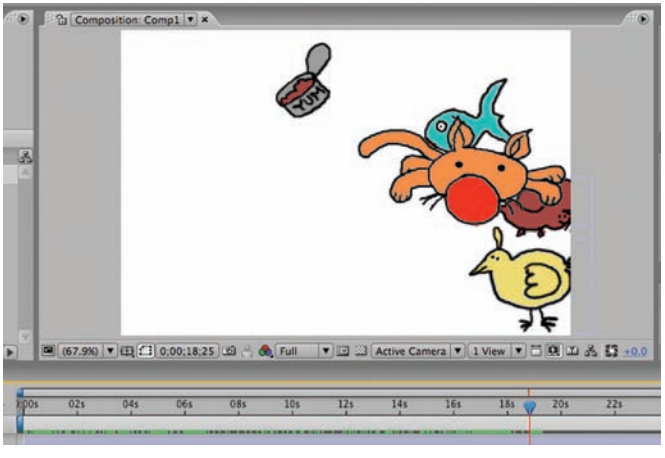
```
var mousePosition = thisComp.layer("mouse").transform.position;
var birdPosition = thisComp.layer("bird").transform.position;
var fishPosition = thisComp.layer("fish").transform.position;
var canPosition = thisComp.layer("can").transform.position;
var averagePosition = (mousePosition + birdPosition +
fishPosition + canPosition)/4;
averagePosition
```



The final line doesn't need to end in a semicolon.

13. Now try previewing the Comp. The kitty should always stay at the average position of the other layers. (How is the poor kitty ever going to eat?)





I want to make sure that you really understand variables. So please take a look at these alternate versions of the same Expression. I've printed changes in bold and, following each version, added some comments.

Alternate 1

```
var mousePosition = thisComp.layer("mouse").transform.position;
var flyingSaucer = thisComp.layer("bird").transform.position;
var fishPosition = thisComp.layer("fish").transform.position;
var canPosition = thisComp.layer("can").transform.position;
var averagePosition = (mousePosition + flyingSaucer +
fishPosition + canPosition)/4;
averagePosition
```

Comments: This is an absurd Expression. Why flyingSaucer? My point is that variable names can be whatever you want them to be. It makes sense to name them something relating to their function, like "birdPosition." But I wanted to make clear that they're not part of the JavaScript language. They're words that you make up. Another programmer might make up totally different words, such as m_Position and b_Position (instead of mousePosition and birdPosition). Get used to seeing different styles for variable names.

Alternate 2

```
mousePosition = thisComp.layer("mouse").transform.position;
birdPosition = thisComp.layer("bird").transform.position;
fishPosition = thisComp.layer("fish").transform.position;
canPosition = thisComp.layer("can").transform.position;
averagePosition = (mousePosition + birdPosition +
fishPosition + canPosition)/4;
averagePosition
```

Comments: This time I removed all the "vars." I like adding them, because they make it clear that what follows is a variable. But they're optional, so some people omit them.

Alternate 3

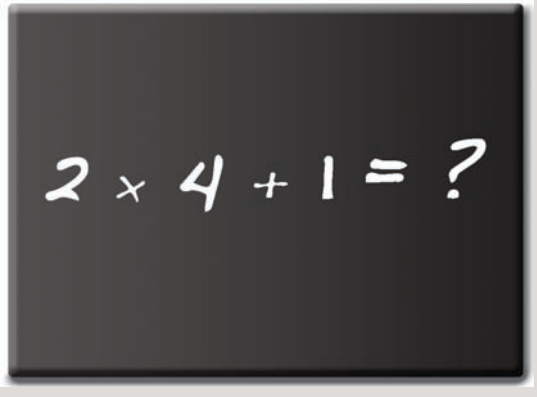
```
var mousePosition = thisComp.layer("mouse").transform.position;
var birdPosition = thisComp.layer("bird").transform.position;
var fishPosition = thisComp.layer("fish").transform.position;
var canPosition = thisComp.layer("can").transform.position;
var numberOfFoodLayers = 4;
var averagePosition = (mousePosition + birdPosition +
fishPosition + canPosition)/numberOfFoodLayers;
averagePosition
```

Comments: Some programmers like to assign all numbers to variables. That way (they argue), numbers are labeled and you know what they stand for. Without the variable, someone might look at the Expression and wonder, "Why is there a 4 in it?" The variable explains what it's doing there. It doesn't help the computer at all, but it's useful for humans who have to read and edit the Expression.

SIDEBAR

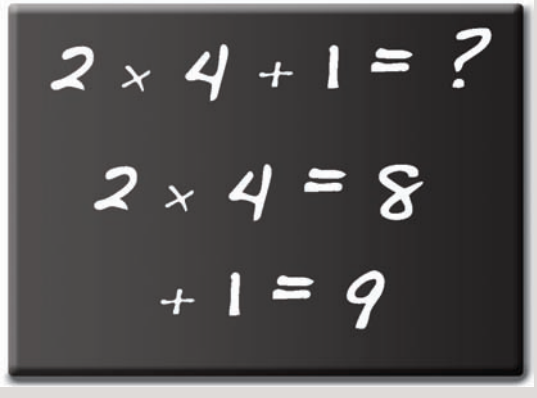
The Power of Parentheses

Can you solve this simple arithmetic problem?

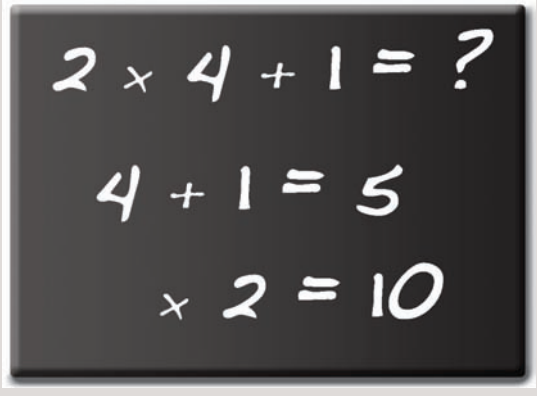


If you answered 9, you're right—maybe. If you answered 10, you're also right—maybe. Huh? Well, let's look at two ways you can work this out.

Let's say you do the multiplication first. Forgetting the "+1" for second, you multiply 2 by 4 and get 8. Now, adding the 1, you get 9.

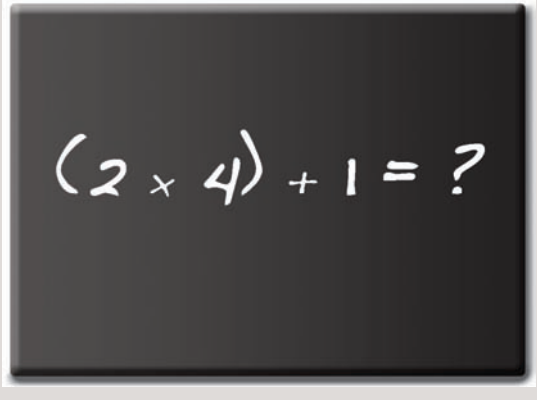


On the other hand, if you start with the addition, you add 4 + 1, which gives you 5. Then, if you multiply that result by 2, you get 10.



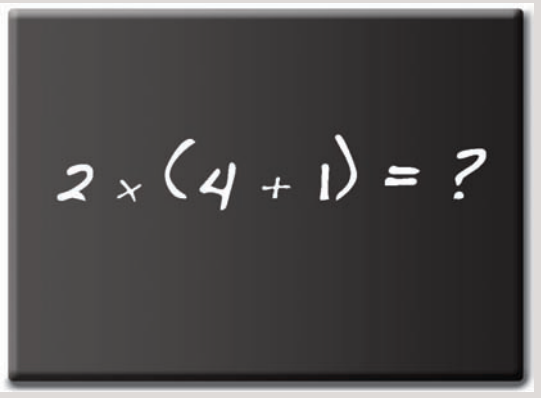
Either answer is (possibly) correct, because the problem is ambiguous. Is it "2 times…4 plus 1" or "4 plus 1…times 2"?

Mathematicians have come up with a complex set of rules to follow when they come across ambiguous problems,* but you don't need to learn them. Instead, you can use parentheses. Just put the calculations that you want to be done first in parentheses. For instance,



means *first* multiply 2 times 4, *then* add the 1. Whereas,

*Here is an abridged version of the rules: First complete any calculations inside parentheses. If there are parentheses inside parentheses, calculate whatever is inside the innermost parentheses first. Then do all multiplication and division calculations, running from left to right. Finally, do all addition and subtraction calculations, running from left to right. According to these rules, there's no need for parentheses in this calculation: 1 + (3 * 5). Multiplication is done before addition. Still, I always add the parentheses. It makes it clear to me which calculation will be done first, and it keeps me from having to memorize all of the complex rules.



means *first* add 4 plus 1, *then* multiply the result by 2.

This is why I used parentheses in the second-to-last line of the "averagePosition" Expression:

```
var averagePosition = (mousePosition + birdPosition +
fishPosition + canPosition)/4;
```

If I'd left them off, the Expression would be ambiguous:

```
var averagePosition = mousePosition + birdPosition +
fishPosition + canPosition/4;
```


What do I want AE to do first, the addition or the division? To clear up the ambiguity, AE would fall back on the complex rules that mathematicians have worked out. Among other things, these rules state that (unless there are parentheses) division should always be done before addition.

So first, AE would have divided canPosition by 4. Then it would have added mousePosition, birdPosition and fishPosition to the result. But that's not what I meant at all. I didn't mean for canPosition to be special. I didn't mean for it to be the only variable divided by 4. I meant that all the variables should be added first. *Then* I wanted the resulting number to be divided by 4. Parentheses make my intentions clear.

COMMENTS

You can leave notes—comments—for yourself (or other coders) in Expressions. For instance,

```
var mousePosition = thisComp.layer("mouse").transform.position;
var birdPosition = thisComp.layer("bird").transform.position;
var fishPosition = thisComp.layer("fish").transform.position;
var canPosition = thisComp.layer("can").transform.position;
var numberOfFoodLayers = 4; //there are four "food" layers
//to get an average, divide the sum of all the values
```

```
//by the total number of values
var averagePosition = (mousePosition + birdPosition +
fishPosition + canPosition)/numberOfFoodLayers;
averagePosition
```

In JavaScript, everything following two forward-slashes is a comment. Comments are for people to read. The computer ignores them.

