Tom 13/2021, ss. 185-196
ISSN 0860-5637
e-ISSN 2657-7704
DOI: 10.19251/rtnp/2021.13(7)
www.rtnp.mazowiecka.edu.pl

Marta Lipnicka
Uniwersytet Łódzki

Artur Lipnicki
Uniwersytet Łódzki

Methods for creating animations based on physical phenomena in JavaScript

Metody tworzenia animacji opartych na zjawiskach fizycznych w JavaScript

Summary: After Effects (AE) is a great tool for prototyping very advanced animations, but we are always looking for ways to speed up our workflow and we strive to simplify editing. AE comes with support for expressions which create relationships between composition properties or keyframes so the designer can animate layers without defining each keyframe by hand. The work contains some instructions regarding modeling and animation of physical phenomena such as reflections from the surface, including friction forces etc. The received functions allow their use in dimensions 2D and 3D.
Keywords: Lattice, Key frames, Covering radius, Expressions
Streszczenie: After Effects (AE) to znakomite narzędzie do tworzenia prototypów bardzo zaawansowanych animacji. Zawsze szukamy sposobów na przyspieszenie naszej pracy i staramy się uprościć edycję. Środowisko

AE obsługuje wyrażenia, które tworzą relacje między właściwościami kompozycji lub klatkami kluczowymi, dzięki czemu projektant może animować warstwy bez ręcznego definiowania każdej klatki kluczowej. Praca zawiera pewne instrukcje i skrypty dotyczące modelowania i animacji zjawisk fizycznych, takich jak odbicia od powierzchni, w tym siły tarcia itp. Otrzymane funkcje pozwalają na ich wykorzystanie w wymiarach 2D i 3D.
Słowa kluczowe: krata, klatki kluczowe, promień pokrywający, ekspresje

1. Introduction

The expression language is based on the standard JavaScript language, but JavaScript uses expressions. We can accept that in JavaScript, a value stored in an object is called a property. However, AE uses the term property to refer to layer components (solid, vector, null objects) as defined in the *Timeline* panel. For this reason, AE refers to JavaScript properties as either methods or attributes. In general, the difference between a method and an attribute is that a method usually does something to create its output value, whereas an attribute simply refers to an existing value to determine its output value. We can tell a method from an attribute most easily by looking for the parentheses following the method name, which surrounds any input arguments to the method.

An object (as solid, null, vector) is an item that can contain other objects, attributes and methods. Specifically, compositions are global objects, which means that they can be referred to in any context without reference to some higher-level object.

Once we have written an expression (functions), we can save it for future use by saving it in an animation preset or template project. However, because expressions are written in relation to other layers (solid, vector etc.) in a project and may use specific layer names, we must sometimes modify an expression to transfer it between other projects. We can create expressions by using the pick whip or by copying simple examples and modifying them to suit our needs (see [Christiansen, 2013; Geduld, 2013; Lipnicki and Drozda (red.), 2020]).

2. Movement model

We can use a pendulum where we will apply our decaying sine wave expression to the rotation property. Let's denote frequency as slider1 (freq), amplitude as slider2 (amp) and decay as slider3 (dec). We can write:

```
freq = thisComp.layer(”Null 2”).effect(”freq”)(”Slider”);
amplitude = thisComp.layer(”Null 2”).e ffect(”amp”)(”Slider”);
decay = thisComp.layer(”Null 2”).e ffect(”dec”)(”Slider”);
//we take the value zero
amplitude * Math.sin(freq * time * 2 * Math.PI)/ Math.
exp(decay * time)
```

If we want to receive motion with a certain delay so that the oscillations disappear, we can write our function as follows:

```
freq = thisComp.layer (”Null 2”).effect(”freq”)(”Slider”);
amplitude = thisComp.layer (”Null 2”).effect(”amp”)(”Slider”);
decay = time * thisComp.layer(”Null 2”).effect (”dec”)
(”Slider”);
amplitude * Math.sin(freq * time * 2 * Math.PI)/ Math.exp
(decay * time)
```

