x
Search for: Search
November 21, 2016
Twitter Linkedin RSS E-mail
Gaffer on Games

Main menu

Skip to content

Fix Your Timestep!

Introduction

Hello, I'm Glenn Fiedler and welcome to the second article in my series on Game Physics.

In the previous article we discussed how to integrate the equations of motion using an RK4 integrator. Integration sounds complicated but really 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 a 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;
}
```

In many ways this code is ideal. If you are lucky enough to have your physics delta time match the display rate and you can ensure that your update loop takes less than one frame then you have the perfect solution for updating your physics simulation.

But in the real world you may not know the display refresh rate ahead of time, VSYNC could be turned off, or perhaps 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 the player falling through the floor!

One thing is for certain though and that is that it's utterly unrealistic to just 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. You can use mathematics to find this exact delta time value given your simulation (there is a whole field on it called numerical analysis, try researching "interval arithmetic" as well if you are keen), or you can arrive at it using a process of experimentation, or by simply tuning your simulation at some ideal framerate that you determine ahead of time. This is usually significantly easier in practice than attempting to make your simulation bulletproof at a wide range of delta time values.

With this knowledge at hand, here is a simple trick to ensure that you never pass in a delta time greater than the maximum value, while still running at the correct speed on different machines:

```
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 of course 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 problems including the so called "spiral of death".

What exactly is this spiral of death? It's what happens when your physics simulation cannot 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 ironically being behind causes your update to simulate more steps, which causes you to fall further behind, which makes you 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, assuming of course that the heavy load is just a temporary spike.

Free the physics

Now lets 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.

But you ask why is it necessary to have fully fixed delta time to do this? Surely the semi-fixed delta time with the small remainder step is "good enough"? And yes, you are right. It is good enough in most cases but it is not *exactly the same*. It takes only a basic understanding of floating point numbers to realize that (v*dt)* + (v*dt) is not necessarily equal to v2dt due to to the limited precision of floating point arithmetic, so it follows that in order to get exactly the same result (and I mean exact down to the floating point bits) it is necessary to use a fixed delta time value.

So what we want 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.

What if the display framerate is 200fps? Well in this case it would seem that we need to take half a physics step each display update, but we can't do that, we must advance with constant dt; So instead we must 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 here. Finally, 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 chunks.

Again:

The renderer produces time and the simulation consumes it in discrete dt sized chunks.

```
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 is important! This left over time is passed on to the next frame via the accumulator variable and is not thrown away.

The final touch

But what do to with this remaining time? It seems incorrect somehow 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 in general all render frames will have some small remainder of frame time left in the accumulator that cannot be simulated because it is less than dt. What this means is that we're displaying the state of the physics simulation at a time value slightly different from the render time. This causes a subtle but visually unpleasant stuttering of the physics simulation on the screen known as temporal aliasing.

One solution 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 pretty 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.

We can use this remainder value to get a blending factor between the previous and current physics state simply by dividing by dt. This gives an alpha value in the range [0,1] which is used to perform a 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.

Click here to download the source code for this article.

Next: Physics in 3D

If you enjoyed this article please consider making a small donation. **Donations encourage me to write more articles!**

288 comments on “Fix Your Timestep!”

1. *deniz*
   August 15, 2016 at 10:05 pm

Why do we let rendering be done at full speed? Isn't it just a waste of CPU if it's above monitor refresh rate? With the last loop you shared, the CPU usage just hits very high rates unless I limit rendering rate.

Reply

-  *Glenn Fiedler*
  August 17, 2016 at 12:48 am

  The example is written assuming that render() blocks on vsync.

  Reply

2. *Zach*
   August 2, 2016 at 12:57 pm

   Great article – even more impressive you're maintaining the comments 12 years later 🙂

   Assuming non-networked, is your biggest hit against variable-update the numerical instability in physics updates with too large/small dt? If so, what are your thoughts on doing variable-updates within a given range (ex. 12ms-20ms). Use an accumulator to stockpile dt upper bound.

   Sure, you lose predictability, but you don't have to fuss with fudging object transforms for rendering purposes – which may look weird for objects moving relatively fast with non-linear motion.

   Moreover, would you change anything about this article if you wrote it today? Has anything changed in 12 years?

   Reply

   - *Zach*
     August 2, 2016 at 1:01 pm

     Hm. The post didn't survive formatting. "Use an accumulator to stockpile dt upper bound." should read: Use accumlator to stockpile time less then lower bound, and run potential multiple updates if greater then upper bound.

     Reply

3. *Peter Keller*
   August 2, 2016 at 12:51 pm

   Thanks for the great article!

   I have a question about the final touch solution.

   When rendering there are two states P[0] and C[0] (where 0 represents the now update) and you interpolate between them to form B[0]–the state actually used to perform the render.

   But, suppose that just after rendering with B[0], you have N updates before rendering again. At the next render time P[N] and C[N] are interpolated to B[N] and the render happens.

   However, notice what occured:

   The last view the viewer saw was from B[0] and the newest view was from B[N]. However the interpolations from B[1] to B[N-1] were lost!

   Does that mean even the final touch solution will still jutter in the right circumstances?

I see that if the interpolation happened actually between B[0] and B[N] to form ((B[N] – B[0]) / 2) it is a bit 2nd order and the game rendering might be behind enough from the physics that gameplay suffers, but it would be indeed rendering the best it could from the last time the user saw something on the screen.

Thank you!

Reply

- *Peter Keller*
  August 3, 2016 at 8:06 pm

  In thinking about it. You could compute and store each entry in I[0] to I[N] and them perform spline interpolation between the states and choose the middle. That would get you very close to the appropriate rendering state for rendering and also provide a means to optimize jitter removal AND rendering an appropriate physics state.

  Reply

  - *Peter Keller*
    August 6, 2016 at 1:54 am

    After a long study. I retract my comment.

    Reply

-  *Michael Fiano*
  August 4, 2016 at 10:36 am

  This is a good question. I'm also interested in the answer to this.

  Reply

4. *Levent*
   July 13, 2016 at 9:57 am

   Thanks for this nice article.

   Even after many years of its initial publication; it is still an important reference and guide.

   Reply

5. *Ben*
   May 4, 2016 at 8:03 am

   Hi,
   thanks for the great article!

   Wouldn't "Free the physics" still possible result in different behavior on different machines due to the fp arithmetic?

   Reply

6.  *Bram Stolk*
   April 8, 2016 at 11:25 am

Great article Glenn. Still the seminal treatise on the web, after all these years.

I wrote down an alternative to this approach, if you can afford small physics steps, and don't mind speeding up / slowing down the game in some cases.

You can just pretend your display period is a multiple of 4⅙ ms:

http://gamasutra.com/blogs/BramStolk/20160408/269988/Fixing_your_time_step_the_easy_way_with_the_golden_48537_ms.php

Reply

7. *Mackie*
March 23, 2016 at 12:59 pm

This is a great article. How would you do a bullet time (0.25x) slow down of your simulation, and then come back to normal time (1x)? Doing all of this smoothly and robustly is the goal.

Is it as easy as integrate(state, t * 0.25, dt). I'm trying to understand how to do it And getting all confused.

Reply

-  *Glenn Fiedler*
March 24, 2016 at 8:22 am

That's easy. If you want to run at half speed, just take the timer value that feeds into the accumulator and divide it by two before adding to the accumulator. If you want to run at twice speed, double the timer value. If you want it smoothly, try a linear interpolation of time scale or a smoothing function on the time scale value.

cheers

Reply

- *Zarkow*
April 30, 2016 at 10:09 am

My approach was to always run Ticks at a fixed time-delta. If you want to run at 25% then run an actual tick every 4 attempted ticks.

I don't like the idea of having a variable delta given to the tick to calculate time passed for movement etc, instead if needed, run multiple ticks per rendered frames (see scenario that client machine is weak and can only run at 30 Hz, but our Tick is always 60 Hz, it means 2 Tick per rendered frame). This allows us to guarantee that the Tick are same when replayed across machines, you can even number ticks with Int for net-sync/replay/server-side-trace purposes.

Reply

  -  *Glenn Fiedler*
April 30, 2016 at 10:16 am

That's great but what happens when you need to run 34.7712371% speed? 😀

Reply

8. *Joel Schumacher*
November 19, 2015 at 6:09 pm

Hi Glenn,

I remember reading this article a few years ago and just now I start to think about the actual value of my update rate.
On the one hand one might as well choose a value very close or above the render rate, because most of the time games are not CPU bound anyways. On the other hand, I think, it is just not necessary since the impact of lower update rates on the game experience is mostly neglible (at least if your lowering it from 200% of the render rate to 80% or something). In my eyes the most reasonable approach would be to choose the update rate as just enough to achieve the required quality of the physics simulation (or just collision detection in smaller games). Am I missing something? What is the usual approach to this? What are the kinds of values you have seen yourself? Thanks a lot in advance already!

Reply

-  *Glenn Fiedler*
  November 20, 2015 at 7:34 am

  This approach also allows you to have a simulation rate that is lower than the render rate. This has often been used for RTS games, they simulate at 10HZ or 20HZ and render and whatever the display will do.

  They are less latency sensitive than most games though.

  Reply

  -  *Glenn Fiedler*
    November 20, 2015 at 7:35 am

    Regarding physics, generally you'll find that lower rates of physics updating induce tunneling and lower quality solver results. Most games tick physics at 60HZ for this reason.

    Reply

9. *Voy*
   October 11, 2015 at 4:41 pm

   Great article on the problem of a smooth game loop! I think there is something missing though – in the final source code.

   First: if the frameTime is smaller than dt then the accumulator may also be smaller than dt (for example when it is evoked for the first time) and no integration is done because the while(accumulator >= dt) condition is false. So the currentState is never updated.

   It is not a problem if dt is small enough but there again – do you really want to run the integration so often within one frame?

   Second: the interpolation you use to create the final state defines a point in time that is located between the points related to the previousState and the currentState. However the remainder left in the accumulator points forward in time (after the currentState)! So the value from the interpolation is in no way related to the point in time defined by the accumulator.

   I see two solutions:

   Solution 1: use the extrapolation (linear) instead of the interpolation. Bad solution, since the integration in the while(accumulator >= dt) part may still be omitted if frameTime is small.

   Solution 2 (better): force an integration for the point in time after the point that the accumulator refers to. The interpolation stays as it is written now.

   Am I right with this or have I missed something in the code?

   Voy

   Reply

   -  *Glenn Fiedler*
     October 11, 2015 at 6:54 pm

First: if the frameTime is smaller than dt then the accumulator may also be smaller than dt (for example when it is evoked for the first time) and no integration is done because the while(accumulator >= dt) condition is false. So the currentState is never updated.

^— Read the article again. That's what the interpolation step is for.

Reply

- *Nick*
  December 13, 2015 at 3:48 pm

  Pretty sure Voy is right here, I had the same concern reading over the loop.

  As an exaggerated example to make the point clear, let's say your physic timestep (dt) is 1 second, and your render loop hums along at 60FPS:

  You very first iteration of the loop, you have a frameTime / accumulator of 0. You interpolate between two identical states with an alpha of zero, and wind up just rendering the base state. That's fine.

  The second iteration, you have an accumulator of 0.016, which is still much smaller than dt. You still don't update anything, and you interpolate between two identical states with an alpha of 0.016. This is not what you wanted — you had hoped to move everything forward a little bit towards the next step.

  This goes on and on until your 60th frame, where you finally hit your (accumulator >= dt) condition. So you get to update currentState to 1 dt forward. And now your accululator is depleted, so alpha is 0. With alpha 0, you blend all the way back in favor of 'previousState', setting you right back where you started.

  This is basically just a big 'off by one' error. As Voy suggested, I think the loop needs to be edited to produce an update state 1 dt *past* what you have actually hit so far in time. Then you blend between the state you absolutely have hit and the 'future state' based on the value of the accumulator.

  Thanks for writing the article, it was very informative.

  Reply

  -  *Glenn Fiedler*
    December 14, 2015 at 3:36 pm

    It's not off by one. It's interpolation. If you want extrapolation, eg. predicting forward into the future you can do that too but don't call the technique in this article "off by one" because it's not.

    Reply

    - *Darren*
      September 26, 2016 at 1:30 pm

      I scratched my head for a long time reading over the solution thinking basically the same thing – we are taking the amount of time left before our next frame and using it to interpolate between the last two computed frames. Thus, if we are halfway to the next frame, we interpolate halfway between the last frame and the current frame. This is worse from a "realtime perspective" than simply using the current frame. I assumed to use this method for accuracy one would precompute the next frame as though we had a full time step remaining in the accumulator (even though we don't) then interpolate between the current frame and the hypothetical next-frame.

      Perhaps I am missing something obvious.

      Reply

      -  *Glenn Fiedler*

September 30, 2016 at 12:10 pm

Yes. That would be extrapolation not interpolation.

Reply

- *Chris*
  October 1, 2016 at 9:24 pm

  It sounds to me that, while you are being technically correct, all it is really saying is that interpolation produces slightly less desirable behavior.

  Reply

  -  *Glenn Fiedler*
    October 4, 2016 at 12:12 pm

    That depends. Do you consider extrapolating through walls desirable?

    Reply

10. *Matt*
    October 6, 2015 at 5:26 pm

    How much of this applies to the bullet physics library? According to the documentation, it interpolates for you and it keeps it's own internal clock. Is the accumulator even necessary?

    See the first section here: http://bulletphysics.org/mediawiki-1.5.8/index.php/Stepping_The_World

    Reply

    -  *Glenn Fiedler*
      October 6, 2015 at 9:03 pm

      I believe the bullet physics library implements the technique described in this article internally.