You can also include comments between a slash-asterisk and an asterisk-slash. This second kind of comment can span multiple lines:

```
/* this Expression was created by Marcus Geduld in 2008.
It ensures that the kitty layer stays at the average
position of the other four layers */
var mousePosition = thisComp.layer("mouse").transform.position;
var birdPosition = thisComp.layer("bird").transform.position;
var fishPosition = thisComp.layer("fish").transform.position;
var canPosition = thisComp.layer("can").transform.position;
var numberOfPreyLayers = 4;
var averagePosition = (mousePosition + birdPosition +
fishPosition + canPosition)/numberOfPreyLayers;
averagePosition
```

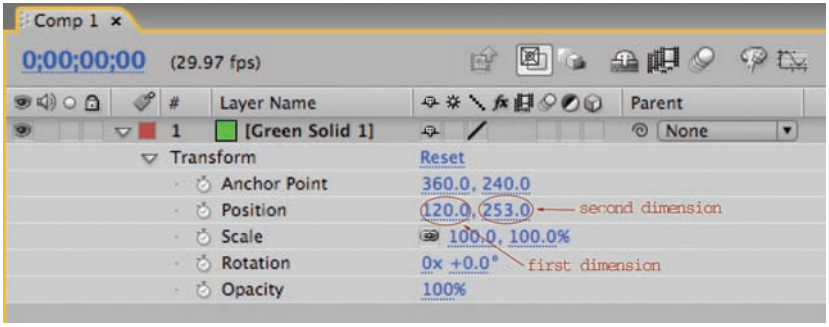
DIMENSIONS

I pulled a fast one on you in the previous chapter. I Pick Whipped willy-nilly, or created the illusion of doing so, and made it look like you could connect any property to any other property. In fact, you *can* connect any property to any other property, but in Chapter 1, I was careful to connect certain properties and to not connect certain others. Specifically, I only connected properties that had the same number of dimensions.

Dimensions? Am I talking about parallel worlds, time travel, or 3D modeling programs? Nope. I'm talking about how many numbers are associated with a particular property.

For instance, when you're working in 2D space,[2] Position has two dimensions, x and y. In other words, there are two numbers associated with Position. You can't say a layer is at Position 120. You *can* say it's at 120x and 253y. Position needs two numbers. Or, if Position is controlling some other property, you can say that it spits out two numbers. Scale is also two-dimensional (in 2D space): it has a width

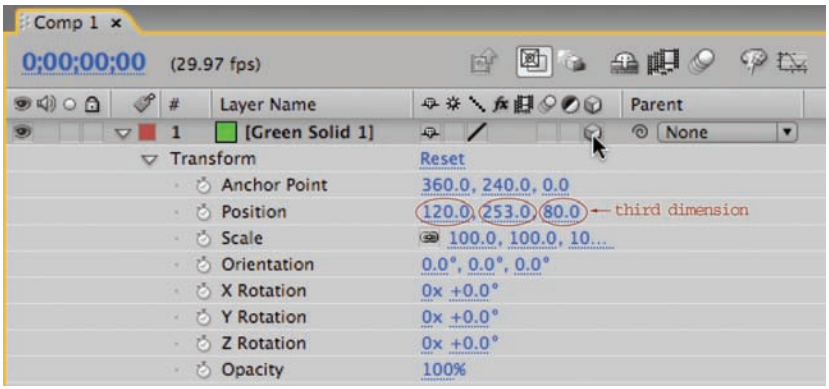
[2] If you turn on a layer's 3D switch, Anchor Point, Position, Scale, and Rotation become three-dimensional. For instance, Position gains a third dimension, called z; x is the left/right dimension, y is the up/down dimension, and z is the closer/farther dimension. A particular layer might have a 3D position of 120x, 253y, 80z.



dimension and a height dimension. We worked with some other 2D properties: Write-on > Brush Position and Beam > Starting Position.

One-dimensional properties include Rotation (when you're not working in 3D mode), Opacity, Gaussian Blur > Blurriness, and Text Animation > Tracking Amount. Each of these properties has just one number associated with it. For instance, Opacity can be 62%, but it can't be 62%, 49%.

After Effects contains 1D, 2D, 3D, and 4D properties. The 3D properties include Position, Anchor Point, and Rotation, if you're working in 3D mode.

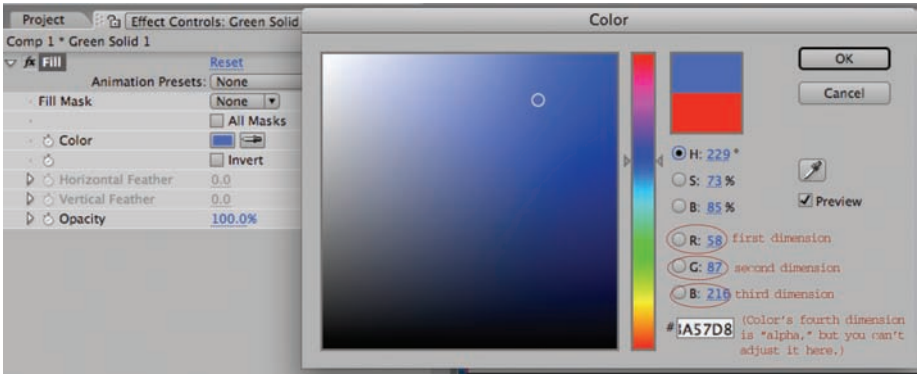


I used to wonder why Scale also gained a third dimension. If we were working in Maya or 3D Studio Max, we'd say Scale's third dimension is depth (of width, height, and depth fame). But in AE, if you scrub a 3D layer's third Scale dimension, it does not gain more depth. It always stays flat, like a postcard. In fact, some people call AE's 3D mode "postcards in space." So what's the purpose of Scale's third dimension? It has several purposes, but here's the one that's applicable to this book: making Scale 3D allows it to better hook up to other 3D Properties. If 3D Scale is controlling 3D Position, the dimensions will "line up." Scale's dimension order is width,

then height, then depth; Positions is x, then y, then z. So width will control x, height will control y, and depth will control z.

There's only one 4D property, and that's Color (in various Effects, such as Generate > Fill). What can that mean? It means Color has four numbers associated with it: a number for how much red is in the color, a number for how much green is in the color, a number for how much blue is in the color, and a number for how transparent the color is (its amount of "alpha").

When you pick a color from one of those handy little color chips, AE converts your choice into four numbers. It doesn't show you these numbers; it just shows you a color, but as we'll discover soon enough, if you understand the four dimensions of color, you can do some neat tricks.



One quick detour before we start to play with dimensions: You need to know how they're numbered. Let's use 2D Position as an example. As you know, Position has two numbers associated with it: x and y. AE always lists dimensions in a specific order. It's always x and then y. It's never y and then x. Along the same lines, Scale is always listed as width and then height. It's never height and then width. Given this fact, a normal person would number Position's dimensions as

Dimension 1: x
Dimension 2: y

And a normal person might say x is Position's first dimension, and y is Position's second dimension. But JavaScript was invented by programmers, and programmers are not normal. (I should know. I program for a living.) Programmers count like this:

Dimension 0: x
Dimension 1: y

Programmers generally start counting with 0, not 1. So x is the zeroth dimension and y is the first dimension. In 3D, the dimensions are numbered:

Dimension 0: x
Dimension 1: y
Dimension 2: z

This isn't too hard to understand, but it's a little odd: there are three dimensions (three numbers associated with 3D Position), but the third dimension (z) is dimension 2, not dimension 3.

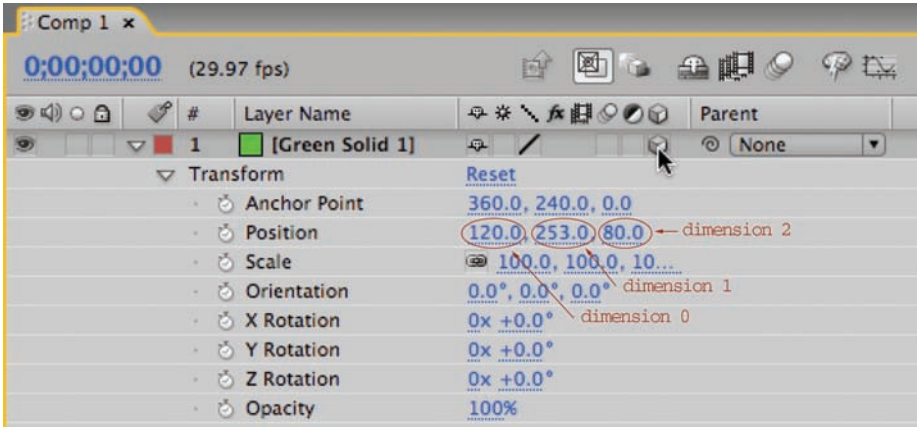
For width, the dimensions are numbered:

Dimension 0: width
Dimension 1: height
Dimension 2: depth (if the layer is 3D)

For color, the dimensions are

Dimension 0: red
Dimension 1: green
Dimension 2: blur
Dimension 3: alpha (Transparency/Opacity)

Because programmers always start numbering with 0, the highest-numbered dimension is always one-less-than the total number dimensions (so if we're working in 3D, the highest numbered dimension is 2).



Sometimes I think that when programmers count on their fingers, then say, "I've got 10 fingers. See: 0, 1, 2 3, 4, 5, 6, 7, 8, 9. Like I said, 10 fingers!" And, when a programmer's first child is born, he calls it his zeroth child.

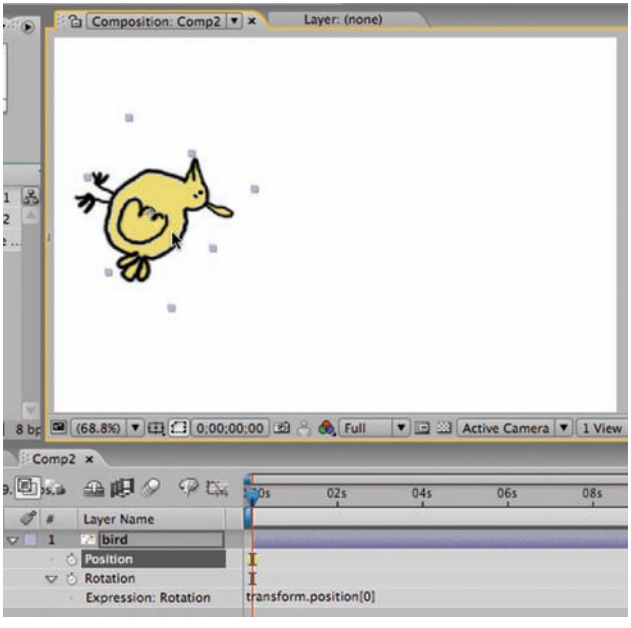
Let's take a look at the properties we connected in Chapter 1:

Rotation (1D)—Opacity (1D)
Position (2D)—Scale (2D)
Fast Blur: Blurriness (1D)—Text: Tracking Amount (1D)
Write-on: Brush Position (2D) —Beam: Starting Point (2D)

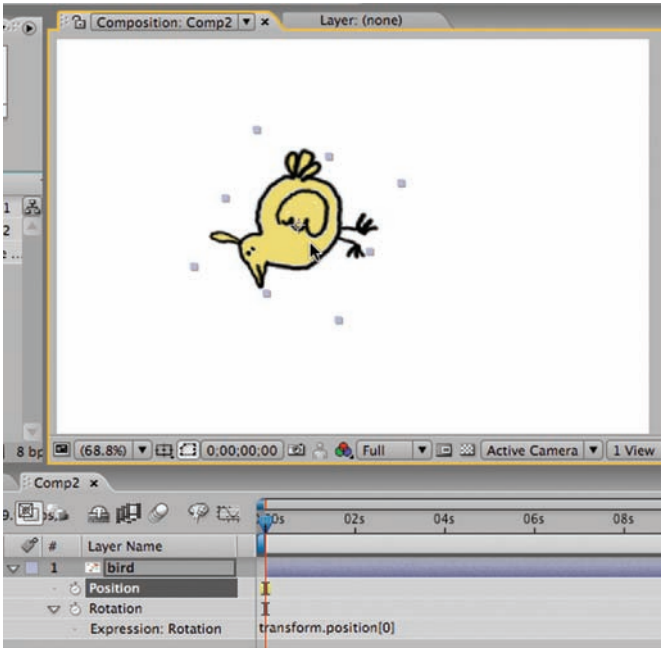
Not very adventurous. With Expressions, you can connect 2D properties to 3D properties, 1D properties to 2D properties, 4D properties to 2D properties, and every other permutation you can think of. But it's important to understand what happens when you mix dimensions. If you don't understand this outcome, you're likely to get a result that's very different from whatever you're trying to achieve. But let me make it crystal clear that you *can* use the Pick Whip to connect any two properties. Somehow, AE will make it work, something interesting will happen, and you'll never get an error.