Consider now sine wave for position to simulate a bouncing ball. You will notice that we are using the cosine wave (so that our animation starts at its peak value at time zero). In this case we can write

```
freq = thisComp.layer (”Null 2”).effect(”freq”)(”Slider”);
amplitude = thisComp.layer (”Null 2”).effect(”amp”)(”Slider”);
decay = thisComp.layer (”Null 2”).effect (”dec”)(”Slider”);
pos = Math.abs(Math.cos (freq * time * 2 * Math.PI));
y = amplitude * pos /Math.exp (decay * time);
position – [0,y]
```

But we note that in this case we do not modify the scale in any way. Thus, we do not get any crushing of our particle when falling. So our function is not good yet. If it squashes in the vertical direction, it gets wider in the horizontal direction at the same time. The main idea in this situation is the fact that in the case of such simple objects such as a rectangle in the case of "splash" the dimension changes, but the area does not change. Thus, between the percentage scales of changing the proportion of "new" sides there is a relationship:

```
1=x _ new _ scale /100 * y _ new _ scale / 100.
```

Thus we see that

```
y _ new _ scale=1/x _ new _ scale * 1000.
```

So we can take advantage of this and write the following function

```
freq = thisComp.layer("Null 2").effect ("freq")("Slider");
amplitude = thisComp.layer("Null 2").effect("amp")("Slider");
decay = thisComp.layer("Null 2").effect("dec")("Slider");
t = time - inPoint;
x = scale[0] + amplitude * Math.sin(freq * t * 2 * Math.PI)/
Math.exp(decay * t);
y = (1/x) * 10000;
```

3. Elliptical paths

We will begin the analysis of the problem of parameterization of the animation by analyzing the example of the expression structure for the needs of animation of orbiting the molecule (for the "circle" effect on the "solid" layer). Let us first determine that our molecule will move in an elliptical orbit 2D without maintaining Kepler's laws. In order to prevent overly large

calculations for rendering, all expressions themselves will refer to the "circle" effect. In this way, we avoid rendering the entire vector layer (properly "solid" type). So the parametric description of the path is simple:

```
angle=ang;
rad1=val _ 1;
rad2=val _ 2;
nx=rad1 * Math.cos(angle);
ny=rad2 * Math.sin(angle);
[nx,ny] + value
```

where `rad1` and `rad2` are path radii respectively and "angle" means the angle in the parametric description. Here we can also automate the ray value process by setting the value on the null layer as the slider. We will then get the following expression

```
angle = thisComp.layer("sliders").effect("slider _ ang")
("Slder");
rad1 = thisComp.layer("sliders").effect("slider _ radius1")
("Slider");
rad2 = thisComp.layer("sliders").effect("slider _ radius2")
("Slider");
nx=rad1 * Math.cos(angle);
ny=rad2 * Math.sin(angle);
[nx,ny] + value
```

But of course in this case we are forced to create our entire animation structure by creating keyframes. We will use the so-called "sliders" as equivalents of ellipse rays and angle. If we would like to receive a spiral motion (the molecule moves away from the center in a spiral-elliptical motion), it would be enough to consider the function:

```
angle=time;
rad1=time*thisComp.layer("sliders").effect("slider _ radius1")
("Slider");
rad2=time * thisComp.layer("sliders" ).effect("slider _
radius2")("Slider") ;
nx=rad1 * Math.cos(angle);
ny=rad2 * Math.sin(angle);
[nx,ny] + value
```

If we want to control the angle (speed versus time) then we can add a slider to the appropriate layer

```
angle=time * thisComp.layer("sliders");
rad1=time * thisComp.layer("sliders").effect("slider _ radius1")
("Slider");
rad2=time * thisComp.layer("sliders").effect("slider _ radius2")
("Slider");
nx=rad1 * Math.cos(angle);
ny=rad2 * Math.sin(angle);
[nx,ny] + value
```

But if we would like our "particle" not to spiral away we would have to stop in some way the arguments about its radius. So let's set the type arguments

```
rad1 = const 1;
rad2 = const 2;
rad1 $ \not=$ rad2;
```