      Reply

11. *Paul*
    September 18, 2015 at 1:51 am

    Just a note of appreciation! Great article. I've implemented this timestep scheme in my hobby game and it works like a charm:

    http://digital-minds-blog.blogspot.co.at/2015/09/decoupling-physics.html

    With credit and link back to here of course!

    Thanks again.

    Reply

12. *George*
    August 29, 2015 at 1:54 pm

    Awesome job, I really like your blog. If you want you can check out my game which i've just released. It's using libGDX
    https://play.google.com/store/apps/details?id=com.blackfishgames.bouncywheel

Reply

13. *JOHN*
August 15, 2015 at 4:53 am

I implemented the interpolation approach at your final touch but I am getting jumps and jitter during gameplay when fps drops to 40 from 60.
I expect the jumpiness to be unusual and there must be a way to avoid it since fps is still larger than 30.

Is this normal? Is there a wayto avoid the jumpiness?

My whole indie game project is stuck because of thi

Reply

- *JOHN*
August 15, 2015 at 5:05 am

and also, after some time, my standing objects starts to juggle.

I uploaded a video of the juggling (link below). The juggling is present when interpolation is implemented. In the video, physical bodies of the balls are not moving but the corresponding sprites that are rendered are moving.

I believe this is due to floating point error.

Is there a way to avoid the juggling?

Reply

  - *JOHN*
August 15, 2015 at 5:24 am

forgot the video link : https://www.youtube.com/watch?v=nO1ApmH904Q&feature=youtu.be

Reply

    - *Paul*
September 20, 2015 at 12:08 am

Yeah that looks more like a rounding problem than a timestep problem.

I'm not sure if there is a standard accepted solution for this, but I'm thinking something like using a minimum "tolerance" value for your vector (I assume you are using vectors for movement) where you simply set the movement vector to zero if the magnitude is less than the tolerance value.

Good luck!

Reply

- *Paul*
September 18, 2015 at 1:50 am

Did you manage to fix your problem? I know how frustrating stuff like this can be!

It's not even an Indie project, just a hobby, but I've implemented the timestep with good results – maybe the non-psuedo code here will help you out?

http://digital-minds-blog.blogspot.co.at/2015/09/decoupling-physics.html

Cheers!

Reply

- *sean*
  March 7, 2016 at 4:10 pm

  in your code sample you put the update loop but not the interpolation code. How did you interpolate?

  Reply

  -  *Glenn Fiedler*
    March 7, 2016 at 4:45 pm

    I don't remember. I wrote this article 12 years ago 😀

    Reply

14. *Enes Battal*
    July 23, 2015 at 4:30 am

    I don't understand why you are interpolating between the current step and the previous step based on the accumulated time that is to be simulated.

    My understanding is that accumulated time should be used to guess the next incomplete step after the current step in the way after.

    Reply

    - *Enes Battal*
      July 23, 2015 at 4:32 am

      isn't this going back in time since the latest step is the current step?

      Reply

      -  *Glenn Fiedler*
        July 24, 2015 at 10:24 pm

        Of course it goes back in time it is interpolation. If you are predicting forward that would be extrapolation not interpolation.

        Reply

15. *wowsers*
    July 19, 2015 at 4:07 pm

    Would it be possible to instead of saving two states and interpolate between them, take the total time of the previous physics "step" and add the interpolation point to the current step? This would also add the possibility of slight graphical glitching, but also at gain he benefit of not delaying the program by one frame?

    Reply

-  *Glenn Fiedler*
  July 19, 2015 at 11:11 pm

  Yes. This is called extrapolation.

  Reply

  - *wowsers*
    July 20, 2015 at 7:52 am

    Is there a simple way to implement that? Interpolation seems to be the only thing covered throughly enough for me around the internet, do you have a simple adaption in code for it? It would be helpful, thanks in advance and to previous comment.

    Reply

    -  *Glenn Fiedler*
      July 20, 2015 at 9:15 pm

      All you need to do is to take the current position and the difference between that and the previous position and then "extrapolate" forward by adding that difference to the current position to get the extrapolated current position.

      The only tricky part is that you have to make sure you interpolate between previous + extrapolation and current + extrapolation guess. So you are interpolating with the accumulation factor between previous + previous_extrapolation and current + current extrapolation at all times.

      Everything else works exactly the same as described in the article.

      cheers

      Reply

16. *Libor Capak*
    March 27, 2015 at 10:13 am

    I've implemented these simple interpolation method. Sometimes my loop (while (accumulator >= dt) …) fall into two integration updates and next frame I get zero updates. So it's 2 updates, 0 updates, 2 updates, 0 updates, etc.. I'm not sure how to treat this unwanted behavior. Any advice please? thanks

    Reply

17. *SlowSnow*
    February 10, 2015 at 9:48 am

    Спасибо за статью,жаль нет перевода на русский.

    Reply

18. *Sergey*
    January 6, 2015 at 12:45 pm

    Очень полезная статья! Кроме того, у вас отличная способность правильно и понятно объяснять! Огромное спасибо!

    Reply

-  *Glenn Fiedler*
  January 6, 2015 at 6:32 pm

  Большое спасибо!

  Reply

19. *bilbo baggins*
    January 5, 2015 at 12:02 am

    I have a question in regards to interpolation. Say you have a ball that moves from one end of the screen and stops at the other end.

    Let's say that the previous x,y coordinates are 4,4 and the current coordinates are 5,5

    and the interpolation delta is 0.5. So the interpolated value is 4.5, 4.5

    So now the ball is at rest and is no longer animating and it is being displayed at 4.5, 4.5. How do you resolve the issue that the ball NEVER really comes to its actual resting place of 5, 5?

    Reply

    -  *Glenn Fiedler*
      January 5, 2015 at 9:22 am

      I don't understand. As soon as time moves past the time of the object coming to rest the interpolation will be between 5,5 and 5,5 -> 5,5.

      Reply

      - *BILBO BAGGINS*
        January 6, 2015 at 10:02 am

        Thanks you just solved my problem! lol!

        In my fixed time step tick function when the object came to rest I was NOT setting the previous and current position to 5,5 and 5,5 respectively. This is why in my example the ball remained at 4.5, 4.5.

        I had forgotten to set the previous and current positions to the final destination position when the ball came to rest in the tick function.

        Thanks so much for the help!

        Reply

20. *hrc*
    December 29, 2014 at 1:08 pm

    I don't understand the part about interpolating using the accumulator. If the point is that there is effectively unsimulated time, surely interpolating between the current and past states is actually going backwards in time? So it may get rid of temporal aliasing but it's going to be like a frame out of time?

    Reply

    -  *Glenn Fiedler*
      December 29, 2014 at 1:10 pm

You're actually delaying the simulation by one frame and then doing the interpolation to ensure smoothness. That is all. Otherwise you get jitter because the simulation rate and the render rate don't match — eg. 1,1,1,2,0,1,1,1,2,0, etc… That doesn't look good. The eye notices. The interpolation with accumulator fixes that, but costs you one frame of added latency. — cheers

Reply

-  *Glenn Fiedler*
  December 29, 2014 at 1:11 pm

  This is why the best possible situation is VSYNC with your game running at a fixed timestep matching that. That's not always possible though, but if it is possible, you should always do that.

  Reply

  - *hrc*
    December 29, 2014 at 1:25 pm

    Thanks for the super fast reply. I already felt that there is no absolute answer and all implementations have their own pros and cons. It's difficult to decide what to use.

    Reply

    -  *Glenn Fiedler*
      December 29, 2014 at 1:27 pm

      Agreed. Like most things there is no silver bullet here. For some applications adding one frame of latency is bad and it's better to run at native render rate (eg. FPS). For others, eg. deterministic lockstep it's super important that everybody runs at the same fixed timestep. You have to look at exactly you want to achieve, consider all the options and think! — cheers

      Reply

      -  *Glenn Fiedler*
        December 29, 2014 at 1:29 pm

        If in doubt, just lock to VSYNC on your computer. 99% sure it'll be 60, and that's a nice fixed timestep for development. Alternatively, you could determine what the refresh rate is and then just force fixed timestep for 1/display_framerate to run on a wider range of computers. — cheers

        Reply

        - *hrc*
          December 29, 2014 at 1:55 pm

          For now i'm going to go with the timestep matching the vsync hz, mainly because it's simpler and a good default. It would be easy to allow multiples of that timestep as an advanced option. My game isn't using a deterministic lockstep either.

          I thought that there's something nice about how if you have an unlocked fixed timestep your game will always behave exactly how you expected, but i'm wondering whether that is really so important above other things. I think i'm right in saying that as long as the timestep is stable increasing it can only get better results.

          Anyway you've been really helpful so thank you.

          Reply

-  *Glenn Fiedler*
  December 29, 2014 at 5:48 pm

  No worries! cheers

  Reply

21. *drandreasdr*
    December 14, 2014 at 12:58 am

    Thanks for the answer, but I'm still confused. How can it be that the code works even if there is no syncing to the display refresh rate? What is restricting the loop in the first example from running at a much higher number of iterations per second than the intended 60? (For instance, integrate() and render() might be so low-effort that the computer is able to run the loop at 6000 iterations per second, in which case the simulation will appear fast-forwarded by a factor of 100.)

    Reply

    -  *Glenn Fiedler*
      December 14, 2014 at 4:34 am

      The answer is contained in the article. Each time it runs through the main loop it accumulates time. Only when that accumulator enough time to perform one or more frame updates does the simulation run. This is how the game updates at 60fps even though it make go through the render loop (non-zero time btw…) at a much higher rate, in effect decoupling rendering and physics.

      Reply

      - *drandreasdr*
        December 14, 2014 at 12:38 pm

        I understand that this is the case for the last two examples shown, but I was referring to the first example, in which the while-loop only does three things: integrates, renders and advances the time variable. Is it true to say that a fast computer is free to run through this loop at a much much higher rate than 60 iterations per second? If not, where in the code is the iteration rate restricted?

        Reply

        -  *Glenn Fiedler*
          December 14, 2014 at 5:59 pm

          Yes a computer would run that loop as fast as it can.

          Reply

          - *drandreasdr*
            December 14, 2014 at 9:31 pm

            Alright, thanks for the answer!

            Reply

22. *drandreasdr*
    December 13, 2014 at 3:20 pm

    There seems to be some piece of information about what the render() function represents that is only implied in this article. For

instance, what is it that is preventing the loop in the first piece of code from running as fast as the computer possibly can manage, rather than at the intended 60 iterations per second?

Does render() in fact contain not only instructions for rendering, but also an instruction to “sleep” until the next monitor refresh cycle begins, before returning?

Reply

-  *Glenn Fiedler*
  December 13, 2014 at 11:37 pm

  The render does the rendering. It is implied that the display refresh rate is up to you — you can choose to sync to retrace or not. The code works in both cases.

  Reply

23.  *Tom*
November 19, 2014 at 5:03 pm

Hi Glenn,

I’ve attempted to implement the most complex version of your timestep (i.e. the bottom one!), but i have a question regarding it. Should the timestep be a constant value, as my implementation has it counting down ever so slightly, i.e. it begins at around the 12th decimal place and as the program runs it gradually effects the preceding decimal places! Or, it there a bug in my implementation? I should note that i have no state manipulation currently, just trying to get an accurate delta time for the time being! Thanks

Reply

-  *Glenn Fiedler*
  November 20, 2014 at 6:08 am

  I have no idea what is going on there. If you are using the most complicated version then you should be stepping forward with constant delta time, subtracting this constant delta time from the time accumulator (where real time passing is added), and interpolating between before render. Please refer to the sample code.

  Reply

24. *HKGames*
September 28, 2014 at 6:08 am

So, the line:
State state = currentState * alpha + previousState * (1.0 – alpha);
render(state);
renders ahead the left over interpolated delta?
How do you do this easily with the whole physics world and many objects?
Do you have to clone your world, then step that to be rendered? That seems like a lot of overhead every frame.

