1.10

Critically Damped Ease-In/Ease-Out Smoothing

Thomas Lowe, Krome Studios
tomlowe@kromestudios.com

Smoothing is an enormously useful concept and can be of great service to the quality of every part of a game. Its use can make the difference between a game that looks rushed, jerky, and rough around the edges, and one that looks slick, polished, and natural.

By *smoothing*, we are referring to a method of gradually changing a value over time toward a desired goal. We can smooth almost any value that changes over time, whether represented by scalars, vectors, colors, or angles, and therefore the technique described here is widely applicable. Here are some examples:

- **Camera motion:** A camera that follows an object can exhibit jerky motion, especially when the object itself has jerky motion or when the camera collides with world geometry. Smoothing the camera gives a more natural look, and is much easier on the eyes of the player.
- **Flexible path following:** One can "smooth" toward a point that follows a path rather than following the path exactly. This will allow for deflection off objects en route. The technique will smooth sharp changes in the path's direction and can provide a smooth speedup and slowdown at the start and end of the path.
- **State changes:** When implementing a complex object's behavior, undesired and sudden changes in position or velocity can often arise, especially when moving from one state to another. Smoothing removes such glitches so that they no longer draw the player's attention.
- **Front end:** Moving text or an item from one part of the screen to another, changing its size or its color can all be done by smoothing toward a desired value.

This article describes a method of ease-in/ease-out smoothing based on a critically damped spring model. The model is implemented as an easy-to-use function, providing a robust and powerful smoothing tool.

Alternative Techniques

Given that smoothing has such widespread uses, it's worthwhile to consider several different methods. Let's start with a short comparison.

S Curve

When one simply needs to smooth from one static value to another over a specified time period, then an S curve gives a smooth ease-in/ease-out motion. Part of a sine wave or two parabolas can be used and provide C^1 continuity, although a C^2 curve (continuous acceleration) can be generated at little extra cost (see Figure 1.10.1).

Exponential Decay

Exponential decay is a common method and often looks something like:

```
y = y + (desiredY - y) * 0.1f * timeDelta
```

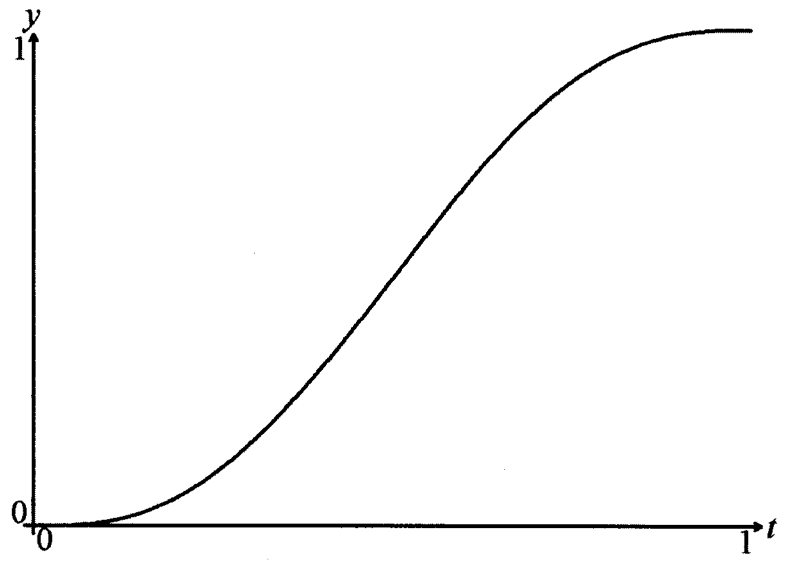

FIGURE 1.10.1 *C^2 S curve: $y = 6t^5 + 15t^4 - 10t^3$.*

where *timeDelta* is the time step between updates of the specified code, in seconds.

The value in this type of smoothing is that you can smooth toward a changing target; additionally, there is no need to retain the "time from start" data. This method can be described as "ease-out," but notice that the initial motion is sudden (see Figure 1.10.2).