We then receive:

```
angle=time * thisComp.layer("sliders").effect("slider _ speed _
time")("Slider");
rad1=thisComp.layer("sliders").effect("slider _ radius1")("Slider");
rad2=thisComp.layer("sliders").effect("slider _ radius 2")("Slider");
nx=rad1 * Math.cos(angle);
```

```
ny=rad2 * Math.sin(angle);
[nx,ny] + value
```

where radius 1 and radius 2 are given by the user using the “slider” function. If the random function is used, the time parameter is obtained

```
value=random(a,b) ;
value=Math.round(value)
```

or

```
value=random(a,b);
value=Math.ceil(value) \\or value=Math.f loor(value)
```

We can also consider randomness in relation to set sliders. More precisely, define two values for which we get the range from which we randomize the value. The expression may then be as follows

```
bv=thisComp.layer(“Null 1”).effect(“Slider Control”)(“Slider”);
sv=thisComp.layer(“Null 1”).effect(“Slider Control 2”)(“Slider”);
random(bv,sv)
```

If we want the random value to be an integer then we have to use the Math.round function additionally. Then we obtain the interval in which we randomize the integers,

```
bv=thisComp.layer(“Null 1”).effect(“Slider Control”)(“Slider”);
sv=thisComp.layer(“Null 1”).effect(“Slider Control 2”)(“Slider”);
rv=random(bv,sv);
nv=Math.round(rv)
```

In the above way, we can easily manage the full randomness of selected elements of our function. Similarly, we can define a random frequency for a wiggle. In this case, one of the

parameters will determine the frequency, and the other the range of random values. We can apply this script to controlled camera movements. In fact, it can also be used for a reverse film stabilization procedure. In this case, we can construct expressions in a character

```
frequency=thisComp.layer("Null 1").effect("Slider Control")
("Slider");
val1=thisComp.layer("Null 1").effect("Slider Control 2")("Slider");
rv=wiggle(bv,sv);
```

Using the above and taking into account the movement along the ellipse, we can also add an element of randomness there (angle, distance, level, etc.), then the script could take into account the values given by the slider function and look like this:

```
fr1=thisComp.layer("Null 2").effect("Slider Control")("Slider");
fr2=thisComp.layer("Null 1").effect("Slider Control 2")("Slider");
angle=time * thisComp.layer("sliders")*fr1;
rad1=time * thisComp.layer("sliders").effect("slider _ radius1")
("Slider");
rad2=time * thisComp.layer("sliders").effect("slider _ radius2")
("Slider");
nx=rad1 * Math.cos(angle);
ny=rad2 * Math.sin(angle);
[nx,ny] + value
```

But in the above cases, we should keep in mind the accuracy of our data. For the *n* dimensional vector, we get the correct data with accuracy. Of course, sometimes nature requires some randomness in events. We can then use the classic random and wiggle functions. In the case of taking care of integer values, we must take into account a way of approximating such real values with integers. When a pair of numbers (a point in the coordinate system or in space) is drawn, what accuracy can we face (this

problem was raised in the paper). We get a very interesting case when we ask for integer- numerical approximations of the *n*-dimensional table values. How exactly can we describe the *n* - dimensional array of real numbers with an integer array. We can ask this question in the context of the lattice theory. It is not difficult to see (more in [Banaszczyk and Lipnicki (red.), 2015]) that by choosing a certain isomorphism, we can reduce the problem to the lattice of some polynomials on the interval [0, 1]. So we assume that *m, n, r* are non-negative integers. We denote by $\boldsymbol{P}_n$ the space of polynomials of degree *n* on the interval [0, 1] with the established norm (in this paper we consider uniform or Euclidean norm). Let $\boldsymbol{P}^{\mathbb{Z}}$ be the additive subgroup of the space $\boldsymbol{P}_n$ consisting of polynomials with integer coefficients. Let $\boldsymbol{M}_r$ be the space of all polynomials divisible by the polynomial $x^r(1-x)^r$. When $n \leq 2r-1$, then $P_n \cap M_r = \{0\}$, therefore, we will still to assume that $n \geq 2r$. Let us denote