Reply

-  *Glenn Fiedler*
  September 29, 2014 at 1:28 am

  Yes, you clone your entire world state, eg. visible quantities such as position/orientation, and you have to specially handle attached objects hierarchically.

  A much easier solution is to just always lock to 60HZ or so, if you can get away with that, it’s the best and simplest option.

Reply

25. *Irlan*
September 15, 2014 at 7:33 pm

Congrats! One thing I think would make it even more clear is say: “In any situation, even if the real interval is bigger or less than the fixed value, it keeps being incremented and considered in the next frame”. Anyway… congrats one more time!

Reply

26. *Scripts*
August 27, 2014 at 12:04 am

And for that, Glenn, you are my #1 Hero.

Reply

-  *Glenn Fiedler*
August 27, 2014 at 3:31 am

  Thanks!

  Reply

27. *MrMan*
July 8, 2014 at 6:52 am

Can someone explain to me; what’s wrong with just using:
world.step(Gdx.graphics.getDeltaTime(), 6, 2);

Reply

-  *Glenn Fiedler*
July 8, 2014 at 7:37 am

  Yes

  Reply

- *Anonymous*
September 27, 2014 at 9:54 am

  You should never use that because of two reasons: 1. Your simulation may jump to light speed or grind to a halt if your FPS fluctuates, potentially causing tunneling and other side effects and 2. It will run at different speeds on different devices, making game play lighting fast or painfully slow.

  Reply

28. *Anonymous*
February 9, 2014 at 12:04 pm

Good post. Thank you. In

Reply

29. *Thomas*
November 21, 2013 at 4:32 am

Cool article! Interestingly, I encountered the issue of the physics exploding when the update time wasn't fast enough (things moving through boundaries) but didn't question the update logic too much.

I have a question, though. Can you think of any ramifications for having separate time deltas for different components? For example, if you have fast moving objects, you could have them update at say 100 updates per second, but for slower moving objects or more static objects you could have only say 50 or 30 UPS. This way you can have a fixed time step (and thus deterministic calculations), without interpolation, but minimize temporal aliasing since all highly dynamic objects are frequently updated so your renderer more closely matches your simulation. Of course, you'd have to know which objects are high activity and what a good enough UPS for them is…(maybe there is a good way to figure this out dynamically?)

So what are your thoughts?

Reply


Glenn Fiedler
November 21, 2013 at 3:58 pm

There are ramifications. If the objects can interact then it will affect that, but it can often be OK.

For example, in 2002 I worked on Freedom Force and that had high quality physics for important objects (60) low quality for others (20) as an optimization, and the AI ticks such as decision making and A* at much lower rates (eg: make decisions 4 times a second)

But you wouldn't want to do different update rates for objects that needed to form a stable stack.

Reply

30. *Geeves*
October 25, 2013 at 2:04 pm

First off a big thank you for providing these concise explanations. I've referenced your site countless times in conversations since I first discovered it years ago.

I've implemented interpolation for my physics engine, but I was wondering if I could circumvent the interpolation step for my particle emitter. Looping through hundreds of particles to interpolate seems unnecessary because a variable timestep for particles is efficient.

I was thinking about adding an additional update call before the interpolation, and passing in accumulator, which updates my particles .

Reply


Glenn Fiedler
October 25, 2013 at 4:19 pm

Yes that is a good plan. For particles and other visual effects it is typical to not fix time step for those, interpolation and the double copy of state is a waste

Reply

31. 
xISTIx
October 18, 2013 at 1:53 pm

Great article! Thanks.
One thing I do not understand. In the while loop you step the physics engine until there is no whole step to catch up with the current render time. In other words if the accumulator is still positive after the while loop it means your physics simulation is still behind the render time. And then you interpolate between two past times. In my understanding you should go one step further where your simulation time is already in the future and the former simulation time is in the past, so you can interpolate to the current render time.

Reply

32. *Texel*
September 14, 2013 at 12:40 pm

Many thanks for the great article.
I just implemented the loop like you suggested and it works seemingly flawless.
However, I had to switch the weighting for the lerp to make it work:

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

instead of:

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

The bigger the remainder of “accumulator” (and therefore alpha) is the bigger the weighting of the previousState has to be. 😉

Reply

33. *Zack*
August 26, 2013 at 11:08 pm

Hm. When you put it like that you’re right, it really doesn’t seem much.

Thanks very much, Glenn. 🙂

Reply

34. *Zack*
August 24, 2013 at 12:14 am

I’m just getting started with programming and I’m looking this up to help with writing my first game loop so I may be wrong, but with the interpolation at the end does this mean every changing value of every object has to have a current and old value stored? This seems like it could be a huge amount of extra data and also a lot more code as at each step every value that changes has to be first ‘backed up’ for the old value. Is this right?
If this is the case do you make a custom data type or custom object to store these temporally linked pairs of values and back them up or is this a bad idea?

Thanks for the article and sorry it’s a lot of questions, this has been a very informative read for me and obviously it’s left me thinking about things I’ve not had to think about before. Haha.

Cheers,
Zack

Reply

-  *Glenn Fiedler*
August 24, 2013 at 3:46 pm

  Yes you have to store current and previous value per-object. It’s not a huge amount of data. It’s exactly double. eg. vec3f oldPosition; vec3f newPosition;

Reply

- *Zack*
  August 24, 2013 at 4:30 pm

  I guess, doubling does seem kind of huge to me though. 😛 I guess it depends a fair bit on the size of the project and capabilities of the target platform though.

  Looks like you just manually take care of all the backing up of variables then? Well it doesn't look to complicated so I think I will have a go at implementing my own version soon. 🙂

  Thank you for replying so fast and clearing all that up for me (especially as I've now seen how long ago you wrote this! :O) and writing such a great article in the first place. 🙂

  Cheers,
  Zack

  Reply

  -  *Glenn Fiedler*
    August 25, 2013 at 2:25 am

    It's not huge, unless you have hundreds of thousands of particles. Each vector takes 12 bytes, or 16 bytes if you are SIMD, so in a typical case with a few hundred objects you increase from 16*100 = ~1.6k -> 2*16*100 = 3.2k. It's really not a big deal.

    If you want to keep it simple just lock to display refresh and assume constant dt = 1/60.

    Reply

    -  *Glenn Fiedler*
      August 25, 2013 at 2:26 am

      Also I think you are confused, you don't need a duplicate of all objects variables, only immediately displayed ones, eg. position/orientation need this duplication. Basically, you just keep track of the previous and current transform for the object and that's all. It's easy.

      Reply

35.  *starwed*
    August 16, 2013 at 6:00 pm

    Curious about one choice with the semi-fixed approach — why not divide the frameTime evenly amongst the steps?

    Reply

    -  *Glenn Fiedler*
      August 16, 2013 at 6:58 pm

      That's another option. The idea with this one however is that you'll never have a timestep *larger* than your ideal dt. If you don't care about that, then your alternate idea is fine.

      Reply

-  *starwed*
  August 16, 2013 at 7:44 pm

  Sure, I meant choosing the minimum number of steps that would let you have a dt below that max value. Something like

  steps = ceiling(frameTime/maxdt);
  dt = frameTime/steps;

  That way your median dt should be smaller. Naively I'd think that would produce more stable physics, but I'm not very familiar with this sort of thing!

  (Oh, and thanks again for the great article!)

  Reply

  -  *Glenn Fiedler*
    August 17, 2013 at 1:09 am

    Sure that's fine too. Really, you can slice and dice it however you want.

    Reply

36. *Paul Kankiewicz*
    July 30, 2013 at 10:09 pm

    Thanks for the help! I appreciate it!

    Reply

37. *Ralph*
    June 24, 2013 at 12:52 pm

    Hey Glenn, thank you for this awesome article!

    But I'm curious if the physics and the render would desync in case 'frametime > 0.25' because currentTime is increased by 'newTime-currentTime' instead of frametime. There is no way for the physics to ever catchup again, is there? What do you think about:

    if (frameTime > 0.25) {
    frameTime = 0.25; // note: max frame time to avoid spiral of death
    currentTime += frameTime;
    } else {
    currentTime = newTime;
    }

    And I'm also wondering if it really limits the frametime to 0.25 (4fps). The time needed for one loop is dominated by '0.25/dt * time_needed_to_process_integrate_function' what can still be greater than 0.25

    Reply

    -  *Glenn Fiedler*
      June 26, 2013 at 3:54 pm

      Yes the time would desync. That's the point, the computer cannot keep up with real time so time slows down.

      Reply

38. *OGLBeginner*
May 22, 2013 at 8:34 pm

I have fixed the Problem now 😉 The Problem was the rasterization on .5 float-positions
in combination with using sub-textures in openGL. Adding one extra pixel-Line arround each sub-texture fixes the flickering problem.
Now my 2D-Tile-engine (With interpolated cam) scrolls NEARLY perfectly smooth. But now I have a last problem. I render on ca. 300 FPS, but my Refreshrate is only 60 HZ. So only If I turn vsync on, then absolutly no flickering is there.
Is it a good idia to combine vsync with fixed timesteps? Maybe 50 Logic Ticks and 60 Renderings per Second? I thought the cool thing on interpolation is the fact, that u can render unbraked on full speed? 😕

Reply

-  *Glenn Fiedler*
  May 22, 2013 at 8:46 pm

  If you do vsync and you know the refresh rate (typically 60) you can just cut through all the crap and use fixed time steps with dt=1/ 60

  Reply

39. *Anonymous*
May 16, 2013 at 6:54 am

Yes, I render already on float coordinates :(. The root problem is, that one target-pixel could change beetween the calcualtion of two equal positons ? 😕 Or maybe I have an other problem?
Maybe I could write a little Test-Program, which reproduces this errors and post it here?

Reply

-  *Glenn Fiedler*
  May 16, 2013 at 3:58 pm

  I really have no idea

  Reply

40. *OGLBeginner*
May 16, 2013 at 5:30 am

U mean in addition to the linear interpolation or as a replacement? OpenGL always uses floats
to draw the sprites (Quad-Vertices from two triangles) and I have never seen, that someone
uses there interpolation for the drawing routine. Do u have a hint for me ? 🙂

Reply

-  *Glenn Fiedler*
  May 16, 2013 at 5:43 am

  My hint is just draw the sprites at float coordinates without clamp to int

  Reply

41. *OGLBeginner*
May 15, 2013 at 8:51 pm

Hi,
I have a big problem using the Frame-Interpolation with the Alpha-Values.
I am drawing a tile grid all over the screen. If I scroll the tiles, sometimes I can see
some really short flickering. After some tests, I decided to cast my tile-positions (float) to full
integers before drawing them (OpenGL) and let the tiles stay on one fix positon. So actual positions (float) and previous positions (float) were always exact the same, only the alpha-values (float) changes in dependence from the delta-time. The flickering was still there! So I think, sometimes some tiles changes their position in the distance of one pixel, If their position is near the next pixel. My int-cast should always be the same, cause it "cuts" the float to an int! So there must be an full integer overflow in dependence of the Delta-Time?

Maybe a Scenario like this:
ActPos = 1,9999999 (Castet to int = 1)
Prevpos = 1,9999999
AlphaValue: 1,9999999 * 0,9999999 + 1,9999999 * 0,0000001 ~~ 2,0 (Castet to int = 2!)

Does anyone have some trick for me….to solve the problem of the bad flickering 😕
Or maybe it is a complete other problem?

Reply

-  *Glenn Fiedler*
May 15, 2013 at 9:06 pm

Generally you want to render your sprites at float positions with bilinear interpolation

Reply

42. *Mick*
May 10, 2013 at 8:10 am

Hi Glenn,

I've been reading through your RK4 implementation and thinking of adding it as an option for more accuracy over my current use of velocity-corrected Verlet. This of course had me looking at your time-step implementation. As far as i see the 't' value continually increments. Is this correct? If so, not that it would come up under most circumstances, is there a solution once it reaches max double?

Reply

-  *Glenn Fiedler*
May 10, 2013 at 6:23 pm

I would recommend against using floating point values for time, but in reality a double value should be sufficient for any reasonable game session length. cheers

Reply

  - *Mick*
May 16, 2013 at 3:51 am

So is my understanding correct that you're using 't' effectively as the life of each object? Your example shows one object, but if there were objects being dynamically added / removed they'd each need their own 't' so that they all updated on their own timestep.

I.e. with a fixed physics delta t = 0.1s
create one object at realtime = 0s, then another at realtime = 0.05s.