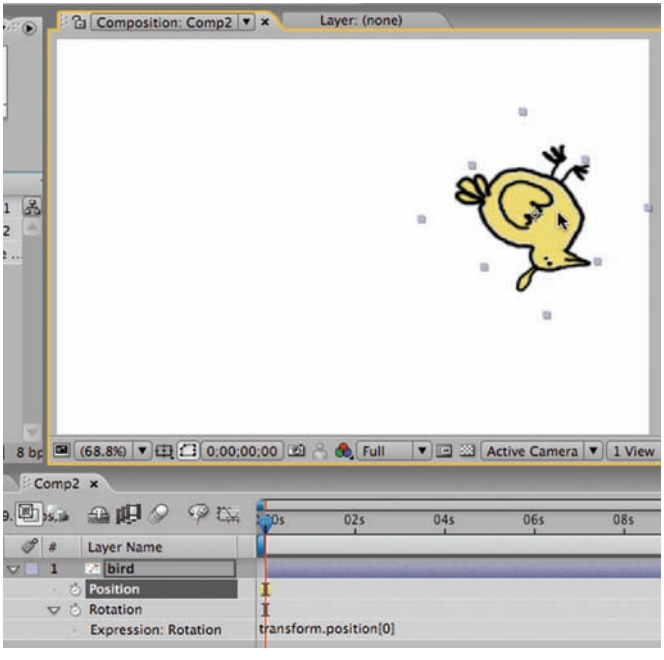
How can properties with different numbers of dimensions communicate with each other? Think about it for a minute. Imagine you're trying to use Position to control Rotation. Position spits out two numbers (x and y), but Rotation can only eat one number. It's like you're trying to stick two keys into the same lock at once. So what happens? Let's see:

1. Open Chapter2.aep, Comp2. (Or create your own Comp containing one small solid.)
2. The Comp only contains one layer. Twirl open its properties, and add an Expression to Rotation.
3. Pick Whip Position.
4. Try dragging the layer around.

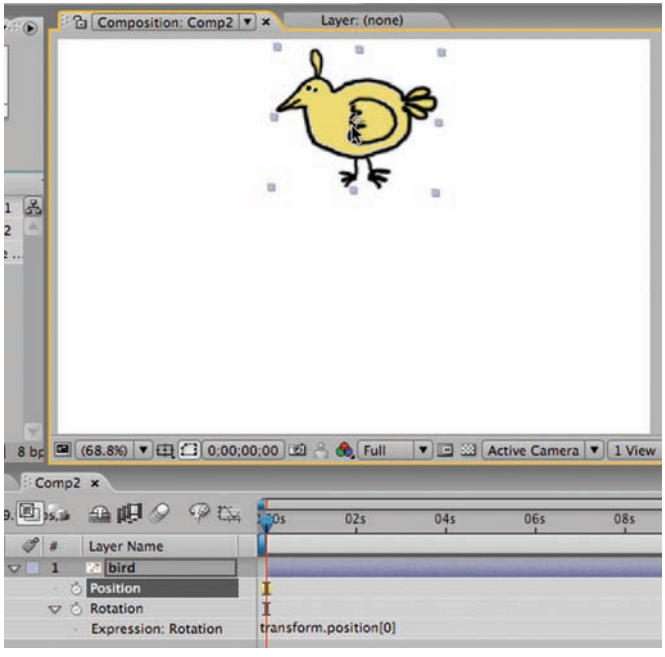


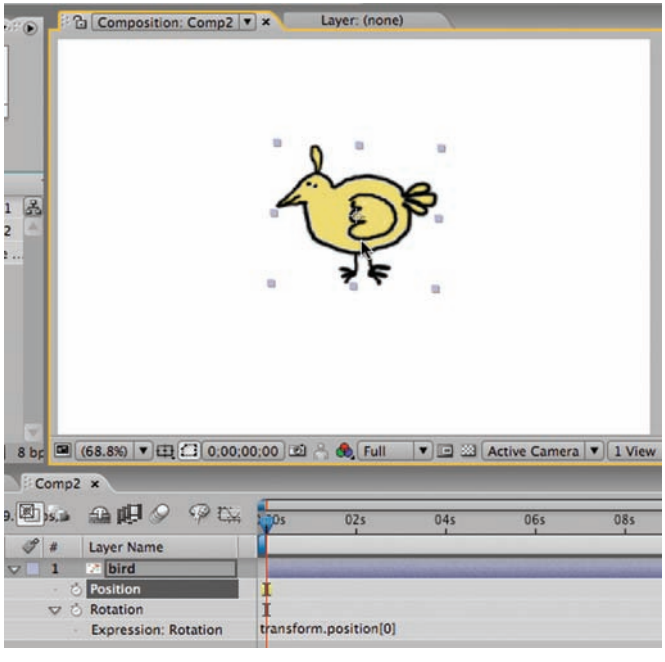
Notice that when you drag the layer left and right, it rotates.

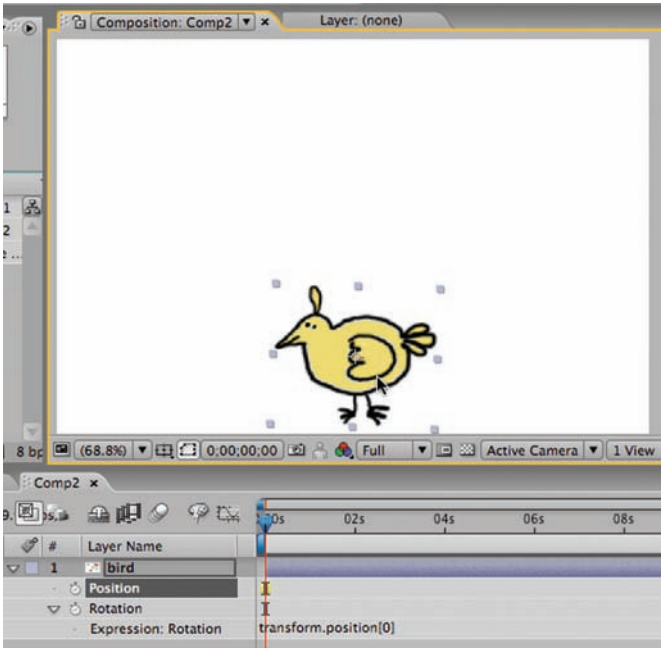




But when you drag it up and down, it doesn't rotate.

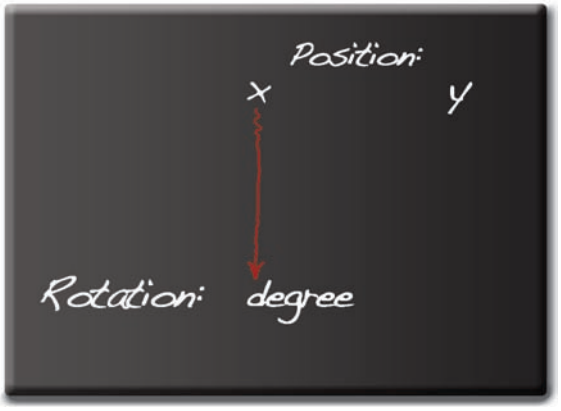

Composition: Comp2
Layer: (none)
(68.8%)
0;00;00;00
Full
Active Camera
1 View
Comp2
0s
02s
04s
06s
08s
Layer Name
1
bird
Position
Rotation
Expression: Rotation
transform.position[0]


Composition: Comp2
Layer: (none)
(68.8%)
0;00;00;00
Full
Active Camera
1 View
Comp2
0s
02s
04s
06s
08s
Layer Name
1
bird
Position
Rotation
Expression: Rotation
transform.position[0]



Why not? Because Position isn't controlling Rotation. Position's *x dimension* is controlling Rotation, and x is the left/right dimension. Remember, Rotation is 1D, so it can only eat one number. But Position spits out two numbers. So Rotation must eat one and ignore the other. But why eat x and ignore y?

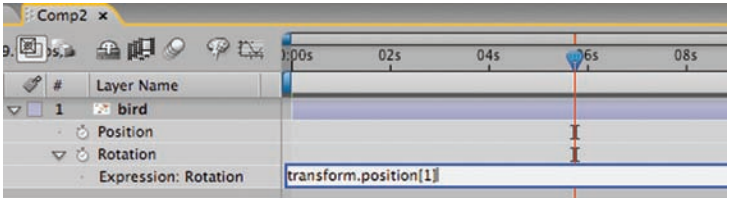
Because x is Position's first dimension (or, as a programmer would say, its zeroth dimension), and degrees is Rotation's first dimension (it's zeroth dimension). Sure, Rotation only has one dimension, but if you only have one child, that child is still your first child. AE lines up the same-numbered dimensions.



