Precise Trajectories in Games

Mario Binder

October 15, 2023

1 Problem

When calculating the trajectory of a moving physical object in Unity or other game engines, the most common way of achieving this is by numerical simulation. The number of iterations depends then on the time of flight and the abort condition. This looks, in code, more or less like so:

```
Vector3 currentPosition = this.transform.position;
Vector3 currentVelocity = shootVelocity;

while(currentPosition.y > targetHeight) {
        currentVelocity += acceleration * Time.fixedDeltaTime;
        currentVelocity *= (1.0f - drag * Time.fixedDeltaTime);
        currentPosition += currentVelocity * Time.fixedDeltaTime;
}
```

There are variations of this, but the things most implementations have in common is the loop. The problem is that it is not clear when it will terminate. If for example the object takes a minute to land on the ground, then for $\Delta t = 0.02$, we would have to iterate $\frac{60s}{0.02s} = 3000$ times!

Of course, the obvious solution is to increase Δt, however, the physics engine will still calculate the velocity each frame, leading to imprecise results.

Another problem comes along, when the hit position has to be known precisely (for example to place a target marker there). Now we do not only iterate an unknown amount of times, but also while doing **Raycasts**!

2 Solution

So the ideal solution would have fewer iterations **AND** a fixed amount of them. In other words, we want a closed formula for the object position at time t... which means we have to do mathematics.

Disclaimer: I am not a mathematician! The formulae will work, but I do not guarantee that they are already optimal. Also: The way in which I

found a solution might be pretty bad too. On the other hand, I might explain things a little bit simpler (than they are). Use and improve everything at your own discretion.

With that out of the way: Let's get started!

2.1 Tangent

Along the trajectory, the velocity describes the tangent of the curve. It is also the first derivative of the position formula we are searching for. The idea is, that when we have a closed formula for the tangent, we only need to integrate it.

First of all, I assume the engine uses linear drag, like in the code above, acting on the total velocity of an object. If that is the case, then we can write what is happening in each frame like so:

$$v_{n+1} = (v_n + a\Delta t)(1 - d\Delta t)$$

where v describes a velocity, a is a constant acceleration (usually *gravity*), d is the linear drag coefficient (defined by the settings of the rigid body), Δt is a constant fixed time step, and n is the current frame number. Notably, we have v_0 as our *start velocity* at frame 0, which can be chosen freely.

Ok, what now? Well, we can try to expand this series and see if we can spot a pattern. Here are some of the first terms:

$$v_1 = (v_0 + a\Delta t)(1 - d\Delta t) = v_0 + a\Delta t - v_0 d\Delta t - ad\Delta t^2 \tag{1}$$

$$\begin{aligned} v_2 &= (v_1 + a\Delta t)(1 - d\Delta t) = [(v_0 + a\Delta t - v_0 d\Delta t - ad\Delta t^2) + a\Delta t](1 - d\Delta t) \\ &= v_0 + 2a\Delta t - 2v_0 d\Delta t - 3ad\Delta t^2 + v_0 d^2\Delta t^2 + ad^2\Delta t^3 \end{aligned} \tag{2}$$

$$\begin{aligned} v_3 &= (v_2 + a\Delta t)(1 - d\Delta t) = [(v_0 + 2a\Delta t - ...) + a\Delta t](1 - d\Delta t) \\ &= v_0 + 3a\Delta t - 3v_0 d\Delta t - 6ad\Delta t^2 + 3v_0 d^2\Delta t^2 + 4ad^2\Delta t^3 - v_0 d^3\Delta t^3 - ad^3\Delta t^4 \end{aligned} \tag{3}$$