By keeping their own ‘t’ value this would ensure that the second object doesn’t update it’s physics until realtime = 0.15s.

Reply

-  *Glenn Fiedler*
  May 16, 2013 at 4:05 am

  t is the amount of time elapsed in the game simulation

  Reply

  - *Mick*
    May 16, 2013 at 6:06 am

    If ‘t’ is the overall time for the simulation then, in the example i gave, both objects would be updated at overall time of 0.1s. Wouldn’t this mean that the second object is being updated before it should be?

    Reply

43.  *Max Li*
March 31, 2013 at 11:02 pm

Ah okay, so i would have to put in multiple asteroids, but just add them to the state struct? Maybe an array world work there.

Reply

44.  *Max Li*
March 31, 2013 at 5:03 pm

Hi Glenn,

First off, thanks for the awesome article, really useful, although I do have a few questions.

1. When I’m using this, the actual alpha for say all my objects in one frame should be the same, so they get the same amount of physics blending correct?

2. Also, what exactly is the part about render(state); supposed to do, what is the render function?

3. What is a State? I understand it’s from the struct, where it is a position in terms of x and y. But say I have a game like asteroids, do I have hundreds of states, or what?

4. Is the accumulator and dt universal, so I only need it once per game loop?

Reply

-  *Glenn Fiedler*
  March 31, 2013 at 6:22 pm

  Yes the alpha is the same for all objects

  State is the position and orientation for all objects in your game

  Render draws your game on the screen.

  Hope this helps, cheers

Reply

-  *Max Li*
  March 31, 2013 at 6:42 pm

  It does for the most part, although can you be more clear on what state is? For example if I have tons of asteroids, do I need a different state created for each of them? and can you answer #4? Thanks again!

  Reply

  -  *Glenn Fiedler*
    March 31, 2013 at 9:06 pm

    State is your whole scene, do yes inside state there would a position and orientation for each asteroid

    Reply

45. *Suminsky*
    March 30, 2013 at 10:01 pm

    Hi,
    In your opinion Glenn, what approach would you suggest for a 2D platformer, where, IMO, responsiveness and input latency are core factors to take into account? Not full physics based, but hard coded on the characters movement (think like genesis sonic and snes super mario world).
    Id like to know your opinion about both game loop and integration methods considering that kind of game.

    I also noticed you mentioning an article about input latency somewhere… >.< do it!
    Thanks a lot for all precious information.

    Reply

46.  *Tristan Matthews*
    March 24, 2013 at 4:30 pm

    Great article! I've written a GNU/Linux version of the header file which uses freeglut, if anyone's curious:
    https://gist.github.com/tmatth/5232538

    It just requires adding #include "Linux.h" in Timestep.cpp and linking against freeglut.

    Reply

    -  *Glenn Fiedler*
      March 24, 2013 at 4:58 pm

      Thanks!

      Reply

    - *Anonymous*
      June 10, 2013 at 9:47 pm

      Tristan, your code works! Thanks 🙂

      Reply

- *Anonymous*
  June 11, 2013 at 1:44 pm

  Glad to hear it, patches welcome.

  Reply

47.  *Elliot Winkler*
    February 20, 2013 at 8:01 am

    Not sure if people are going to see this with so many comments, but it's worth noting that:

    State state = previousState * (1.0 – alpha) + currentState * alpha
    [or State state = currentState * alpha + previousState * (1.0 – alpha) as you have it in the code in this post]

    is the same as saying:

    State state = previousState + alpha * (currentState – previousState)

    which is how you might intuitively think of interpolating. To take position as an example:

    state.position = a.position * (1 – alpha) + b.position * alpha

    is the same as saying:

    state.position = a.position + alpha * (b.position – a.position)

    In English this is "advance position A toward position B by a certain percentage 'alpha'". (well, except alpha isn't a percentage, it's a decimal, but anyway)

    I assume you choose the shorter way, though, because there's slightly less vector math involved.

    Reply

48.  *Silver Moon*
    January 3, 2013 at 7:13 am

    The accumulator approach is exactly what I was looking for, to make the physics simulation speed free from the frame rate. And I have used it in my game here

    http://thegamehill.com/fruitfrenzy/

    Reply

49. *Cory*
    December 28, 2012 at 7:27 am

    This may seem like a very poor question, but what exactly does your "hires" stand for in your "hires_time_in_seconds" function?

    Reply

    -  *Glenn Fiedler*
      December 29, 2012 at 12:55 am

      High resolution, eg. nanoseconds or better

Reply

50. *Netrick*
December 25, 2012 at 1:24 pm

I have a question to you Glenn. Is it better to check input (mouse, keyboard and packets) in fixed step loop or in render loop?

I mean #1:
while (true) //main loop
{
while(fixed step loop) //ie 50/100 times a second
{
check input
do logic
}
render
}

or rather #2:

while (true) //main loop
{
check input
while(fixed step loop) //ie 50/100 times a second
{
do logic
}
render
}

I think the approach #1 is better especially for lower framerates (you check input whenever you need for logic, and in case #2 @5fps you could press a key for a short moment and it could be processed 20 times by logic loop).

Could you advice me what's better?

Thanks

Reply

51. *Tsanas*
December 15, 2012 at 9:03 pm

I cannot seem to understand your last bit of code about interpolating.
'Space' left a comment about the same thing, saying:

"To do the interpolation for the correct position, you should use the current position and the NEXT position, meaning you would have to always calculate the next step, a frame before you actually render it."

Then Glenn replied with:

"yes thats exactly what i'm doing, interpolating between the previous frame and the current – effectively adding one frame of latency"

Which I still dont get.. Why interpolate between current and previous, if you are looking for a "future" frame?

If we take a look at the code and let 'State' represent a number which is incrementing.
Lets say that after the simulations, we are left with an alpha of 0.75, the current State is a number of 10 and the previous State is a number of 5.

The way that I understand it, we would want a number higher than 10, but when we interpolate between 5 and 10, with 0.75, we

would get 8,75, which is "older" than the current State.

What have I misunderstood?

Reply

- *Glenn Fiedler*
  December 16, 2012 at 5:20 pm

  But I'm not looking for a future frame. I'm interpolating between two frames.

  If you want a future frame then you would be extrapolating.

  That's fine, and you can choose to do that – guess where it will be in the future – if you want, but that's not what I'm doing

  I'm interpolating between the current and the previous state, and yes in the common case the interpolated state will be "behind" the current

  cheers

  Reply

  

  - *Elliot Winkler*
    February 20, 2013 at 8:17 am

    I think what's a little confusing is – in a way you are calculating a future frame, as the frame will not get fully rendered until alpha reaches 1 (at which point the future frame becomes the current and we calculate another frame). I guess I don't really get what the difference between interpolation and extrapolation is, the way you've defined them in code, it just seems like difference in semantics to me ("interpolation" is spelled current/previous, "extrapolation" is spelled next/current). Unless the difference has to do with responding to collision detection. I suspect it might, but can you expound on this a little bit? (or point me to some resources)

    Reply

    

    - *Elliot Winkler*
      February 20, 2013 at 8:26 am

      Perhaps a better question is, how would you change your example above to implement extrapolation rather than interpolation. I'm just curious so I know what to watch out for.

      Reply

      

      - *Glenn Fiedler*
        February 20, 2013 at 5:11 pm

        Interpolation is between two known values. Extrapolation is predicting past known values with a guess

        Reply

52.  *Molo Xiao*
    November 15, 2012 at 9:34 am

    It's great for me! I'm starting to do some cross-platform game, its help me. Thanks…

    Reply

53. *Jonathan Palencia*
November 9, 2012 at 5:19 pm

Hello! I just wanted to congratulate you for such a great article. It's a very important subject to consider when writing games and interestingly enough not many people write about it.

Reply

54. *Subu*
September 6, 2012 at 5:48 pm

I see that my explanation above has been mangled by wordpress – my full reasoning is at http://pastebin.com/bzbn2mjW. Feel free to mail me for any discussion / answers.

Thanks, Subu

Reply

55. *Subu*
September 6, 2012 at 5:43 pm

I know this is late – but great article. It's really a wonder how an article written 6 or so years back is still valid and correct. Math never gets old.

Anyway – I had forgotten what linear interpolation is so the last calculation indeed threw me for a loop. en.wikipedia.org/wiki/Linear_interpolation and http://en.wikipedia.org/wiki/Extrapolation#Linear_extrapolation helped me fix that – It would be helpful indeed if you can add a note pointing to this for people who are lagging on their math. They can probably derive the equation and convince themselves that what you are doing is correct.

Question here though: do correct me if I am wrong. I see that you have a previousState, currentState and a delta that has value in the future that you have not computed into your equation. This means two things to me:

1. you are extrapolating, not interpolating, since the point lies outside your known points. This is unless you calculate one frame extra all the time – which I don't see happening here.
2. Since you have a while (accumulator >= dt), you will infact move the data one step behind to previousState in case accumulator is an exact multiple, because of your equation.

is #2 intentional? See below for my reasoning:

Regarding #1 above, from http://en.wikipedia.org/wiki/Extrapolation#Linear_extrapolation

Equation for linear Extrapolation (and interpolation if Xk-1 < X* = 0) // might need to do with with FP tolerance.
{
previousState = currentState;
integrate(currentState, t, dt);
t += dt;
accumulator -= dt;
}

//at this point, pS contains this step. cS contains next step. accumulator contains (delta – dt) = ((X – Xk-1) – (Xk – Xk-1)) = (X – Xk). Remember, we want (X – Xk-1) in numerator to interpolate between pS and cS. So add dt to this value. It will make it (X – Xk + (Xk – Xk-1)) = (X – Xk-1).

accumulator += dt; //Make accumulator contain only delta. This is also needed for future iterations to be correct.

const double alpha = accumulator / dt;
State state = currentState*alpha + previousState * (1.0 – alpha);

…

What am I missing here?

Reply

56. *netrick*
September 3, 2012 at 7:00 pm

First, thanks for amazing article Glenn!

I have one question because there is one thing that I don't understand. Let's say, we have the fixed step physics simulation being done 25 times per second. And my question is about interpolation – how can I deal with getting into walls etc (basically – physics is calculated only 25 times per seconds while render with movement interpolation gets done 60 times per second). So not only you can get into the wall, but later when it's time to calculate physics you will be teleported some distance behind, which is for sure noticeable on high fps, 120 hz screen etc. How to deal with it? Because as I see calculating physics for the second time in loop at the moment of interpolation with variable delta time is just stupid as it fucks up the whole idea (no fixed step).
I hope that you understand what I mean because I'm not native english speaker – if you don't understand fully please tell me it and I will try to tell it another way!

Thanks in advance!

Reply

57. *grom*
July 27, 2012 at 6:15 am

I was wondering if you can weigh in on, http://gamedev.stackexchange.com/questions/33074/interpolate-and-collisions

I am taking it the solution to that is to run the physics at a high rate. After looking at your articles again I see you using dt = 0.01 so 100 ticks per second. That way avoiding the problem that I encountering.

Reply

58.  *Chris Latham*
July 20, 2012 at 3:15 pm

Hi Glenn,
I have implemented the fixed timestep system you have described in this article, using the RK4 integrator you described previously. For the most part it seems to be working nicely, but I've run into a small issue. My player movement works on a 'target speed' system, so if I want to walk left then the target speed on the X axis is, for example, -250. I have tried to implement the acceleration() function something like this:
return (target_v – current_v) * acceleration_constant;
This doesn't seem to be framerate independent, however. When I limit my game to run at 20 frames per second to test it, the current_v value never manages to reach the value of target_v. When I allow the simulation to run as fast as possible, it does.
How can I implement this behaviour in a framerate independent manner?

Thanks in advance.

Reply

59. *K. Otto*
July 16, 2012 at 5:50 pm

Yes you are right. And the good thing is, that if I use collision-detection and small deltas, I cannot run in trouble (Skipping some walls etc.). I think this is the best solution at the moment for me 🙂 Thanx so much for this article! Maybe you want to write some game-programming-book some day?!

Reply

60. *K. Otto*