Let's take a look at the actual Expression on Rotation, created by the Pick Whip:

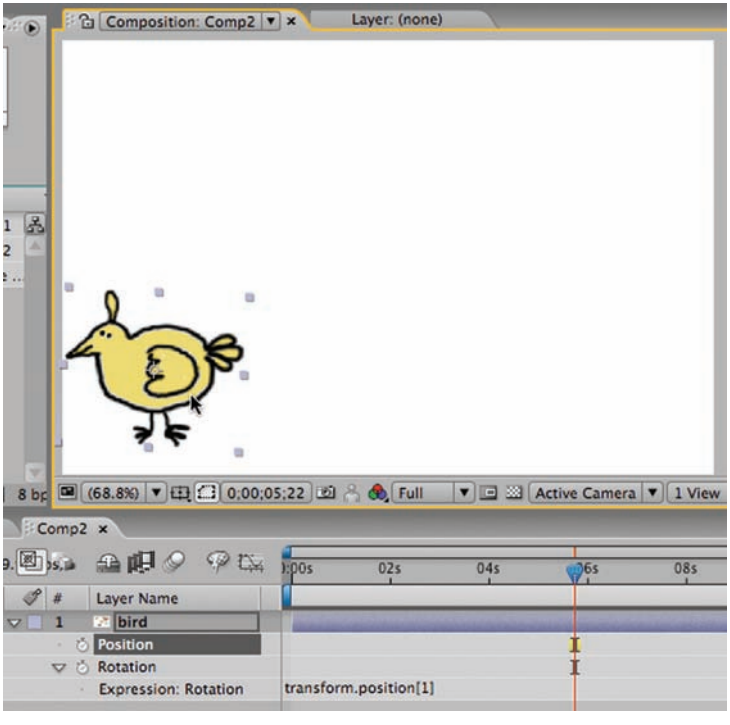
```
transform.position[0]
```

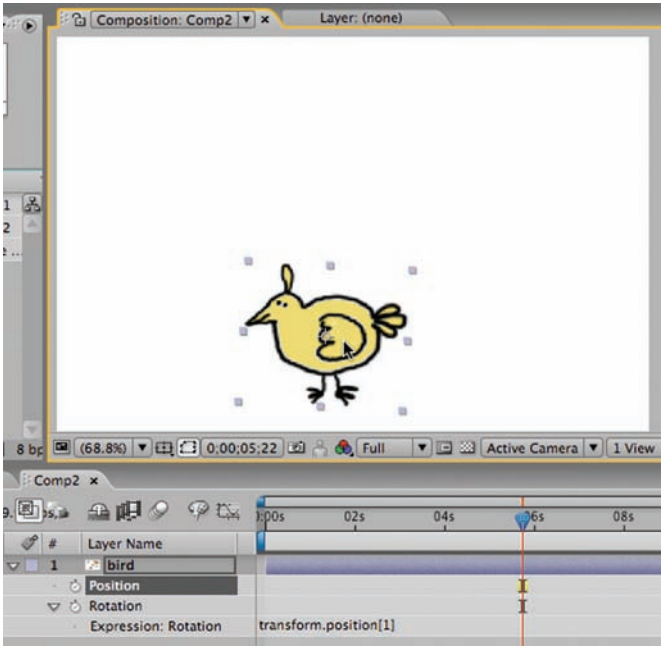
Notice the "[0]" after "position." That refers to Position's zeroth dimension, which is x. Try carefully editing the Expression, changing the 0 to a 1 (don't accidentally delete the brackets):

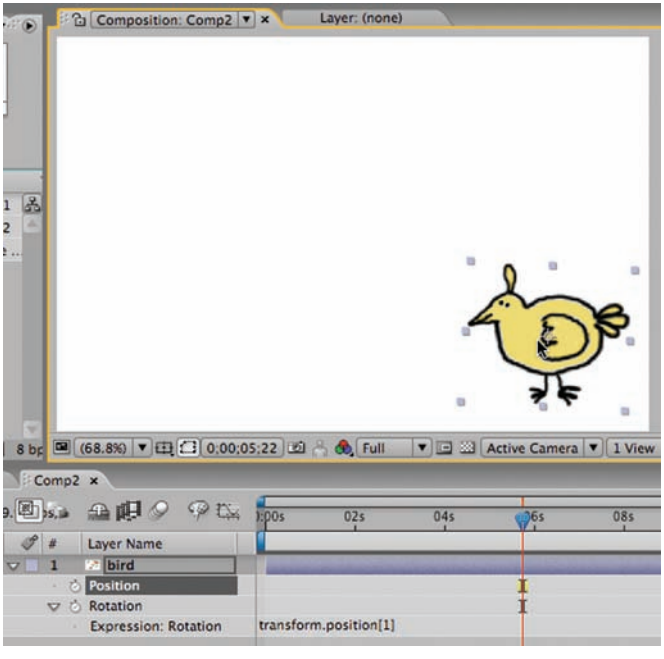



```
transform.position[1]
```

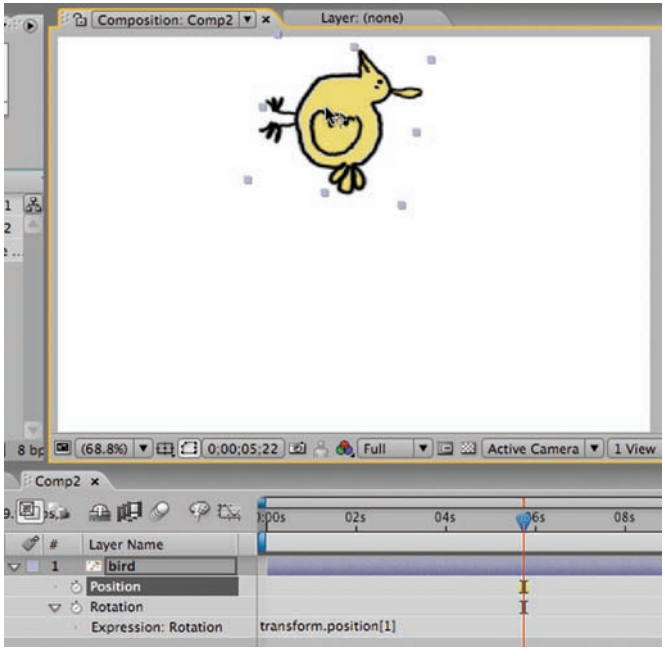
Now, once again, try dragging the layer around. Now, when you drag left and right, nothing happens:

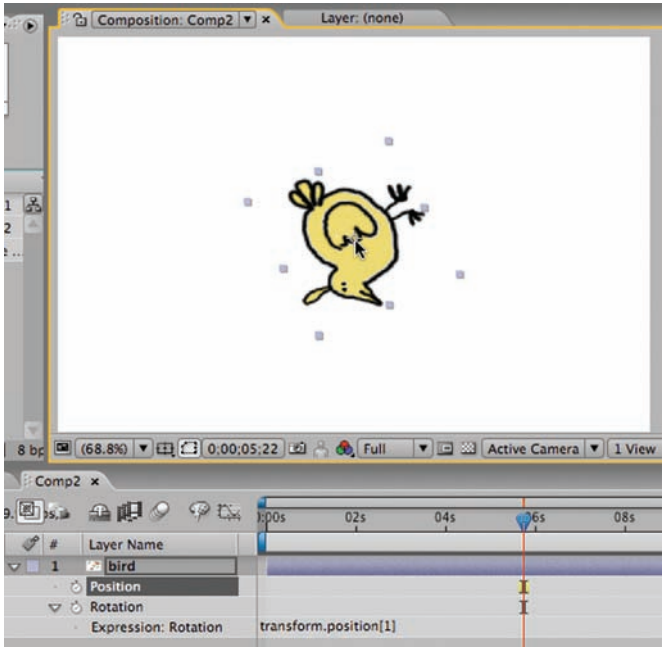


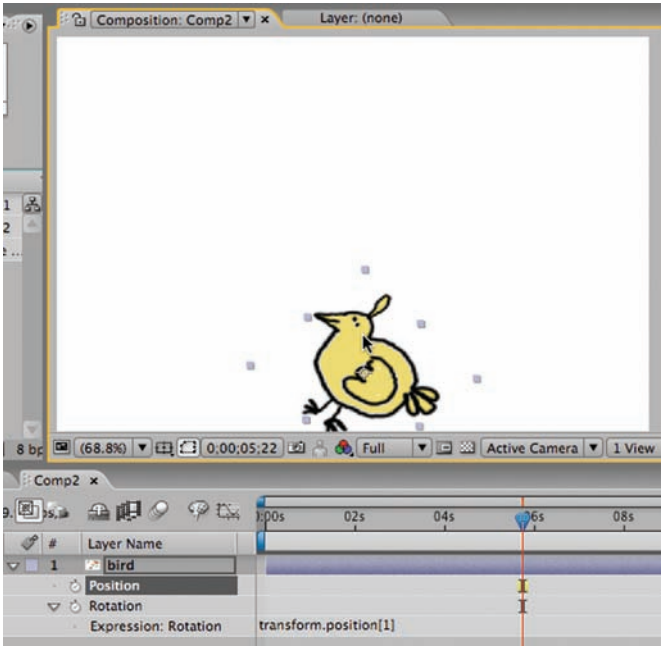

Composition: Comp2
Layer: (none)
(68.8%)
0;00;05;22
Full
Active Camera
1 View
Comp2
Layer Name
1
bird
Position
Rotation
Expression: Rotation
transform.position[1]


Composition: Comp2
Layer: (none)
(68.8%)
0;00;05;22
Full
Active Camera
1 View
Comp2
Layer Name
1
bird
Position
Rotation
Expression: Rotation
transform.position[1]

But when you drag up and down, the layer rotates:

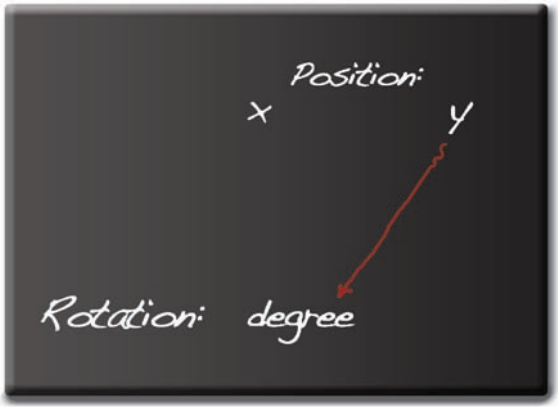


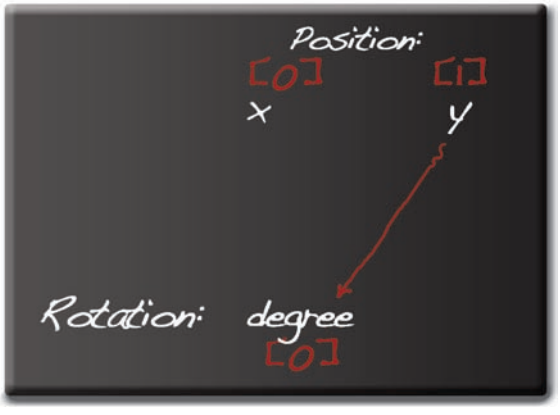




This is because dimension 1 (what programmers call the second dimension) is y:

Here's the same chart with dimension numbers added:

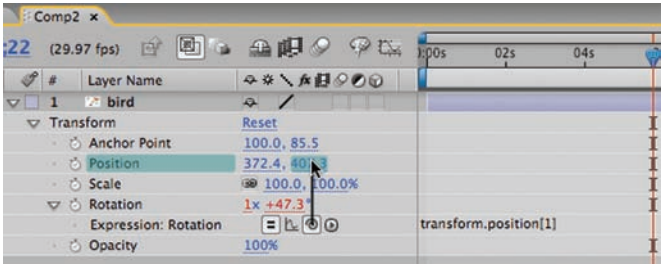




By the way, if this is the effect you want, there's an easier way to achieve it than Pick Whipping and then changing the 0 to a 1.