Ok, we have expanded it. Already looking complicated. But hold on, let's shuffle the terms a little bit, maybe we find some hints:

$$v_1 = v_0 - v_0 d\Delta t + a\Delta t - ad\Delta t^2 \tag{4}$$

$$v_2 = v_0 - 2v_0 d\Delta t + v_0 d^2\Delta t^2 + 2a\Delta t - 3ad\Delta t^2 + ad^2\Delta t^3 \tag{5}$$

$$v_3 = v_0 - 3v_0 d\Delta t + 3v_0 d^2\Delta t^2 - v_0 d^3\Delta t^3 + 3a\Delta t - 6ad\Delta t^2 + 4ad^2\Delta t^3 - ad^3\Delta t^4 \tag{6}$$

That looks a lot better already. First, notice that the velocity and acceleration are independent (i.e. each addition and subtraction only contains either v_0 or a but not both). Which is absolutely **fantastic**! This means, that if we can find a pattern, we can split it into **two**. One for how the velocity changes with time and drag ("*velocity over time*") and one for how the acceleration changes the velocity with time and drag ("*velocity change over time*").

2.1.1 Velocity Over Time

Let's start with *Velocity Over Time* v^t. Let's only consider the terms with v_0 now, and see how they change:
$v_1^t = v_0 - v_0 d\Delta t$
$v_2^t = v_0 - 2v_0 d\Delta t + v_0 d^2\Delta t^2$
$v_3^t = v_0 - 3v_0 d\Delta t + 3v_0 d^2\Delta t^2 - v_0 d^3\Delta t^3$

From school, you should quite quickly spot that the multiplication factors are the **binomial coefficients**. We can also see, that the + and − signs are alternating. Additionally, the power of the d and Δt in each term is simply increasing by one. To make it short: We can write it as a sum!

$$v_n^t = v_0 \sum_{k=0}^{n} \binom{n}{k} (-1)^k d^k \Delta t^k \tag{7}$$

Thinking about it... all we're doing is some binomial expansion, the same as $(a+b)^n$. And we do alternate the signs when we have $(a-b)^n$... and asking a math solver (at least I admit it) confirms our suspicion, that this can actually be simplified to:

$$v_n^t = v_0(1 - d\Delta t)^n \tag{8}$$

2.1.2 Velocity Change Over Time

It worked the first time, why wouldn't it again? Here are the terms:
$a_1^t = a\Delta t - ad\Delta t^2$
$a_2^t = 2a\Delta t - 3ad\Delta t^2 + ad^2\Delta t^3$
$a_3^t = 3a\Delta t - 6ad\Delta t^2 + 4ad^2\Delta t^3 - ad^3\Delta t^4$

Alright, there are still **binomial coefficients**, but they are more hidden. That is because they do not start at $\binom{n}{0}$ but at $\binom{n+1}{1}$. Additionally the first

term is missing $1a\Delta t$. Well, all we have to do then is to add one to the indices and then subtract the $a\Delta t$ we have added too much:

$$a_n^t = (a\sum_{k=0}^{n}\binom{n+1}{k+1}(-1)^k d^k \Delta t^{k+1}) - a\Delta t \tag{9}$$

A bit more complicated, but nothing a solver cannot handle:

$$a_n^t = \frac{a(d\Delta t(1-d\Delta t)^n - (1-d\Delta t)^n + 1)}{d} - a\Delta t \tag{10}$$

Hmm... ok, not really a nice formula, but still not too bad. And... we're actually done finding a closed formula for the tangent. All we have to do is to add the two sums together:

$$v_n = v_n^t + a_n^t = v_0(1-d\Delta t)^n + \frac{a(d\Delta t(1-d\Delta t)^n - (1-d\Delta t)^n + 1)}{d} - a\Delta t \tag{11}$$

So what that means, is that we can calculate the tangent for each frame. E.g. when an object is starting to move with v_0 at frame 0, and we want to know what velocity it has at frame 60, we simply set $n = 60$ and this formula will give us the answer!

More than that, we also can get accurate velocities and tangents **between** frames, by considering n to be an element of $\mathbb{R}$, i.e. a floating-point number. How cool is that?