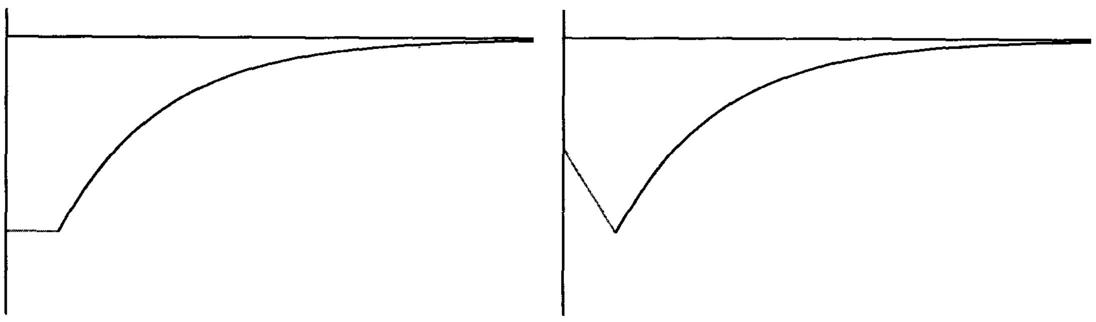

FIGURE 1.10.2 *Exponential decay, initially static and initially moving.*

Critically Damped Spring

This model combines the robustness of the exponential decay with the ease-in property of the S curve; one can smooth toward a changing target while maintaining a continuous velocity. Unlike the exponential decay technique, a velocity variable must be maintained (see Figure 1.10.3).

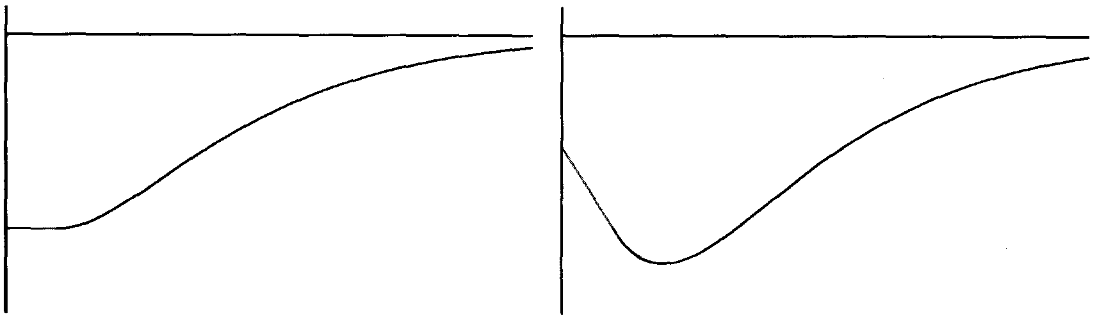

FIGURE 1.10.3 *Critically damped spring, no sudden change in velocity.*

Damped Springs and Critical Damping

How does the last method work? The technique is based on a damped spring (a spring with drag). A point at the end of a spring (y) has a force on it proportional to the extension from the spring's natural length (the desired position y_d); this is Hooke's law. A damped spring additionally has a force acting against the point's velocity. Therefore, the motion of a point y on a damped spring can be modeled by the differential equation:

$$m\frac{d^2y}{dt^2} = k(y_d - y) - b\frac{dy}{dt} \tag{1.10.1}$$

where m is the point's mass, k is the spring constant (or spring strength), and b is the damping constant (or amount of drag).

The constants affect how the spring recoils toward y_d, or in the context of our smoothing function, how our value approaches the desired value. A small value for b produces overshoot and oscillations, while large values for b give us a slow convergence. Critical damping occurs when b falls between these two extremes such that it produces no oscillations and approaches y_d at an optimal convergence rate. This can be shown to occur when $b^2 = 4mk$. Therefore, we can simplify Equation 1.10.1:

$$\frac{d^2y}{dt^2} = \omega^2(y_d - y) - 2\omega\frac{dy}{dt} \text{ where } \omega = \sqrt{\frac{k}{m}} \tag{1.10.2}$$

ω (omega) is the spring's natural frequency, or less formally a measure of the stiffness or strength of the spring.