Instead of dragging the Pick Whip to the word "Position," drag it directly to Position's y value. AE will put a 1 between the brackets instead of a 0, because

it will know—via your Pick Whipping—which dimension you want to use to control the Expression.



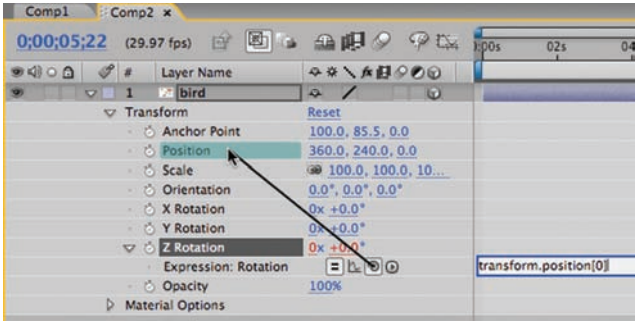
But if you just point to Position, how is it supposed to know which of Position's two dimensions you want to pipe into Rotation, a one-dimensional property? It doesn't, so it just falls back on a default rule, which is to line up same-numbered dimensions (dimension 0 to dimension 1). Even though I know this, I always forget to Pick Whip the dimension I want. But I don't fret about it. It's pretty easy to change a 0 to a 1.

Let's try another experiment, this time in 3D:

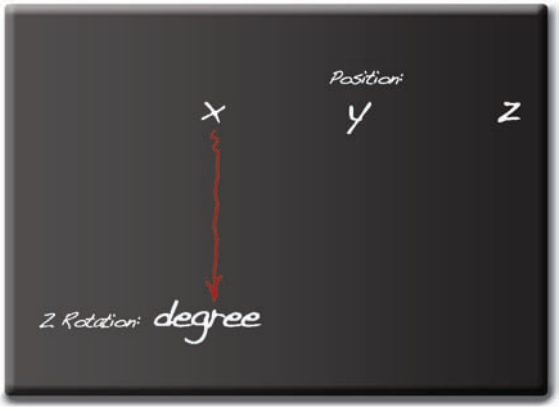
1. Remove the Expression from Rotation.
2. Turn on the layer's 3D switch.

Position and Rotation will now expand to three dimensions. In fact, Rotation will turn into four properties: X Rotation, Y Rotation, Z Rotation, and Orientation. X, Y, and Z Rotation are 1D properties. Orientation is a 3D property that single-handedly controls X, Y, and Z. We'll experiment with both.

3. Add an Expression to Z Rotation, and Pick Whip Position.



Try dragging the layer around. Once again, dragging left/right rotates the layer and dragging up/down doesn't. If you try scrubbing the layer's z Position, you'll see that, as with y, the layer doesn't rotate.



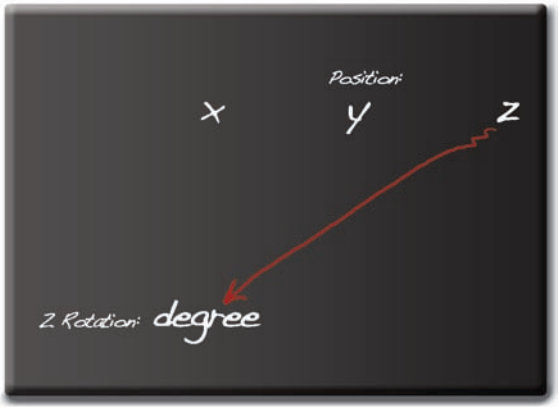
The Expression looks like this:

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

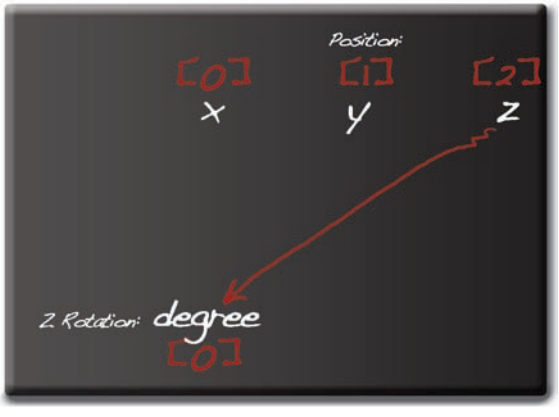
4. Change the 0 to a 2.

```
transform.position[2]
```

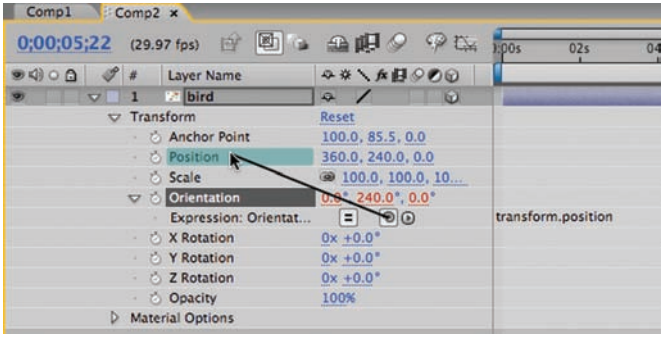
Now when you drag the layer left and right (or up and down), it doesn't rotate. But because dimension number 2 is z, if you scrub Position's z value, you'll see the layer rotate. You've created this sort of relationship.



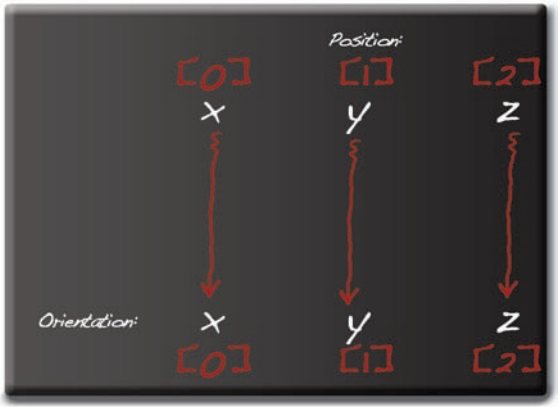
Here's the same chart with dimension numbers added:



Before we move on, try removing the Expression from Z Rotation and adding one to Orientation. Then Pick Whip Position (the word "Position").



Unlike Z Position, which is 1D, Orientation is a 3D property. It can hook directly up to another 3D property, such as 3D Position, with no fuss. Try scrubbing the x, y, and z values for Position. Predictably, when you move the layer back and forth, it rotates around its x axis; when you move it up and down, it rotates around its y axis; and when you move it in and out, it rotates around its z axis.

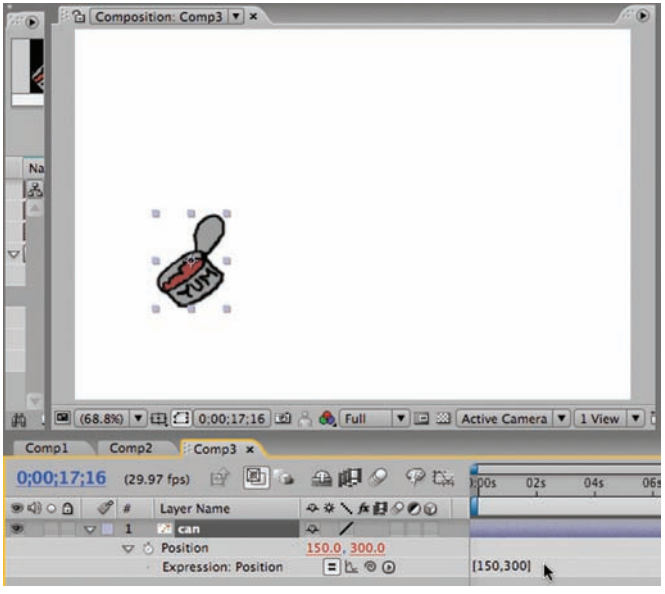


Earlier, we connected a 2D property (Position) to a 1D property (Rotation), so that the 2D property controlled the 1D property. Let's try it the other way around. What if you want to control a 2D property with a 1D property? Using our old friends Position and Rotation, what if we added an Expression to Position (2D) and Pick Whipped Rotation (1D)? Position *needs* two values, but Rotation only has one value to feed it. Where will the other value come from?

Before I answer that question, let's backtrack to the simplest possible sort of Expression: a number. In Chapter 1, we added an Expression to Rotation and typed "45." That makes sense: 45 degrees. But what if you wanted to write a similar Expression for Position. You can't just type 45, because Position

demands two numbers, one for x and another for y. Let's say you wanted to set x to 150 and y to 300. How would you type that as an Expression?

1. Open Chapter02.aep, Comp3 (or create a Comp with one small solid in it).
2. Twirl open the one layer's transform properties, and add an Expression to Position.
3. In the text-entry area, type "`[150,300]`" (no quotation marks) and finalize the Expression.

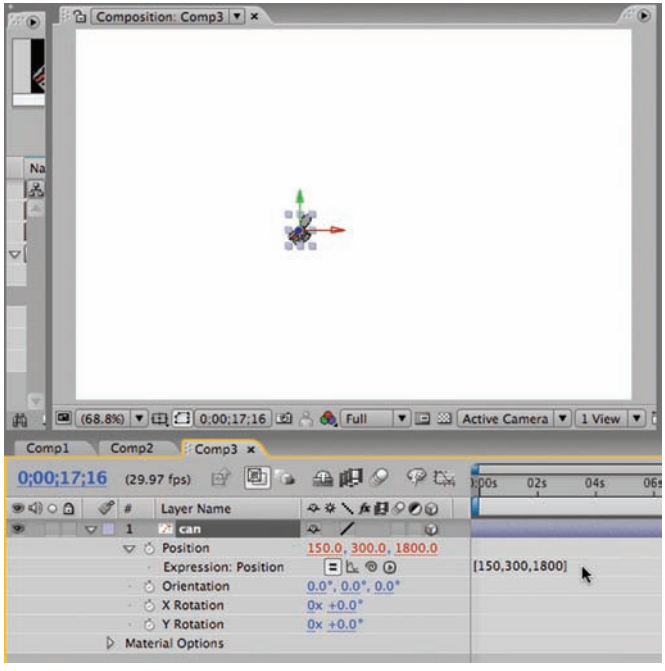


The layer moves to 150x, 300y.

In JavaScript, a comma-separated list, surrounded by brackets, is called an array. "Array" is just a fancy programmer's term for a list. Because Position is a 2D (or sometimes 3D) property, it can't accept just one number; it requires a list of numbers. You must type that list in a way that JavaScript understands, and JavaScript understands array lists, which are surrounded by square brackets.