3 Position

Now to find out how far the object has travelled at time t, we simply integrate the equation above (we = math solver). This gives us the area under the curve, which is the distance travelled (+ a constant, which is our starting position), like in classical mechanics.

However, first we should do the basics. Ok, so we want to integrate with respect to frame time t. Now we define the domain. In this case, we do not need and want the area under the curve for $t < 0$, so the integral should be a definitive one from 0 to an arbitrary target time in frames r_t (I had to choose *some* letter...).

Then... wait... we do not have t anywhere in our equation? Well, that is because we started with a series and frames, and are now going for continuity. But all we have to do is to rename a few variables and cross our fingers that it works out (Spoiler: It does).

I will rename n to t (we said that $n \in \mathbb{R}$, so that shouldn't be an issue). Now t and Δt is confusing, so we rename it to Δf (which stands for *frame time*).

Our equation now looks like this:

$$v_t = v_0(1 - d\Delta f)^t + \frac{a(d\Delta f(1 - d\Delta f)^t - (1 - d\Delta f)^t + 1)}{d} - a\Delta f \quad (12)$$

Now just put on the integral:

$$p_t = \int_0^{r_t} (v_0(1 - d\Delta f)^t + \frac{a(d\Delta f(1 - d\Delta f)^t - (1 - d\Delta f)^t + 1)}{d} - a\Delta f)dt \quad (13)$$

Do not confuse the dt at the end with $d * t$, dt just means we're integrating with respect to time (and here I thought naming variables was only a problem in programming).

Ok, so before we hand over control to a solver again, let us begin ourselves. First off, we can integrate each term separately:

$$\begin{aligned} p_t &= \int_0^{r_t} (v_0(1 - d\Delta f)^t + \frac{a(d\Delta f(1 - d\Delta f)^t - (1 - d\Delta f)^t + 1)}{d} - a\Delta f)dt \\ &= \int_0^{r_t} (v_0(1 - d\Delta f)^t)dt + \int_0^{r_t} (\frac{a(d\Delta f(1 - d\Delta f)^t - (1 - d\Delta f)^t + 1)}{d} - a\Delta f)dt \end{aligned} \quad (14)$$

After we have integrated each term we will get some constants c_0 and c_1. However, those are not really meaningful by themselves as we can essentially choose them freely without affecting the form of the position curve so to speak. But they **represent** the start position p_0 in our context, which is how I will write it in the final p_t-equation!

The first term integrated is:

$$\int_0^{r_t} (v_0(1 - d\Delta f)^t)dt = v_0 \int_0^{r_t} (1 - d\Delta f)^t dt \quad (15)$$

Now we use the fact that:

$\frac{d}{dx}a^x = ln(a)a^x$

Which means (even when using the quotient rule):

$\frac{d}{dt}\frac{(1-d\Delta f)^t}{ln(1-d\Delta f)} = (1 - d\Delta f)^t$

Therefore:

$$v_0 \int_0^{r_t} (1 - d\Delta f)^t dt = v_0 \frac{(1 - d\Delta f)^{r_t} - 1}{ln(1 - d\Delta f)} \quad (16)$$

The -1 stems from the fact that we integrate from 0 (definite integral). Now to the second one:

$$
\begin{aligned}
&\int_0^{r_t} (\frac{a(d\Delta f(1-d\Delta f)^t - (1-d\Delta f)^t + 1)}{d} - a\Delta f)dt \\
&= a\int_0^{r_t} (\frac{d\Delta f(1-d\Delta f)^t - (1-d\Delta f)^t + 1}{d} - \Delta f)dt \\
&= \frac{a}{d}\int_0^{r_t} (d\Delta f(1-d\Delta f)^t - (1-d\Delta f)^t + 1 - d\Delta f)dt \\
&= \frac{a}{d}\int_0^{r_t} ((d\Delta f - 1)(1-d\Delta f)^t - d\Delta f + 1)dt
\end{aligned}
\tag{17}
$$