$$\gamma_{r,n} := \max_{P \in P_n \cap H_r} d(P, P_n^Z)$$

In other words $\gamma_{r,n}$, is covering radius of the lattice $P_n^Z \cap M_r$ with norm $L_p(0,1)$. Our question concerning the accuracy of approximation level can be stated as follows: what are the values of $\gamma_{r,n}$? We can prove that (W. Banaszczyk and A. Lipnicki proved – see [Banaszczyk and Lipnicki (red.), 2015]) the following inequalities as $n \to \infty$. More information can be found in [Banaszczyk and Lipnicki (red.), 2015; Lipnicki, 2016]. By specifying a certain type of lattice (the space from which we draw the values), we can get the approximate error of the correct selection of this value. Multidimensional lattice then corresponds to many variables for a given type of draw. In this way we get a fairly good error estimate.

4. Inertial Bounce

Now consider the instruction giving motion trial expressions. In the case of animation, we can use it together with other expressions related to physics. We get the impression of an echo of an object moving along a specific path. Suppose we have a given layer of the null object type with the slider effect. Let slider delay mean number of frames to delay, value integer means an integer that reduces the index value. So our function on the position parameter has the form:

```
delay = thisComp.l ayer(″Null 1″).effect(″delay″)(″Slider″);
vi=thisComp.layer(″Null 1″).effect(″value _ integer″)(″Slider″);
value _ d = delay * thisComp.frame Duration * (index – vi);
thisComp.layer(1).position.valueAtTime(time-value _ d)
```

In the case of the opacity parameter, we assume that the null layer contains two sliders: *opacity _ factor* and *opacity _value_ integer* for integer-index values. So our function for the opacity parameter will take the form:

```
opacityFactor=thisComp.layer(″Null 1″).effect(″opacity _
factor″)(″Slider″);
id=thisComp.layer(″Null 1″) .effect(″opacity _ value _ integer″)
(″Slider″);
Math.pow(opacity Factor, index – id) * 100
```

However, if we want the values of arguments (sliders) to change within a certain range (e.g. linear), we can use the linear function like linear. We will now consider a problem in the animation and physics of the object such as "Inertial Bounce". The problem obviously concerns 2D and 3D objects. Our function is to simulate the phenomenon of rejection of object movement based on its speed. Therefore, we can use the function for the motion, rotation or rotation parameter. Suppose we have a given layer of the null object type and the slider effect.

We will then assign some arguments to our function to the slider data. Letting them then decay will mean friction (a larger value means a shorter time of decay). Then we introduce the amplitude markings as the parameter responsible for the reflection strength and freq is responsible for the frequency, i.e. how often it will reflect. Therefore, taking into account the parameters of our slider, we can define our function as follows:

```
decay= thisComp.layer(″Null 1″).effect(″decay″)(″Slider″);
amplitude = thisComp.layer(″Null 1″).effect(″value _ integer″)
(″Slider″);
freq = thisComp.layer(″Null 1″).effect(″opacity _ factor″)
(″Slider″);
nv = 0;
if(numKeys > 0)
{
nv = nearestKey(time).index;
if(key(nv .time>time)
{nv--;}
}
if(nv == 0)
{t=0;}
else
{t=time – key(nv).time;}
if(nv>0)
{
v = velocity At Time(key(nv).time – thisComp.frame
Duration/10);
value + v * amplitude * Math.sin(freq * t * 2 * Math.PI)/
Math.exp(decay * t);
}
else{value}
```

References

Banaszczyk W., Lipnicki A. 2015. On the lattice of polynomials with integer coefficients: the covering radius in $L_p(0, 1)$. Annales Polonici Mathematici vol. 115.2, pp. 123–144.

Christiansen M. 2013. Adobe After Effects CC Visual Effects and Compositing Studio Techniques. Adobe Press.

Geduld M. 2013. After Effects Expressions. Amazon Digital Services LLC.

Lipnicki A., Drozda J. Jr. 2020. JavaScript Function in Creating Animations, ICICI 2019, Springer Nature Switzerland AG 2020, LNDECT 38, pp. 1–8.

Lipnicki A. 2016. Uniform approximation by polynomials with integer coefficients, Opuscula Math. 36, no.4, pp. 489–49.