4. Turn on the layer's 3D switch.
5. Update the Expression so that it looks like this

```
[150,300,1800]
```

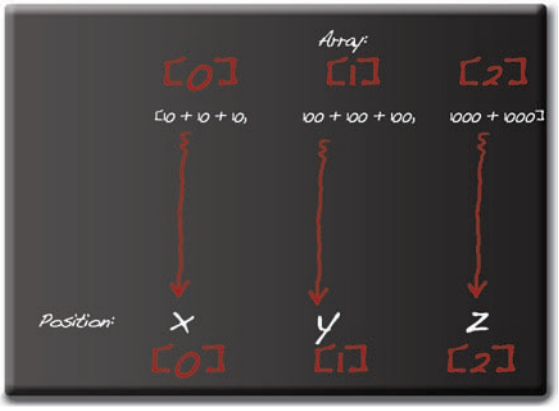


Arrays can contain as many, or as few, values as you like, but the values must be separated by commas and be enclosed within brackets. And the order is important. Because Position's dimensions are ordered x, y, and z, the array's values will be piped into Position in that order:

The array values needn't be simple numbers. Try updating the Expression as follows:

```
[10 + 10 + 10, 100 + 100 + 100, 1000 + 1000]
```

I can't think of a good reason to write an Expression like that, but it works.



SIDEBAR
Arrays for All Properties

Many properties, even the 1D ones, accept arrays. For instance, Rotation can be [45] and Opacity can be [50]. An array is a list, and you use it to list all the values for a property's dimensions. Rotation and Opacity only have one dimension, so it makes sense to give them lists that only have one item in them.

There's no advantage to typing [50] instead of 50. Either works. The second is shorter, so most people use it. But there's a neat logic to arrays:

- Rotation [45]
- Scale [210, 80]
- 3D Position [150, 200, 900]
- Color [.5, 1, 0, 1]

Here's another variation you can try:

1. Remove the Expression from Position. Add it back. (Add a new Expression to Position.)
2. Type an open bracket:

   ```
   [
   ```

3. Pick Whip X Rotation:

   ```
   [transform.xRotation
   ```

4. Type a comma and a space. The space is optional:

   ```
   [transform.xRotation,
   ```

5. Pick Whip Y Rotation:

```
[transform.xRotation, transform.yRotation
```

6. Type another comma and a space:

```
[transform.xRotation, transform.yRotation,
```

7. Pick Whip Z Rotation.

```
[transform.xRotation, transform.yRotation, transform.
zRotation
```

8. Type a close bracket:

```
[transform.xRotation, transform.yRotation, transform.
zRotation]
```

Try rotating the layer around its different axis. As you do, it will also change its position.

That Expression works fine, but I find it a little hard to read. So here's my rewrite, using variables to clarify what's going on:

```
var xPosition = transform.xRotation;
var yPosition = transform.yRotation;
var zPosition = transform.zRotation;
[xPosition,yPosition,zPosition]
```


Now, just to prove to you that all work and no play makes Marcus a dull boy, I'm going to show you a great practical joke you can play on an AE-using friend. First ask him, "Have you read *After Effects Expressions*?" If he says no, you're on:

1. Wait until he leaves his desk for a minute and then quickly reveal the Position property of any layer in his Comp. (You can practice with Chapter02.aep, Comp4 or any Comp with a solid in it.)
2. Add an Expression to Position.
3. Type an open Bracket.
4. Pick Whip Position's y value:

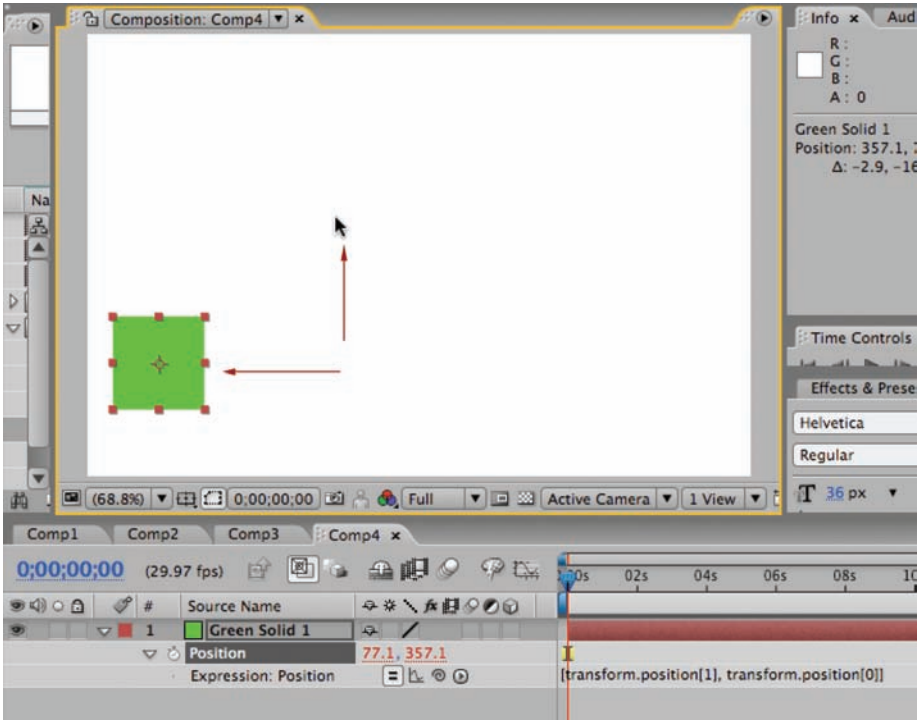
```
[transform.position[1]
```

Yes, you're controlling Position with it's own y value.

5. Type a comma and a space.
6. Pick Whip Position's x value.
7. Type a close bracket:

```
[transform.position[1], transform.position[0]]
```

8. Quickly hide the Position property and dash back to your seat. When your friend returns, he'll try to drag the layer up and down, but it will move left and right. He'll try to drag it left and right, but it will move up and down.



Why does this trick work? Because `transform.position[1]` is Position's y value and `transform.position[0]` is its x value. And you're listing them in the wrong order.

Here's a variation on the practical joke that I've actually used in a couple of projects:

1. Open Chapter02.aep, Comp5, or create your own Comp with two solids in it, a red one and a green one
2. Select both layers and press the P key to reveal their Position properties.
3. Add an Expression to Red's position.
4. Type an open bracket; then Pick Whip Green's y-position value:

```
[thisComp.layer("Green").transform.position[1]
```

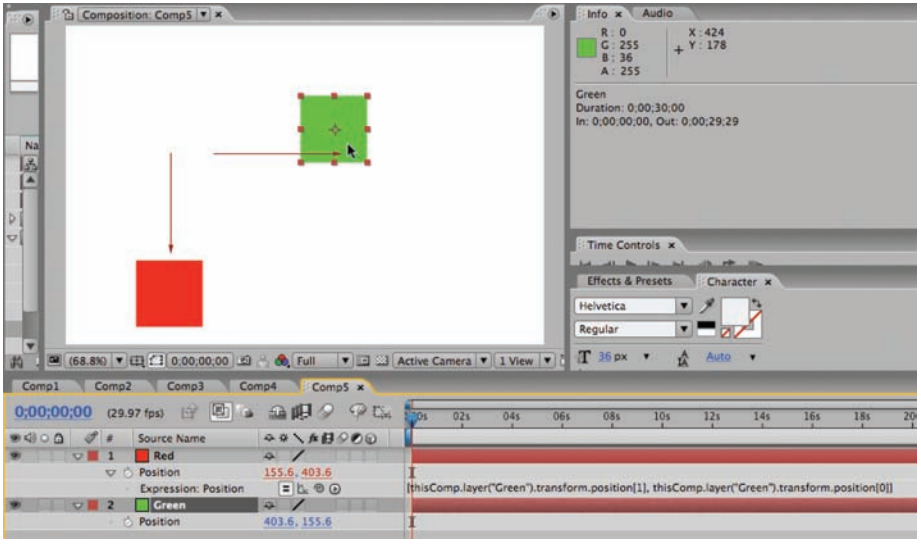
5. Type a comma and a space, Pick Whip Green's x-position value, then type a close bracket:

```
[thisComp.layer("Green").transform.position[1], this
Comp.layer("Green").transform.position[0]]
```

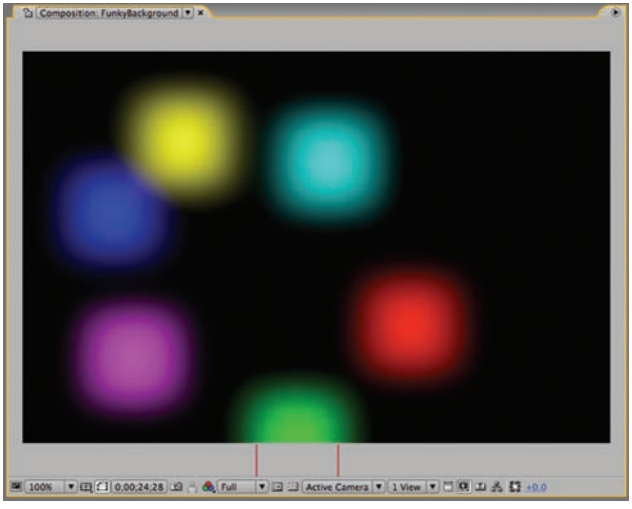
6. Drag the green layer around. The red layer will mirror it, but in an odd way. When Green moves up and down, Red moves left and right; when Green moves left and right, Red moves up and down. If you understand the Expression, you know what's going on, but to the casual observer, the layers just seem to have an interesting relationship.

Here's a rewrite of the Expression, for clarity:

```
var xPosition = thisComp.layer("Green").transform.
position[1];//green's y
var yPosition = thisComp.layer("Green").transform.
position[0];//green's x
[xPosition, yPosition]
```

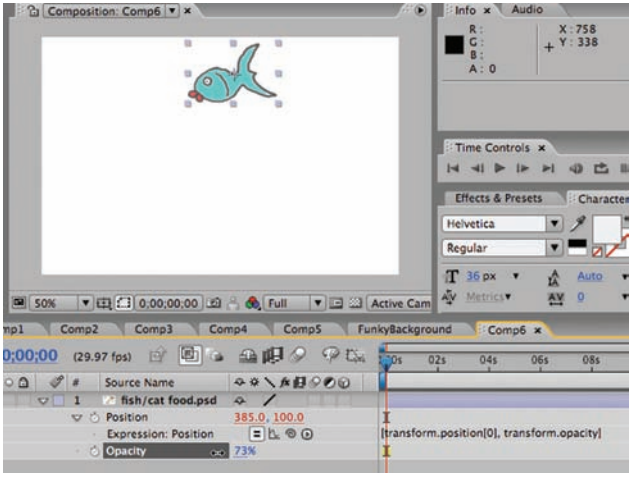



Using this Comp as a starting point, I added two more pairs of layers: a Blue and Yellow pair that mirror each other and a Cyan and Magenta pair that mirror each other. I then animated the "leader" layers (the ones without Expressions: Green, Blue, and Cyan) moving randomly around the screen. Their mirrors followed suit. Finally, I topped all the dancing layers with an adjustment layer, to which I added a Gaussian blur. You can check out the result in Chapter02.aep, FunkyBackground.



