Fix Your Timestep!

How to step your physics simulation forward

Posted by Glenn Fiedler (http://web.archive.org/web/20181107181440/https://gafferongames.com/about) on Thursday, June 10, 2004

Introduction

Hi, I'm Glenn Fiedler (http://web.archive.org/web/20181107181440/https://gafferongames.com/about) and welcome to **Game Physics (http://web.archive.org/web/20181107181440/https://gafferongames.com/categories/game-physics/)**.

In the previous article (http://web.archive.org/web/20181107181440/https://gafferongames.com/post/integration_basics/) we discussed how to integrate the equations of motion using a numerical integrator. Integration sounds complicated, but it's just a way to advance the your physics simulation forward by some small amount of time called "delta time" (or dt for short).

But how to choose this delta time value? This may seem like a trivial subject but in fact there are many different ways to do it, each with their own strengths and weaknesses - so read on!

Fixed delta time

The simplest way to step forward is with fixed delta time, like 1/60th of a second:

```
double t = 0.0;
double dt = 1.0 / 60.0;

while ( !quit )
{
    integrate( state, t, dt );
    render( state );
    t += dt;
}
```

then you already have the perfect solution for updating your physics simulation and you can stop reading this article.

But in the real world you may not know the display refresh rate ahead of time. VSYNC could be turned off, or you could be running on a slow computer which cannot update and render your frame fast enough to present it at 60fps.

In these cases your simulation will run faster or slower than you intended.

Variable delta time

Fixing this *seems* simple. Just measure how long the previous frame takes, then feed that value back in as the delta time for the next frame. This makes sense because of course, because if the computer is too slow to update at 60HZ and has to drop down to 30fps, you'll automatically pass in $^1\!/_{30}$ as delta time. Same thing for a display refresh rate of 75HZ instead of 60HZ or even the case where VSYNC is turned off on a fast computer:

```
    double t = 0.0;

    double currentTime = hires_time_in_seconds();

    while ( !quit )
    {
        double newTime = hires_time_in_seconds();
        double frameTime = newTime - currentTime;
        currentTime = newTime;

        integrate( state, t, frameTime );
        t += frameTime;

        render( state );
    }
```

But there is a huge problem with this approach which I will now explain. The problem is that the behavior of your physics simulation depends on the delta time you pass in. The effect could be subtle as your game having a slightly different "feel" depending on framerate or it could be as extreme as your spring simulation exploding to infinity, fast moving objects tunneling through walls and players falling through the floor!

One thing is for certain though and that is that it's utterly unrealistic to expect your simulation to correctly handle *any* delta time passed into it. To understand why, consider what would happen if you passed in 1/10th of a second as delta time? How about one second? 10 seconds? 100? Eventually you'll find a breaking point.

Semi-fixed timestep

It's much more realistic to say that your simulation is well behaved only if delta time is less than or equal to some maximum value. This is usually significantly easier in practice than attempting to make your simulation bulletproof at a wide range of delta time values.

```
double t = 0.0;
double dt = 1 / 60.0;

double currentTime = hires_time_in_seconds();

while ( !quit )
{
    double newTime = hires_time_in_seconds();
    double frameTime = newTime - currentTime;
    currentTime = newTime;

    while ( frameTime > 0.0 )
    {
        float deltaTime = min( frameTime, dt );
        integrate( state, t, deltaTime );
        frameTime -= deltaTime;
        t += deltaTime;
    }

    render( state );
}
```

The benefit of this approach is that we now have an upper bound on delta time. It's never larger than this value because if it is we subdivide the timestep. The disadvantage is that we're now taking multiple steps per-display update including one additional step to consume any the remainder of frame time not divisible by dt. This is no problem if you are render bound, but if your simulation is the most expensive part of your frame you could run into the so called "spiral of death".

What is the spiral of death? It's what happens when your physics simulation can't keep up with the steps it's asked to take. For example, if your simulation is told: "OK, please simulate X seconds worth of physics" and if it takes Y seconds of real time to do so where Y > X, then it doesn't take Einstein to realize that over time your simulation falls behind. It's called the spiral of death because being behind causes your update to simulate more steps to catch up, which causes you to fall further behind, which causes you to simulate more steps…

So how do we avoid this? In order to ensure a stable update I recommend leaving some headroom. You really need to ensure that it takes *significantly less* than X seconds of real time to update X seconds worth of physics simulation. If you can do this then your physics engine can "catch up" from any temporary spike by simulating more frames. Alternatively you can clamp at a maximum # of steps per-frame and the simulation will appear to slow down under heavy load. Arguably this is better than spiraling to death, especially if the heavy load is just a temporary spike.