In Practice

We will now show how to implement this model. Our goal is to write a function that will update a position and velocity given a desired position, a time period, and some type of smoothness factor—something like this:

```
y = SmoothCD(y, desiredY, velY, smoothness);
```

The critically damped spring model (Equation 1.10.2) can be approximated using standard numerical integration techniques, but there is really no need because an exact closed form (analytical) solution exists (see [Stone99]). It can be shown to be:

$$y(t) = y_d + ((y_0 - y_d) + (\dot{y}_0 + \omega(y_0 - y_d))t)e^{-\omega t} \tag{1.10.3}$$

where y_0 is the initial position and $\dot{y}_0$ is the initial gradient or velocity.

Applying this equation incrementally and differentiating, we get:

$$y_1 = y_d + ((y_0 - y_d) + (\dot{y}_0 + \omega(y_0 - y_d))\Delta t)e^{-\omega\Delta t} \tag{1.10.4}$$

$$\dot{y}_1 = (\dot{y}_0 - (\dot{y}_0 + \omega(y_0 - y_d))\omega\Delta t)e^{-\omega\Delta t} \tag{1.10.5}$$

These two equations give us (exactly) a new position and velocity after a time step Δt, precisely what we want.

Regarding our smoothness factor, note that we could use ω but it is usually more intuitive to control a smooth function with some type of *smooth time* rather than a spring strength. A good definition for our smooth time is "the expected time to reach the target when at maximum velocity" (see Figure 1.10.4). This definition is useful for two reasons. First, it is equivalent to the lag time (due to drag) when smoothing toward a moving target, making lag calculations easy. Second, it provides us with this simple conversion: $\omega = 2 / \textit{smooth time}$, which can be derived from Equation 1.10.2.

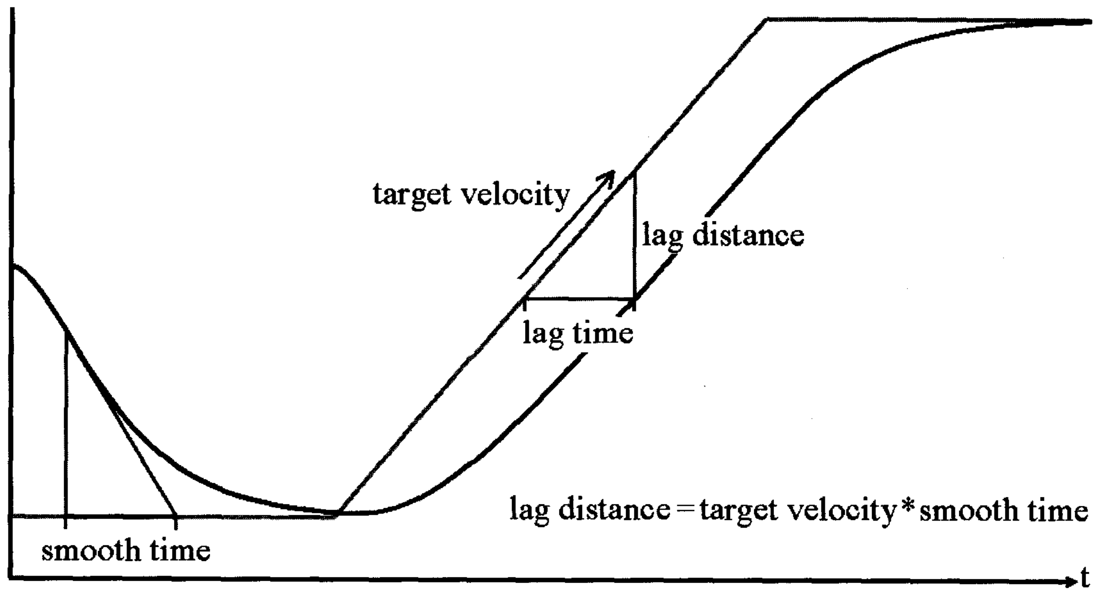

FIGURE 1.10.4 *Equivalence of smooth time and lag time.*

The one remaining problem is that exponential function call, which is computationally expensive. Fortunately, it can be approximated precisely for the range that we'll use. This results in our completed function.

```
float SmoothCD(float from,
               float to,
               float &vel,
               float smoothTime)
{
  float omega = 2.f/smoothTime;
  float x = omega*timeDelta;
  float exp = 1.f/(1.f+x+0.48f*x*x+0.235f*x*x*x);
  float change = from - to;
  float temp = (vel+omega*change)*timeDelta;
  vel = (vel - omega*temp)*exp;  // Equation 5
  return to + (change+temp)*exp; // Equation 4
}
```