Before moving on, I'd like to suggest some simple variations of the sort of array Expression we've been making. Starting with a one-layer Comp, such as Chapter02.aep, Comp6, try adding the following Expression to Position:

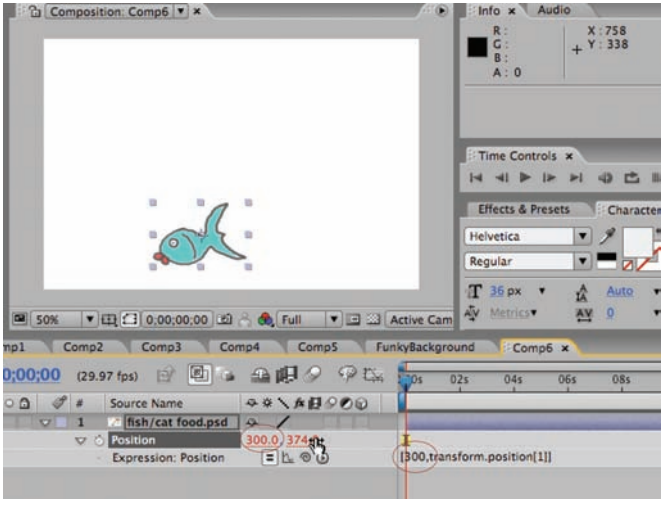
```
[transform.position[0], transform.opacity]
```



You'll be able to drag this layer left and right, because its x is value `transform.position[0]`, which is its x. It's like saying, "let this layer's x be whatever it is." On the other hand, you won't be able to drag it up and down. Instead, scrub its Opacity value to change the layer's y Position.

Here's another variation:

```
[300,transform.position[1]]
```



You won't be able to drag this layer left and right. It's x Position is frozen at 300. On the other hand, you can drag it up and down.

One more:

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

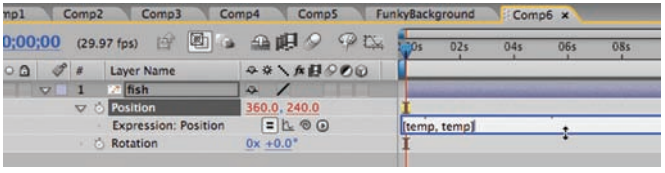
You can't drag this layer in either dimension, but you can scrub Opacity to move it left and right; and you can scrub Rotation to move it up and down.

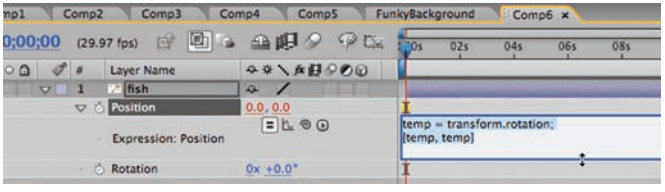
Now that you understand arrays, variables, and dimensions, I can finally show you what happens when you add an Expression to a two-dimensional property and Pick Whip a one-dimensional property:

1. Staying with our current Comp, remove the Expression from Position.
2. If necessary, press Shift+R to reveal the Rotation property.
3. Add an Expression back to Position.
4. Pick Whip Rotation.

AE adds an Expression to Position. It's two lines long, but you can only see the first line by default.

5. Point to the lower lip of the text-entry area. When your cursor becomes a double-headed arrow, drag the lip downward to reveal the whole Expression:





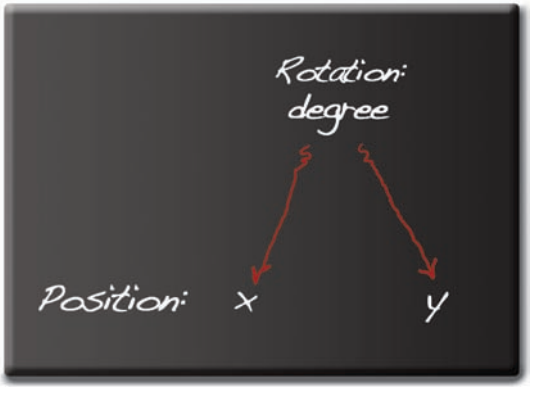

```
temp = transform.rotation;
[temp,temp]
```


AE used a variable (with the boring name of "temp") to hold the value of Rotation. If it had skipped this step, it could write the Expression as

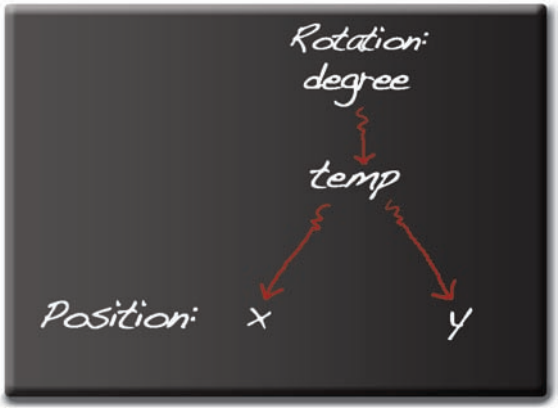
```
[transform.rotation, transform.rotation]
```

6. Rotate the layer, and you'll see it move diagonally. This is because the same value (Rotation) is being fed to both x and y. So if Rotation is 45 degrees, Position is 45x, 45y.

You can think of Rotation's relationship with Position like this:



And if you factor in the variable temp, you get this:



The variable version is useful if you want to add additional calculations to the Expression:

```
temp = transform.rotation + 10;
[temp,temp]
```

That's easier to write, make changes to, and understand than

```
[tranform.rotation + 10, transform.rotation + 10]
```


If you use this second version and decide to change the 10 to a 20, you might accidentally forget to change both 10s:

```
[tranform.rotation + 20, transform.rotation + 10] //oops!
```

If you stick to the variable version, you can change the 10 in one place, and both dimensions will be updated.

Before we move on from dimensions (and from this chapter), a word about color. I mentioned that AE sees a color as four-dimensional. Getting more specific, it sees color like this:

```
[red, green, blue, alpha]
```

(Alpha is the same as opacity.)

Color is an array of four values. In a real Expression, those values would have to be numbers, not the words "red," "green," "blue," and "alpha." But what numbers?

If you've spent years working with the RGB-color system, you're probably used to thinking of each color's value as being a number between 0 and 255. Alas, AE doesn't work this way. It sees each color's value as a number between 0 and

1. 0 means "none of that color," whereas 1 means "100% of that color." For instance,

```
[1, 0, 0, 1]
```

means red: 100% red, no green, no blue, and completely opaque (100% for alpha);

```
[1, 0, 0, 0]
```

is also red, but it's invisible, because the alpha is 0;

```
[1, 0, 0, .5]
```

is 50% transparent red.

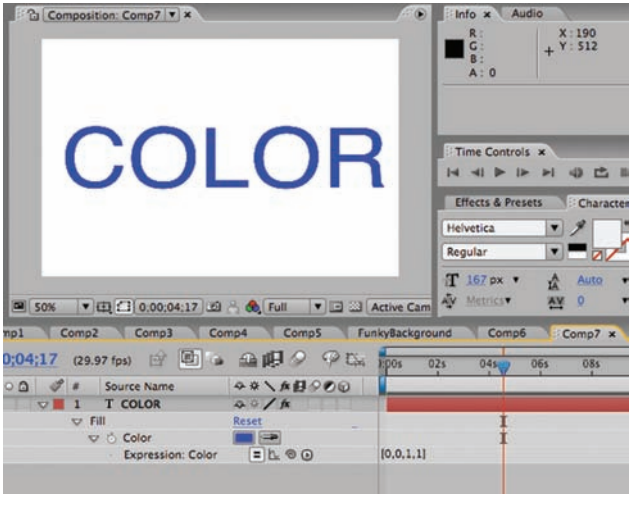
`[0, 0, 1, 1]` is blue.
`[0, 0, 0, 1]` is black.
`[1, 1, 1, 1]` is white.
`[1, 0, 1, 1]` is magenta.
`[.2, .2, .2, 1]` is a very dark shade of gray.

To test out this wacky version of color, open Chapter2.aep, Comp7 (or any Comp containing a solid):

1. Select the layer and, from the menu, choose Effect > Generate > Fill.

   Fill is a really simple effect that colorizes all opaque pixels in a layer with a color of your choice.

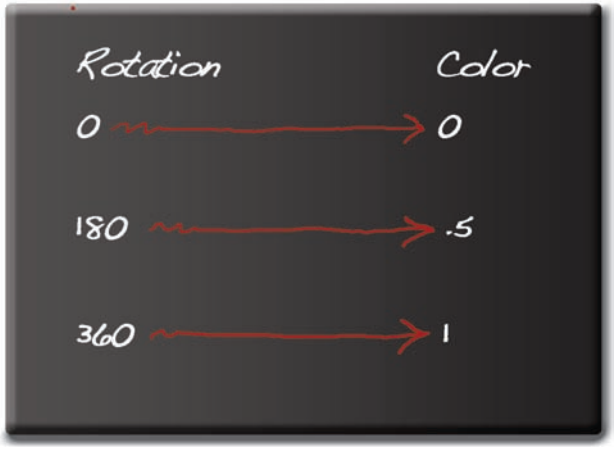
2. In the Effect Controls panel, add an Expression to Fill's Color property.
3. In the Timeline, enter the Expression `[0,0,1,1]`.



The layer will turn blue. You can now play around with some of the color Expressions we just went through (and some more of your own), but note that the Fill effect ignores the alpha value. You can set it to 0, 1, or something in between. It won't make any difference. The color will always be opaque.

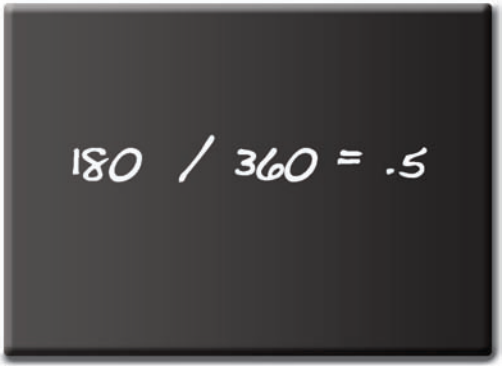
It would be fun to hook those color values up to other properties, such as Rotation. For instance, you could write an Expression that makes a layer get redder as you rotate it. The problem is that red needs to be a value between 0 and 1, but Rotation is going to spit out values much higher than that. There are various ways to deal with this problem. For the time being, we'll solve it by using an equation. Don't worry right now if you don't understand how it works.