Free the physics

Now let's take it one step further. What if you want exact reproducibility from one run to the next given the same inputs? This comes in handy when trying to network your physics simulation using deterministic lockstep, but it's also generally a nice thing to know that your simulation behaves exactly the same from one run to the next without any potential for different behavior depending on the render framerate.

most cases but it is not *exactly the same* due to to the limited precision of floating point arithmetic.

What we want then is the best of both worlds: a fixed delta time value for the simulation plus the ability to render at different framerates. These two things seem completely at odds, and they are - unless we can find a way to decouple the simulation and rendering framerates.

Here’s how to do it. Advance the physics simulation ahead in fixed dt time steps while also making sure that it keeps up with the timer values coming from the renderer so that the simulation advances at the correct rate. For example, if the display framerate is 50fps and the simulation runs at 100fps then we need to take two physics steps every display update. Easy.

What if the display framerate is 200fps? Well in this case it we need to take half a physics step each display update, but we can’t do that, we must advance with constant dt. So we take one physics step every two display updates.

Even trickier, what if the display framerate is 60fps, but we want our simulation to run at 100fps? There is no easy multiple. What if VSYNC is disabled and the display frame rate fluctuates from frame to frame?

If you head just exploded don’t worry, all that is needed to solve this is to change your point of view. Instead of thinking that you have a certain amount of frame time you must simulate before rendering, flip your viewpoint upside down and think of it like this: the renderer **produces time** and the simulation **consumes it** in discrete dt sized steps.

For example:

```
double t = 0.0;
const double dt = 0.01;

double currentTime = hires_time_in_seconds();
double accumulator = 0.0;

while ( !quit )
{
    double newTime = hires_time_in_seconds();
    double frameTime = newTime - currentTime;
    currentTime = newTime;

    accumulator += frameTime;

    while ( accumulator >= dt )
    {
        integrate( state, t, dt );
        accumulator -= dt;
        t += dt;
    }

    render( state );
}
```

Notice that unlike the semi-fixed timestep we only ever integrate with steps sized dt so it follows that in the common case we have some unsimulated time left over at the end of each frame. This left over time is passed on to the next frame via the accumulator variable and is not thrown away.

But what do to with this remaining time? It seems incorrect doesn't it?

To understand what is going on consider a situation where the display framerate is 60fps and the physics is running at 50fps. There is no nice multiple so the accumulator causes the simulation to alternate between mostly taking one and occasionally two physics steps per-frame when the remainders "accumulate" above dt.

Now consider that the majority of render frames will have some small remainder of frame time left in the accumulator that cannot be simulated because it is less than dt. This means we're displaying the state of the physics simulation at a time slightly different from the render time, causing a subtle but visually unpleasant stuttering of the physics simulation on the screen.

One solution to this problem is to interpolate between the previous and current physics state based on how much time is left in the accumulator:

```
    double t = 0.0;
    double dt = 0.01;

    double currentTime = hires_time_in_seconds();
    double accumulator = 0.0;

    State previous;
    State current;

    while ( !quit )
    {
        double newTime = time();
        double frameTime = newTime - currentTime;
        if ( frameTime > 0.25 )
            frameTime = 0.25;
        currentTime = newTime;

        accumulator += frameTime;

        while ( accumulator >= dt )
        {
            previousState = currentState;
            integrate( currentState, t, dt );
            t += dt;
            accumulator -= dt;
        }

        const double alpha = accumulator / dt;

        State state = currentState * alpha +
            previousState * ( 1.0 - alpha );

        render( state );
    }
```

This *looks* complicated but here is a simple way to think about it. Any remainder in the accumulator is effectively a measure of just how much more time is required before another whole physics step can be taken. For example, a remainder of dt/2 means that we are currently halfway between the current physics step and the next. A remainder of dt*0.1 means that the update is 1/10th of the way between the current and the next state.

linear interpolation between the two physics states to get the current state to render. This interpolation is easy to do for single values and for vector state values. You can even use it with full 3D rigid body dynamics if you store your orientation as a quaternion and use a spherical linear interpolation (slerp) to blend between the previous and current orientations.

NEXT ARTICLE: Physics in 3D (http://web.archive.org/web/20181107181440/https://gafferongames.com/post/physics_in_3d/)

● (http://web.archive.org/web/20181107181440/https://www.linkedin.com/in/glennfiedler/)

● (http://web.archive.org/web/20181107181440/https://twitter.com/gafferongames)

● (http://web.archive.org/web/20181107181440/https://github.com/gafferongames)