Let us split it into parts:

$$
\begin{aligned}
&\int_0^{r_t} 1dt = r_t + c \\
&\int_0^{r_t} d\Delta f dt = d\Delta f r_t + c \\
&\int_0^{r_t} ((d\Delta f - 1)(1-d\Delta f)^t)dt = \int_0^{r_t} (1-d\Delta f)^t dt
\end{aligned}
\tag{18}
$$

The last one we already know from the v_0 part:

$$
(d\Delta f - 1)\int_0^{r_t} (1-d\Delta f)^t dt = (d\Delta f - 1)\frac{(1-d\Delta f)^{r_t} - 1}{ln(1-d\Delta f)} + c \tag{19}
$$

So combining the parts together we get:

$$
\frac{a}{d}[(d\Delta f - 1)\frac{(1-d\Delta f)^{r_t} - 1}{ln(1-d\Delta f)} - d\Delta f r_t + r_t] \tag{20}
$$

Which can be simplified into:

$$
\frac{a}{d}\frac{(d\Delta f - 1)[(1-d\Delta f)^{r_t} - r_t ln(1-d\Delta f) - 1]}{ln(1-d\Delta f)} \tag{21}
$$

So now we have both integrals. If we now do not forget to add the start position... then we are **done**! Are you ready? Here is the final equation for the position of an object affected by linear drag over time:

$$
p_t = p_0 + v_0\frac{(1-d\Delta f)^{r_t} - 1}{ln(1-d\Delta f)} + \frac{a}{d}\frac{(d\Delta f - 1)[(1-d\Delta f)^{r_t} - r_t ln(1-d\Delta f) - 1]}{ln(1-d\Delta f)} \tag{22}
$$

Heck yeah! What a beast! Because r_t will be used as *frames* in our methods in the code, I'll just write it one more time replacing r_t with f:

$$p_t = p_0 + v_0 \frac{(1 - d\Delta f)^f - 1}{ln(1 - d\Delta f)} + \frac{a}{d} \frac{(d\Delta f - 1)[(1 - d\Delta f)^f - fln(1 - d\Delta f) - 1]}{ln(1 - d\Delta f)} \tag{23}$$

Ok, so this will now give us the position of an object with a starting velocity, affected by drag, at a frame f. If we want to use the usual time $t = f * \Delta f$, we simply multiply each term (except the start position p_0) with Δf.

Now we have something we can work with... or can we? Because there is one tiny problem with that formula... when $d = 0$ we divide by zero. Luckily, this can be easily fixed by using Newton Physics in that case!
(This is how it looks like with our notation)

$$p_t = p_0 + v_0 t + \frac{1}{2} a t^2 \tag{24}$$

The other time we have a problem is when $d\Delta f = 1$ ($ln(0)$ is undefined). The default value for Δf in Unity is $\frac{1}{50}$, so the drag would have to be ≥ 50 before this problem occurs.

This is an example implementation of the equation in Unity (should be adaptable to most engines that use linear drag):

```
public static Vector3 GetPosition(Vector3 startPosition, Vector3 startVelocity,
                                  float drag, float time)
{
    Vector3 acceleration = Physics.gravity;
    Vector3 position = startPosition;

    if (drag > 0.0f)
    {

        float frameDrag = 1.0f - drag * Time.fixedDeltaTime;
        float frames = time / Time.fixedDeltaTime;

        float frameDragLog = (float)Math.Log(frameDrag);
        float frameDragPower = Mathf.Pow(frameDrag, frames);

        position += startVelocity * ((frameDragPower - 1.0f)
                / frameDragLog) * Time.fixedDeltaTime;

        float accDragFactor = (drag * Time.fixedDeltaTime - 1);
        accDragFactor *= (frameDragPower - frames * frameDragLog - 1.0f);
        accDragFactor /= (drag * frameDragLog);

        position += acceleration * accDragFactor * Time.fixedDeltaTime;

        return position;
    }
    else
    {
        //Newton Physics
        return position + startVelocity * time + 0.5f * acceleration * time * time;
    }
}
```

4 Root-finding

Now that we have removed the annoying while-loop, what else can we do? Turns out: **A lot!** We can, for example, figure out when an object is at a

certain height (Y-Value) a lot easier than before. Why? Because we can travel back and forwards in time in any interval we like. And that means we can use **root-finding algorithms**!