July 15, 2012 at 8:29 pm

This article is great! I really want to use it for my game (Jump'n Run), but I cannot figure out how to implement it correct to be able to handle alle my objects with this game-loop. I have a player-object, many enemy-objects and some camera-positions (tile-engine) and much more.

Should I save for all of the objects the states and the previous states and also do the interpolate-calculations for all of them? Maybe this is some overkill? Maybe I need something like a state-container with all positions in there? I looked into the source code of some jump'n runs (Hurrican for example) ..it is super smooth..but no one uses fixed time steps. For fixed time steps I must do a complete rework of my game-engine…is it that worth or only from interest when dealing with more complex physics?

Reply

-  *Glenn Fiedler*
July 16, 2012 at 2:05 am

it's not that hard, but if you don't feel that you need fixed timesteps why not just go with the semi-fixed timestep instead.

Reply

61. *Malcolm Crum*
July 8, 2012 at 5:09 pm

Thanks for the great explanation. I've been trying to figure this stuff out for a while.

To help myself understand it better I implemented this algorithm in Python, using Pygame:
http://www.malcolmcrum.com/wp/python-implementation-of-rk4-with-variable-timestep/94/

Reply

62.  *Dirley Rodrigues*
June 29, 2012 at 1:24 am

Man, these articles are simply amazing. Thank you very much!

Reply

63. *Martin*
May 29, 2012 at 11:15 pm

I have a question about the final touch example, when you 'blend the states'.
How do you solve the issue when a game character Dies, or anything is removed from one state to the next? How can you blend between the 2 states of an object, where the object doesnt exist in the next state?

Reply

-  *Glenn Fiedler*
May 30, 2012 at 1:38 am

Simple answer: you don't!

Reply

64.  *Lars Butler*
[May 24, 2012 at 6:53 pm]

By the way: In the last block of code, you have a couple of minor typo errors.

Your `State` variables are called `previous` and `current`. Below that (inside the while loop), they are used as `previousState` and `currentState`.

[Reply]

-  *Glenn Fiedler*
[May 25, 2012 at 3:04 am]

I'm just lucky that the article doesn't need to compile.

[Reply]

65. *Anonymous*
[May 20, 2012 at 7:26 pm]

I'm seeing a jitter effect which is most noticeable during small oscillations of a spring.

It's only noticeable during subtle movements, but I'm wondering if that's something that you've dealt with?

[Reply]

-  *Glenn Fiedler*
[May 21, 2012 at 6:26 am]

That jitter is probably the particle render aliasing to pixel resolution.

[Reply]

66. *Robinson*
[March 30, 2012 at 10:22 pm]

Fantastic. Just what I was looking for. My frame-rate never looked so level :-).

Thanks.

[Reply]

67. *Imran Shafiq*
[February 14, 2012 at 3:32 am]

I currently use variable timestep in my WP7 2D game "Air Soccer Tour", and havent seen any issues besides the fact that if a frame takes a long time (physics+rendering+AI) the next frame naturally shows a jump in positions of objects. So I have reduced the frequency of AI updates (not computing AI every frame – but physics and render happens every frame) and the game stays between 50FPS to 60FPS and the results are satisfactory.
But maybe thats because I have very less objects in my game for physics simulation. i.e. 10 dynamic objects with continuous physics (TOI stuff) and 10 static objects. and the game coordinates are 6 meter by 10 meter. small world.
I thought Fixed timestep was a bad idea cause since the game will run fast on good hardware and slow on bad hardware. But you discussion on floating type precision, fixed physics timestep and accumulator logic makes sense.
I do have a question on networked physics. In my game there is no authoritative server, its a casual game so not worried about cheating :). I did a very basic implementation where two phones/players have their own physics + rendering happening and the server

is only relaying messages like player1 took this action/moved or player 2 moved. Now obviously the issue is that the physics simulation needs to be absolutely synchronized otherwise the two phones show different world state. And with latency of 100 ms to 200 ms over the internet, the physics go off on both ends with different results. So the next thing I did was use teleportation in the sense that when player 1 moves, it not sends its "action" info but also the whole world state. player 2 receives it and all objects teleport to the received positions before the player 1 action is executed. This of course looks completely un-acceptable to the players 🙂 So my next try is to somehow synchronize the simulation time i.e. instead of just sending the player moved message (which happens every 2-4 seconds I might add and its sort of turn based game) I was thinking the player who is active at the moment (meaning he can take his turn) will fire off simulation time messages to the inactive player (who is waiting on the turn) the inactive player will try to slow down the simulation step time to stay in sync with the time.. so when finally the player moved message arrives, the teleportation wont loook to bad.. but now the issue is that the first player will have to double slow his simulation and it wil become a mess soon 🙂 So, any ideas on a non-authoritative server based two player game?

Reply

-  *Glenn Fiedler*
  February 14, 2012 at 5:09 am

  Obviously if you have continuous collision the need for fixed timestep to avoid tunneling is greatly reduced or eliminated entirely.

  Reply

  -  *Glenn Fiedler*
    February 14, 2012 at 5:11 am

    And yes, your approach of sending state seems good. You can either run the sim on both sides and have each player broadcast the state for the object they control, or attempt to interpolate between the two last samples received from the other machine, and run the simulation only for "real" physics objects. I recommend the first approach, but first try it without any fancy time synchronization. Just apply the most recent state received over the network and do a visual smoothing (eg. reduce linear/angular error over time). You may find this is good enough. cheers

    Reply

    - *Imran Shafiq*
      February 20, 2012 at 12:05 pm

      I tried a few things, but still have issues:

      1. I have fixed simulation timestep (20 milliseconds) and a simulation step counter on both sides.
      2. Every 300 milliseconds the "Active team" is sending positions/velocities for all objects alongwith the currentSimulationStep number e.g. 8. The "Inactive team" receives the state message after lets say 400 milliseconds when he is already at simulation step 12. I apply the received state positions/velocities, set simulation step at receving end to 8 and then re-run simulation steps in a loop to to get back to step 12 (so basically adjusting and catching up).
      3. When the active team finally takes its turn within 4 seconds, (lets say at step 16) I dont apply that move right away I schedule it on the local system 20 simulation steps in future (i.e step 36) and send that info right away to the "inactive team". This way both systems will apply the "move" command at simulation step 36.

      Even with all this in place the game still gets out of sync. One reason I believe is that while the inactive team is re-applying state and catching up on simulation steps, it keeps falling behind in wall clock time as compared to the active team.

      Then when the active team makes its move done at step 36. The other team becomes active – and its lets say 500 milliseconds behind the previously active team now. So, now it starts doing teh same routine as an active team.. but this time the inactive team is far ahead in simulation steps. Kind of a mess 🙂

      I cant really assign object ownership to the teams. The two teams have 3 players each.. but then there is the ball

which belongs to no one.

i wish a simple 2d physics engine was deterministic and wouldnt use floats 🙂 the issue is that in my game teh playing field is so small that .001 makes a difference since everything is in motiong and players are flying at the ball trying to deflect it into the goal.. so after a while the ball is completely in different positins. I took care of this with the state sync every 300 milliseconds. but now one phone is so far behind the other one – sometimes by over a second.

Reply

68. *William C*
January 16, 2012 at 8:22 pm

Hi,

There's something I've always been curious about interpolating between the two different states. If say sprite animations were based off specific update states, how is that handle when the states are interpolated? For example, a player is at rest at the last state but now moving in the current state. When the player moves in the current frame, an animation is triggered. If alpha = 0.5, what do you do with the animation? If it is played, then the time passed for the animation will be a negative number. If I had to guess, I'd say to start the animation and if the time elapsed for the anim is less than 0, just display the first frame and if the time is positive, then the animation time is the actual program time minus a fixed step time. Can you clarify if this makes any sense? Thanks!

Reply

69. *Ingo*
January 15, 2012 at 2:30 pm

Hi Glenn,

great article! I find your interpolation technique interesting. Is this something that is commonly done in games? Somewhere above, you say that it does not create (visual) penetration. I don't see why not. Every inexpensive (relative to the actual physics computation) interpolation technique can create state that is physically not valid. If you take your example of a V-shaped bounce, interpolation could create a position that lies somewhere in white space, e.g., between the top corners of the "V". If there is another small object at that location that won't actually be hit, you will get visual penetration, don't you? Or am I missing something?

Cheers

Reply

-  *Glenn Fiedler*
January 16, 2012 at 6:47 am

  yes it has been used in many shipped games. of course there are inaccuracies with interpolation, but typically if the game runs acceptably at the simulation framerate — the interpolation errors are too small to notice. if you choose to extrapolate instead these errors are usually more noticeable. cheers

  Reply

  - *Ingo*
  January 18, 2012 at 10:21 am

    Thanks, Glenn. I can see why simple extrapolation can be more jerky, at least generally.

    Reply

70. *Mangan*

January 8, 2012 at 1:57 pm

Excellent post. Very insightful and well said. Continuing on your “Final Touch” approach, many people mentioned what I am about to state but I’m surprised this solution didn’t show up…

My concern is that rendering is always rendering (0,1] timestep behind. It will never be up-to-date.

My proposed solution follows the change from “previous/current” state mindset to one of “current/next”. We always solve up to the “nextState” and then interpolate from current to next. This shouldn’t be an issue under the concept that a single-simulation-iteration should take far less time than rendering. We also end out loop with a negative value accumulator (meaning how much time we must “wait” before we arrive at the “nextState”).

Code changes:

Current state = Next state (Purely for readability)
Previous state = Current state (Purely for readability)
Inner while-loop would read: “while (accumulator >= 0)”
Assigning of alpha value would read: “const double alpha = – accumulator / dt;”

Any reasoning that this wouldn’t work that you can see, and if not, then any reason that it would not work better? (Takes away the concept of always being 1 step behind.)

The only thing that comes to mind would be interactions (or any event) that would alter the way the state will integrate from the rendered state to the “next state” that has been stored. Easy work around here is to define that such non-determinstic events are to be applied after the time has been spent to reach the “next state”… which should be hardly noticeable I would think.

Looking forward to your response, and I apologize if someone else mentioned this and I misread.

Reply

-  *Glenn Fiedler*
  January 8, 2012 at 8:37 pm

  Measure the time between the player input and the frame incorporating that input to be displayed. Does your method actually reduce this, or are you just renaming prev/current to current/next?

  Reply

  - *Mangan*
    January 9, 2012 at 7:26 am

    I’ve gone over it again, with fresh eyes, and I see what you mean. Since alpha is never going to be 1, it is in sense equivalent. Although the way I said it seems more intuitive (to me, obviously), in terms of effect your version is the same… and arguably less likely to catch people up on the details than the derivative version I submitted.

    As others said, you have a knack for teaching – explaining what needs to be said in a manner that is easy to understand quickly. I’m looking forward to reading the rest of your posts.

    Reply

71. *brighthead*
December 14, 2011 at 5:50 am

hello glenn fiedler:
i spend almost whole day to get a fully perspective about your article,however ,i have to say thank you ,after i followed your method ,all but my confusions had been settled ,the only one left is that my project has been building with box2d,so when the world->step within every deltatime , there are some remainder left over ,by your advice ,i should put the interpolation between two frames,but how i can do that ? do you have any advice further particular or there are not at all needed the interpolation?because in my observation ,my game character now running smoothly in the world.

my code here is:

```
static double TIMEINTEVERAL = 1.0f/60.0f;
static double MAX_CYCLE_NUM = 5;
static double timeAccumulator = 0;

timeAccumulator += timeStep;
if (timeAccumulator > MAX_CYCLE_NUM * TIMEINTEVERAL) {
timeAccumulator = TIMEINTEVERAL;
}
while (timeAccumulator > TIMEINTEVERAL) {

timeAccumulator -= TIMEINTEVERAL;
world ->Step(TIMEINTEVERAL,
velocityIterations,
positionIterations);

}

if (timeAccumulator > 0) {
const double alpha = timeAccumulator / TIMEINTEVERAL;

world -> Step(timeAccumulator, velocityIterations, ); // here i dont know how to do ???
}
```

hopefully ,my expression is clearly

Reply

-  *Glenn Fiedler*
  December 15, 2011 at 10:29 pm

  Hi. When you interpolate you want to linearly combine the previous and the current frame positions and orientations according to the alpha factor. You only need to do this if you completely fix the timestep and have the accumulator left over. Perhaps you could just try the partially fixed timestep instead, where you just simulate the remainder of time.

  cheers

  Reply

- *phil*
  June 5, 2012 at 9:21 am

  Hi brighthead,

  did you solve this with box2d?
  i am very interested in your solution.
  please give me a hint to: its.just4phil at googlemail com

  thanks
  phil

  Reply

72. *Aaron*
    October 30, 2011 at 10:23 pm

    Should this method of fixed time step only apply to physics updates and have separate update for other things?

    .

ıe:

while (accumulator >= dt) { … OnFixedUpdate(); … }
OnUpdate();
OnRender(alpha);

OnFixedUpdate() { updatePhysics(..); }
OnUpdate() { updateScene(); updateGUI(); etc }

Reply

-  *Glenn Fiedler*
  November 2, 2011 at 12:10 am

  you can go either way

  Reply

73.  *Orson Peters*
October 25, 2011 at 10:29 pm

Very nice read. I was just wondering, when you are talking about interpolation, are you talking about linear interpolation, like this?

interpolated_x = alpha * previous_x + (1 – alpha) * current_x

Reply

-  *Glenn Fiedler*
  October 26, 2011 at 7:06 am

  Yep.

  Reply

74. *Davis Issac*
October 15, 2011 at 6:14 pm

this was very helpful

Reply

75. *Martin Felis*
October 5, 2011 at 9:08 pm

Hi Glenn,
just wanted to say thanks. The described method is brilliant and so is this article!
Awesome!
Martin

Reply

76. *Jean*
September 12, 2011 at 10:32 pm

Many thanks, it’s working great :-))

Reply

77. *Jean*
September 6, 2011 at 5:02 pm

Hi Glenn

Thanks for such a nice article.

In “the final touch” paragraph, for your example:

– Can I assume that the following code
const double dt = 0.01;
equates to a maximum frame cap of 100FPS? (10millisecs per frame)

– And could you then please confirm that the following code equates to a minimum frame cap of 4 FPS? (that is 250millisecs per frame)
if (frameTime > 0.25)
frameTime = 0.25; // note: max frame time to avoid spiral of death

Many thanks for your great effort.

Kind regards,
Jean

Reply

-  *Glenn Fiedler*
September 7, 2011 at 10:01 pm

  yes and yes

  Reply

78.  *Jez Hammond*
August 14, 2011 at 2:33 pm

Hi Glenn, thanks for the excellent infos.

If separate threads are used, then when to base alpha on? In my case the physics will run at 60Hz, and render will have forced v-sync usually at 30Hz. I’m hoping to interpolate and avoid stuttering. But now the render thread doesn’t care about the physics threads’ accumulator remainder or ?

Cheers!

Reply

-  *Glenn Fiedler*
August 15, 2011 at 12:52 am

  one thread produces state, the other one consumes it — you’ll have to come up with a technique to work out the minimum delay is required to ensure that the render thread always has simulation state to consume. this is a closed form equation you can work out assuming you have min/max bounds on render time and the sim thread is constant (and sim thread does not skip frames…)

  cheers

  Reply

79.  *terryhau*
[August 4, 2011 at 4:02 pm]

To avoid the 1 frame lag,
Instead of interpolating between the previous state and current state, is it possible to step the physics one extra time to get a “future” state and then interpolate between the current state and the “future” state?

[Reply]

-  *Glenn Fiedler*
[August 7, 2011 at 4:51 am]

I’m not sure. Why don’t you try it and see?

[Reply]

  -  *Glenn Fiedler*
[August 14, 2011 at 4:47 pm]

You would need to measure delay between input and visual result on screen to prove that it works.

[Reply]

80. *Martin*
[July 23, 2011 at 3:23 pm]

Why are you setting the

deltaTime to 0.01; in the “The final touch” example,
and to 0 in the “Free the physics”?

[Reply]

-  *Glenn Fiedler*
[July 24, 2011 at 7:26 pm]

But i’m not, dt = 0.01 in both cases as far as I can see. Are you confusing dt and accumulator?

[Reply]

  - *Martin*
[July 26, 2011 at 12:12 am]

Sorry thats what I meant
double accumulator = 0.0;

and in the second case, where dt = 0.01
double accumulator = dt;

What doesnt that start also with 0.0?

[Reply]

  -  *Glenn Fiedler*

August 4, 2011 at 1:38 am

Fixed. I think 0.0 is more correct. Not 100% sure though 🙂

Reply

81.  *opatut*
July 18, 2011 at 2:48 pm

I have already used this great article many times for different projects I am working on. For timing, I always return here. This never gets obsolete. Well done!

Reply

-  *Glenn Fiedler*
July 25, 2011 at 12:49 am

Thanks

Reply

82. *planetfunk*
May 15, 2011 at 1:27 pm

Hi Glenn,

Great articles! I'm finding them really informative. I've created a javascript demo/example from "Integration Basics, and "Fix your Timestep!".

http://mndlss.com/2011/05/rk4-in-javascript/

Looking forward to reading the rest – and given time – implementing them in my game as well.

Reply

83.  *Bharathan Rajaram*
May 10, 2011 at 12:41 pm

Hi Glenn,
Thanks for the reply. I do understand that the curve would flatten out if we interpolate. I only suspected it at first and I'm glad to have it confirmed. If I guessed ahead before calling the collision response code, wouldn't the trajectory be corrected? I do realize that I would have to call the collision detection and response code twice, but would this strategy work?

Reply

-  *Glenn Fiedler*
May 13, 2011 at 3:26 pm

Possibly, but then you are no longer really doing a decoupled rendering and physics. You'd be basically doing continuous collision detection and collision response with variable dt. This could work well for bouncing points against infinite planes but generally, it's better to just fix the timestep at least and just stick with interpolation. Consider the extrapolation more of a "guess" … you could attempt to say, bound that guess via collision detection, if it will help. It's an interesting idea. cheers

Reply

-  *batbrat*
  May 10, 2012 at 9:36 am

  Hi Glenn, I'm sorry I haven't replied to your comment for a year. I somehow missed the comment notification and I haven't been back on the site or doing physics simulation for a while now. Thank you so much for your reply and your article. I have more questions, but I need to get my head sorted first.

  Thanks again!

  Reply

84.  *Glenn Fiedler*
May 9, 2011 at 4:16 pm

If you interpolate you'll see no penetration, but on bounces you'll see the curve flatten out strangely. consider a bounce like a "V" but the bottom point of the V is not a sample point, now the bounce is flattened out.

If you interpolate instead of extrapolate, you are "guessing" ahead, and in the V case you'll actually guess a bit into the surface before bouncing out, at this point you'll see discontinuity due to extrapolation. So don't extrapolate.

cheers

Reply

85.  *Bharathan Rajaram*
May 8, 2011 at 1:56 pm

Hi Glenn,

You've already discussed why interpolation works better for this case. However, I don't quite understand why you'd see jerks/snap-to-place stuff happening if you follow the following scheme:
1. Integrate
2. Test for collisions
3. Respond to collisions
within the (accumulator >= dt) loop. Except that you modify it to (accumulator > 0)
Now, you'd be interpolating between the current and the next state. But if there was an object bouncing off at 45 degrees (i.e. a sudden change in velocity) then we'd have the correct result irrespective of interpolation or extrapolation, wouldn't we?

I'm very confused by this issue and would appreciate your walking me through the problem, if you have the time to do it.

Reply

86. *anon*
April 7, 2011 at 6:03 pm

aha

currentstate*alpha + previousState*(1.0-alpha)

cs*alpha + ps*1 – ps*alpha
-> (cs-ps)*alpha + ps which imho much more comprehensible…

Reply

-  *Glenn Fiedler*
  April 7, 2011 at 6:09 pm

Sure. But from my friend Fabian Giesen (@rygorous)

"lerp(t,a,b) = (1-t)*a + t*b eval'd as a + t*(b-a) is well known but has precision issues: lerp(1,a,b) != b generally."

also,

"Less well known+also 2 ops with FMA/FNMS: lerp(t,a,b) = b*t + (a – t*a) – and exact for t=0, t=1."

Reply

87. *Gurki*
March 16, 2011 at 9:23 pm

Hi Glenn!

I personally enjoy simulating time-scaling, such speeding up, slowing down or even reversing time. Your method is great, but the while-routine simply stops when the elapsed time is negative. This could be solved e.g.