Let's say you have a range of numbers, such as 0 to 360. You'd like to convert those numbers to the range 0 to 1, as follows:



Note: In AE, Rotation can go over 360. I'm going to assume here that you're not going to let that happen. (You'll just have to restrain yourself from rotating the layer more than one time around. You'll also have to restrain yourself from rotating it counterclockwise, because that yields negative numbers for degrees.)

To make the conversion, you take the Rotation value and divide it by 360. For example, if Rotation is 180 degrees:

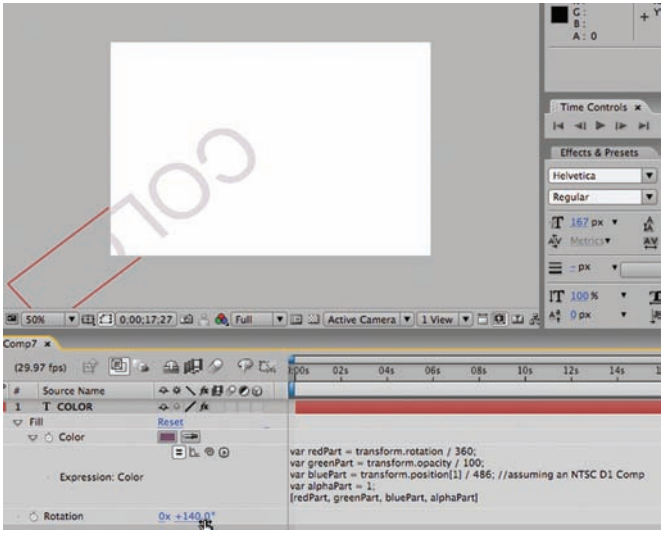


Generalizing this equation, to make it more useful, we get

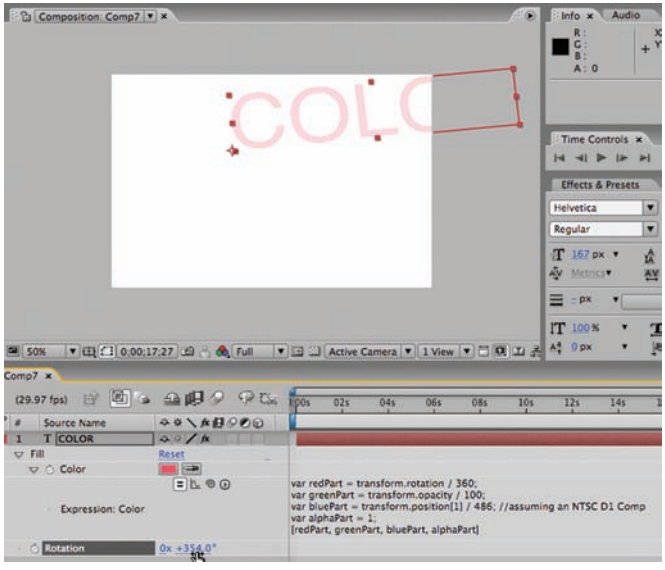
```
ORIGINAL_VALUE/MAXIMUM_VALUE = NEW VALUE
```

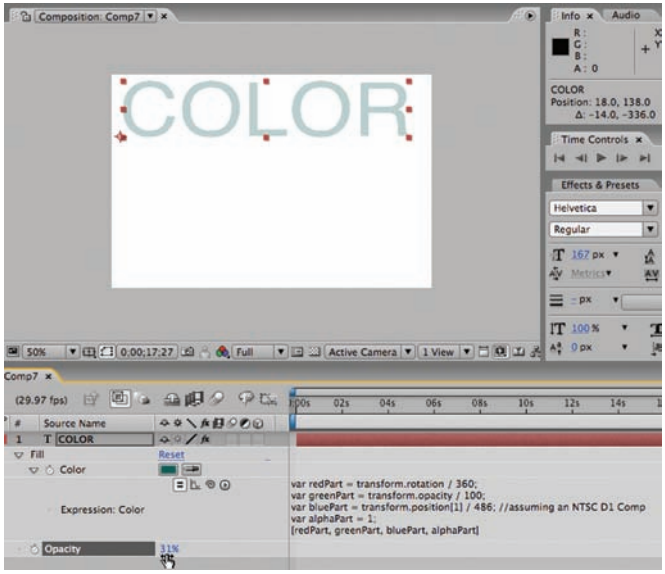
Putting this to use, let's change the Expression to

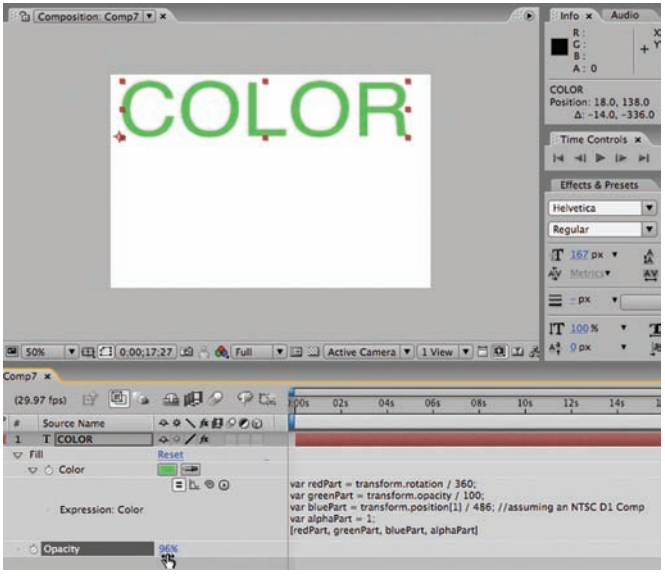
```
var redPart = transform.rotation/360;
var greenPart = transform.pacity/100;
var bluePart = transform.position[1]/486; //assuming an NTSC
D1 Comp
var alphaPart = 1;
[redPart, greenPart, bluePart, alphaPart]
```

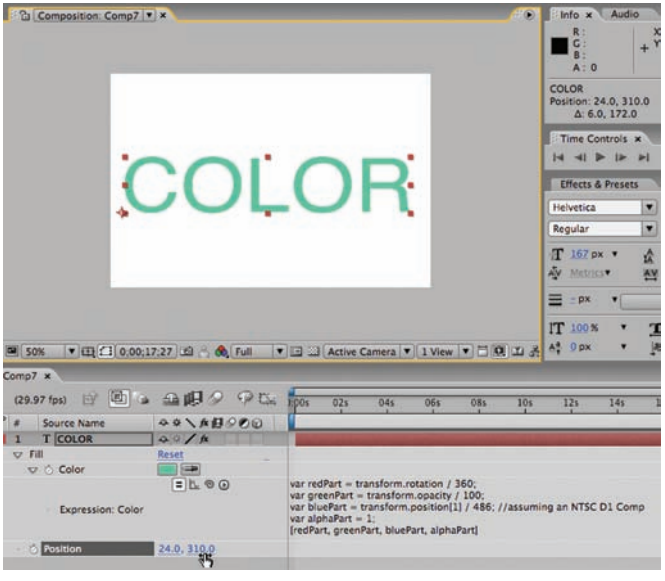



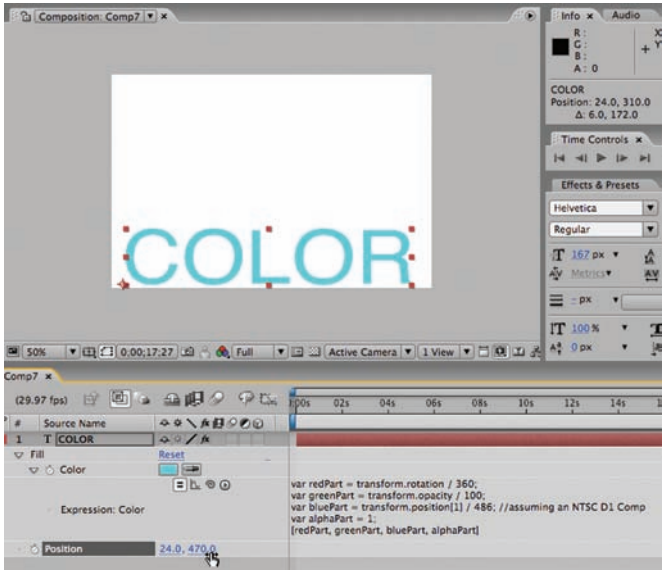
If you rotate the layer clockwise, it will get redder; if you make it more opaque, it will get greener; if you move it down, it will get bluer:


Composition: Comp7
COLO
Info
Audio
Time Controls
Effects & Presets
Helvetica
Regular
Comp7
Source Name
COLOR
Fill
Reset
Color
Expression: Color
Rotation
var redPart = transform.rotation / 360;
var greenPart = transform.opacity / 100;
var bluePart = transform.position[1] / 486; //assuming an NTSC D1 Comp
var alphaPart = 1;
[redPart, greenPart, bluePart, alphaPart]


Composition: Comp7
COLOR
Info
Audio
COLOR
Position: 18.0, 138.0
Δ: -14.0, -336.0
Time Controls
Effects & Presets
Helvetica
Regular
Comp7
Source Name
COLOR
Fill
Reset
Color
Expression: Color
Opacity
31%
var redPart = transform.rotation / 360;
var greenPart = transform.opacity / 100;
var bluePart = transform.position[1] / 486; //assuming an NTSC D1 Comp
var alphaPart = 1;
[redPart, greenPart, bluePart, alphaPart]


Composition: Comp7
COLOR
Info
Audio
R :
G :
B :
A : 0
COLOR
Position: 18.0, 138.0
Δ: -14.0, -336.0
Time Controls
Effects & Presets
Helvetica
Regular
167 px
100 %
0 px
50%
0;00;17;27
Full
Active Camera
1 View
Comp7
(29.97 fps)
Source Name
1 T COLOR
Fill
Reset
Color
Expression: Color
var redPart = transform.rotation / 360;
var greenPart = transform.opacity / 100;
var bluePart = transform.position[1] / 486; //assuming an NTSC D1 Comp
var alphaPart = 1;
[redPart, greenPart, bluePart, alphaPart]
Opacity
90%


Composition: Comp7
COLOR
Info
Audio
R :
G :
B :
A : 0
COLOR
Position: 24.0, 310.0
Δ: 6.0, 172.0
Time Controls
Effects & Presets
Helvetica
Regular
167 px
100 %
0 px
50%
0;00;17;27
Full
Active Camera
1 View
Comp7
(29.97 fps)
Source Name
1 T COLOR
Fill
Reset
Color
Expression: Color
var redPart = transform.rotation / 360;
var greenPart = transform.opacity / 100;
var bluePart = transform.position[1] / 486; //assuming an NTSC D1 Comp
var alphaPart = 1;
[redPart, greenPart, bluePart, alphaPart]
Position
24.0, 310.0



There's a big problem with this conversion method: it only works if both ranges start at 0 (0 to 360, 0 to 1). It will fail if, say, a range runs from 10 to 110 or −200 to 200. Never fear. We'll solve that problem in Chapter 3.