Why do we want to do this? So that instead of iterating an unknown amount of times (possibly hundreds of steps), we only iterate a fixed amount of time (and a lot less; the code below usually converges way before even 10 steps)! Additional bonus: It is incredibly precise (within reasonable time frames).

One of the algorithms we can use to find a root is **Newton's Method**. But for that we would need the derivative... which we already have, because we started with it. It's the tangent, so yeah. There is however something we have to be very careful about, and that is, that there may be **two** points fitting a Y-Value. One for the way up, and one for the way down.

We usually only care about the points on the way down, so what we can do, is to simply waste time until we fall. And we know when we fall, because that is when the tangent is < 0 (if you want to search for a height on the way up, the tangent has to be > 0).

Here the implementation:

```
public static float GetTimeForReachingYOnTheWayDown(Vector3 startPosition,
    Vector3 startVelocity, float drag, float targetY,
    float startTime = 1.0f, int iterations = 16, float epsilon = 0.01f)
{
    float time = startTime;
    Vector3 tangent = GetTangent(startVelocity, drag, time);

    for (int i = 0; i < iterations; i++)
    {
        //On the way up
        if (tangent.y >= 0.0f)
        {
            time *= 2.0f;
            tangent = GetTangent(startVelocity, drag, time);
        }
        else
        {
            //Using Newton's method
            var position = GetPosition(startPosition, startVelocity, drag, time);
            tangent = GetTangent(startVelocity, drag, time);

            if (Mathf.Abs(position.y - targetY) < epsilon) return time;

            time = time - ((position.y - targetY) / tangent.y);
        }
    }

    return time;
}
```

The roots are at height $= 0$, therefore we subtract the target height from the position. To spend time, I simply multiply it by 2 until the tangent is negative (which it might be from the start if you shoot down). Note that the algorithm converges faster if you can get a good estimate for the start time (you could cache the time when the tangent becomes < 0 for example... or you find the turning point analytically by setting the tangent equation equal to 0).

As for calculating the tangent:

```
public static Vector3 GetTangent(Vector3 startVelocity, float drag, float time)
```

```
{
    if (drag > 0.0f)
    {

        float frameDrag = 1.0f - drag * Time.fixedDeltaTime;
        float frames = time / Time.fixedDeltaTime;

        float frameDragPower = Mathf.Pow(frameDrag, frames);

        Vector3 tangent = startVelocity * frameDragPower;

        Vector3 accTangent = (Physics.gravity / drag) * (drag * Time.fixedDeltaTime
                            * frameDragPower - frameDragPower + 1.0f);
        accTangent -= Physics.gravity * Time.fixedDeltaTime;

        return (tangent + accTangent);
    }
    else
    {
        //Derrivative of the classic mechanics formula
        return (startVelocity + Physics.gravity * time);
    }
}
```

5 Raycast Strategy

It is a sad fact of life, that the closer you get to something in the world, the more flat it will appear. In a world full of polygons... well, you get the point. But we can use this to our advantage as well.