A good basis for approximating e^x is to take a truncated Taylor expansion (shown below). Our exponential $e^{-w\Delta t}$ can then be calculated as $1/e^x$ where x is $w\Delta t$

$$e^x \approx \sum_{i=0}^{n} \frac{x^i}{i!} \tag{1.10.6}$$

The coefficients can be tweaked to better approximate within the most commonly used range. For our function, this range is roughly $0 < x < 1$ and here the approximation *exp* used above has less than 0.1% error. It is also roughly 80 times faster than the function `exp()` on PC! An even better approximation could be achieved by using higher order polynomials.

Adding a Maximum Smooth Speed

We will finish with a brief examination of a handy extension: how to add a maximum smooth speed. Since a source and target moving at speed *s* will have a lag distance equal to *s*smoothTime*, if our distance to target is clamped to be no longer than this lag distance, then *s* becomes the maximum speed.

This idea can be implemented by modifying *change* after it is set, leading to a smoothly approached top speed.

```
float maxChange = maxSpeed*smoothTime;
change = min(max(-maxChange, change), maxChange);
```

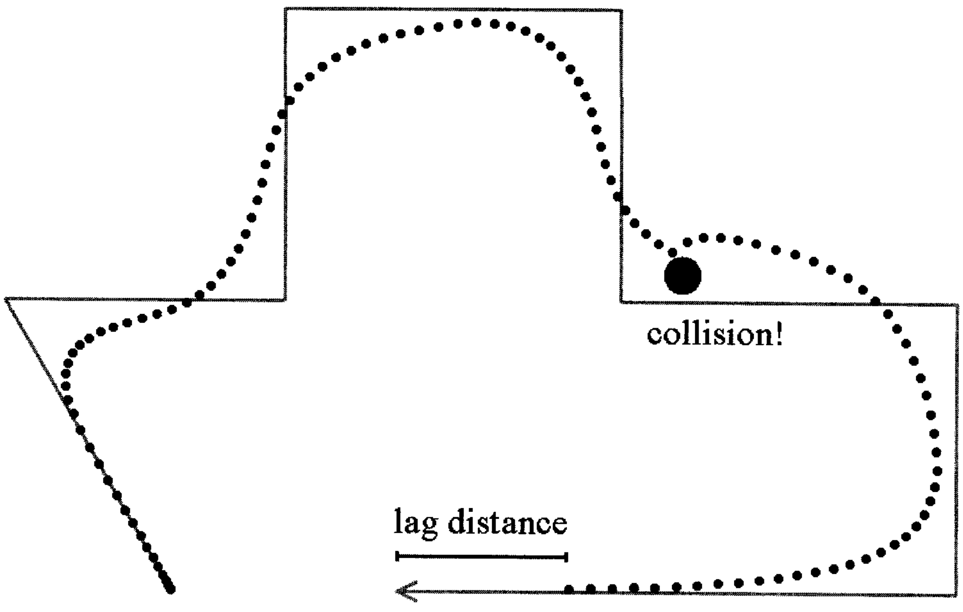

FIGURE 1.10.5 *Smoothing a vector, path following with deflection en route.*

Conclusion

ON THE CD

Critically damped spring smoothing can be achieved with a simple algorithm. The implementation shown here is a precise approximation to the exact solution and is always stable no matter how large the time step or small the smooth time. Since smoothing can be performed on floats, colors, vectors, and so on, it is an ideal candidate for a template function, and just such a version has been provided for you on the companion CD-ROM.

References

[Stone99] Stone, B. J., "A Summary of Basic Vibration Theory," available online at *www.mech.uwa.edu.au/bjs/Vibration/OneDOF/1DOF.pdf.*