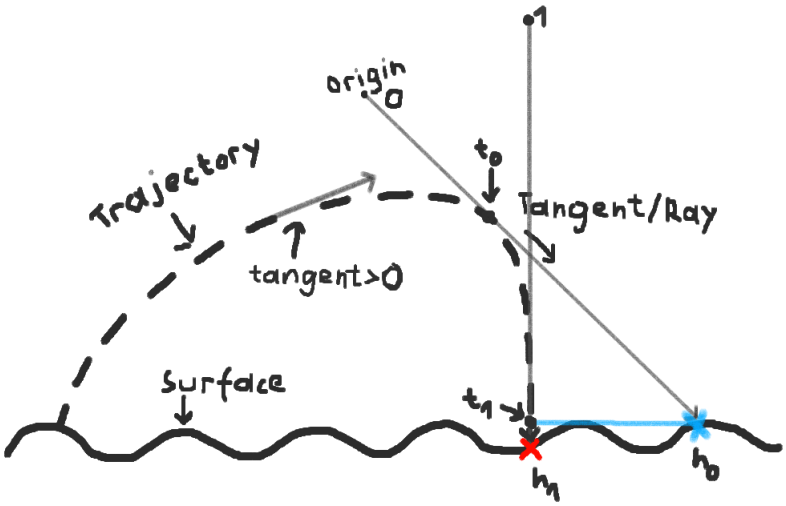
Often times, the target height were the object will land is not known. But minimizing the amount of Raycasts is important for the performance. Now that we can skip time, there are some cool new algorithms we can try. I assume a few things here: The acceleration is on the Y-Axis, the surface is generally speaking on the XZ-Plane and there are no obstacles in the air.

Then, a simple strategy could be to cast Raycasts along the tangent (the velocity vector) at a time t (when the tangent is < 0). If the position is p_t, then the origin of the cast should start quite some distance away from it in the opposite direction to the tangent (so that something is hit, even if p_t is below the surface).

If we do not hit anything, we just double the time again. But otherwise, just because we hit *something* with a Raycast does not mean it is the landing height h_l though, as the surface we hit, which could be very far away as well, might lie higher or lower. But what we can reasonably assume is that when we get very close to a surface, say $< \epsilon$, that it is going to be flat. What we also know, because the tangent is < 0, that the hit point of the cast is below our current position (which we calculate from time t).

Therefore, we can take the hit point Y-Value as an initial approximate guess for h_l. We then use the root-finding algorithm from before to find a new t-value. We then repeat this process, until the raycast hit point and the current position are equal or within a small distance.

Note, that when the current position p_t lies precisely on the target surface, then the result of a raycast with the tangent as direction, will return the same point p_t (i.e. the algorithm will abort in that case and the point is returned).



As you can see, we get very close to the actual h_l with very few iterations. Depending on the geometry of your world, this might reduce the amount of raycasts for calculations like these quite drastically.

6 Disadvantages

Despite the obvious advantages, there are also a few problems with this way of calculating trajectories, and you should be aware of them.

6.1 Precision

When the position you want to get is far ahead in the future, then the number of frames can get quite high. Because the evaluation then contains powers with large exponents, the results will get inaccurate. However, as the velocity tends to 0 as well over large time spans, the same as the power factor, the result remains stable at least.

By using *double* precision (very easy solution) we can delay that problem quite long though.

6.2 Performance

While the formula is of course a lot faster than an iterative approach over longer time periods, it might not be so at short ones. This is for one, because the calculation done in the loop is cheap, but also because in contrast to a root-finding algorithm, it can be vectorized. But really, it is not so much a disadvantage as more like something we can optimize for.

If only there was a way to estimate or take an educated guess for the number of iterations we have to take in that while-loop... well... that is actually precisely what you can also do with the formula (by taking some wild guesses at the time it is going to take when the object hits the ground and checking, or by simply saving the result from some earlier points in time)

Another situation, when the formula is not ideal, is when you want to calculate points for the complete trajectory (for example to get all positions as an input to a line renderer). In that case, the iterative approach is certainly faster. However, still, with the equation, we might increase the time step quite a bit without sacrificing precision. Additionally, with the equation, we can compute points in parallel. So estimating what is faster is not so easy, and requires experimentation.