```
while ( abs( accumulator ) >= dt )
{
previousState = currentState;
integrate( currentState, t, sign( scaled_elapsed_time ) * dt );
t += sign( scaled_elapsed_time ) * dt;
accumulator -= sign( scaled_elapsed_time ) * dt;
}
```

I wrapped the whole time-functionality in a Timer-Class, which makes things easier, but I think you got the idea.

Great articles though, all of them! Looking forward for new ones 🙂
Sincerely,

Gurki

Reply

-  *Glenn Fiedler*
December 9, 2011 at 8:10 am

  Nice idea, thanks!

  Reply

88. *Richard*
November 13, 2010 at 2:14 pm

The code in the "Final Touch" section calculates the frameTime but does not use it.
Is this right or should the line:
accumulator += deltaTime;

instead read
accumulator += frameTime;

Thanks

Reply

-  *Glenn Fiedler*
November 13, 2010 at 4:52 pm

yes, it was meant to use frameTime instead of deltaTime – thanks!

Reply

89.  *Mitchell*
September 4, 2010 at 11:56 pm

Wouldn't it be a lot simpler to just have a max time delta, so a physics step is never bigger than that maximum? Like this, keeping with your pseudocode:

```
double currentTime = hires_time_in_seconds();
State state;
double maxdt = 1.0 / 60.0;

while ( !quit )
{
double newTime = hires_time_in_seconds();
double dt = newTime – currentTime;
currentTime = newTime;

while (dt > maxdt) {
state.physicsStep(maxdt);
dt -= maxdt;
}
state.physicsStep(dt);

render(state);
}
```

Reply

-  *Glenn Fiedler*
September 6, 2010 at 3:18 am

It's a lot simpler but it does not step forward with fixed delta time. If you don't care about completely fixed delta time then it's a perfectly good technique. I have a reworked version of this article which includes this as one of the options discussed.

Reply

-  *Glenn Fiedler*
September 6, 2010 at 6:50 pm

I've published the article update, let me know what you think!

Reply

  -  *Mitchell*
September 6, 2010 at 9:58 pm

Yea, that's what I meant, thanks a lot for covering it, I understand it better now.

Reply

    -  *Glenn Fiedler*
September 7, 2010 at 3:39 am

There is more to come ... an analysis of input latency and multithreaded approaches etc, but these will have to wait

There is more to come… an analysis of input latency and multithreaded approaches etc. but these will have to wait for a bit later. I felt the old article was getting a bit stale and I'm a bit happier with it now! cheers

Reply

90.  *Gregor*
July 16, 2010 at 3:53 am

Great tutorials, thanks for taking the time to write them!

What if the update frequency was dynamic as in per computer or per scene change or even per fps drop. Take for example a simple game which fps will vary a lot (from 30-400fps on different computers). Would changing the physics update step help the game at all and when should it be done if its not in the physics initialization (after updates were done on the objects).

Reply

-  *Glenn Fiedler*
July 16, 2010 at 9:13 am

  if your game physics is mostly simple linear motion, you'll probably get no real benefit. if you have more complex physics you'll benefit from more consistent behavior with fixed timesteps.

  so, work out how differently your game sim code behaves depending on framerate and if it's significant or problematic then fixing the timestep is a good idea. some examples of where this is good, tunneling at low framerates if you don't raycast, different behavior of springs at different delta time due to integrator errors etc.

  Reply

91.  *Glenn Fiedler*
July 15, 2010 at 9:18 pm

you can interpolate, you can extrapolate — if you interpolate you'll add *up to* one frame of latency depending on how much accumulator you have left. the alternative is to extrapolate forward networking style and trade latency for misprediction

cheers

Reply

-  *Gaute Løken*
July 15, 2010 at 11:43 pm

  With your code you're interpolating between the states that happened the current state (which happened some accumulator time ago, and previousStep which happened 1 dt + accumulator time ago. So by interpolating the state at previousTime + accumulator you're rendering the interpolated state at exactly 1 dt behind rendertime, not up to 1 dt.

  With my proposed method I would interpolate like you, but on a predicted state rather than the two previous states. What I'd like to know is why one method would be preferable to the other, or if you'd think the way I propose would be feasible/reasonable.

  Reply

  -  *Glenn Fiedler*
  July 16, 2010 at 9:15 am

    either way is good and it depends on the error you consider most acceptable during a sharp change in velocity – consider a particle moving very quickly and bouncing @ 45 degrees, cheers

    Reply

Reply

92. *Erwan*
June 30, 2010 at 8:30 am

Hi ! I find these articles on physics very useful.
I wanted to ask Glenn a question:
If I had to buy a game's physics book which one would you recommend ?
I am looking for something that covers the basics and provides solid tools such as the ones presented in your articles.

Thank !

Reply

-  *Glenn Fiedler*
June 30, 2010 at 9:58 am

I recommend:

"Essential Mathematics for Games and Interactive Applications, Second Edition: A Programmer's Guide" by James M. Van Verth and Lars M. Bishop

"Real Time Collision Detection" by Christer Ericson

Also, the siggraph course notes by Baraff are a great introduction to physics simulation.

Reply

  - *Erwan*
July 1, 2010 at 3:42 am

Thanks a lot, I'll try "Essential Mathematics for Games and Interactive Applications, Second Edition: A Programmer's Guide"

Reply

  -  *Gregor*
July 17, 2010 at 1:47 am

siggraph course is brilliant, thank you

Reply

93.  *George Stagas*
June 29, 2010 at 5:34 am

You are fantastic! I was having a really hard time getting a Javascript physics animation to work smoothly and accurately on all browsers/all systems, and your solution fixes everything and required minimal coding to implement! The interpolation kicks ass, I can now do super slow smooth motion just by changing a variable! It's great for replays etc. and keep everything consistent among browsers and their crappy timings. Thank you!

Reply

94.  *Manuel Bua*
June 11, 2010 at 1:01 pm

@David

@David
Don't use time(), it will be wrong on multicore processors: if available, use monotonic time.
On *nix systems you can use "clock_gettime(CLOCK_MONOTONIC, …" and linking with -lrt

Reply

95. *David*
June 9, 2010 at 5:00 am

Great article.

Only problem I encountered was instances of negative deltaTime (which made the dot animation vibrate as it moved). Apparently this can sometimes happen on duel-core CPUs. Easy way around this was to simply add another loop to update deltaTime until it is positive again (ignoring it basically).

…
float newTime = time();
float deltaTime = newTime – currentTime;
while(deltaTime < 0){
newTime = time();
deltaTime = newTime – currentTime;
}
currentTime = newTime;
…

Reply

96.  *Glenn Fiedler*
May 25, 2010 at 5:39 pm

also for what its worth, using a float value to accumulate time t from dt is also craziness, please consider that you should have some 64bit fixed point representation of time if you use this technique, otherwise the precision of your time t gets worse the longer your program runs. i only use float here for clarity because i use it for the dt and accumulator only. if you have a truly fixed timestep, then you can use your integer frame count as time

Reply

97.  *Glenn Fiedler*
May 25, 2010 at 5:37 pm

i agree that checking if two floats are equal is very bad, but as far as I can tell conditionals you presented are exactly equivalent:

bool a = accumulator >= dt;
bool b = ! (accumulator < dt);
assert(a == b);

(try this and let me know)

of course, bool c = accumulator == dt;

is craziness…

Reply

98. *mariusz j*
May 25, 2010 at 2:25 pm

Hi,
Thank you for great article!

Thank you for great article!

I have on remark, though.
I think that it would be better, if instead of following condition:

while (accumulator>=dt)

you've used:

while (!(accumulator < dt))

Comparing if two floats are equal can get you into trouble.

Reply

99. *Gregory Defaisse*
May 19, 2010 at 8:13 am

Thanks for the article I get my timestep fixed !

But for those you think it's a bit complex to use a coefficient to calculate the exact position of all their physic objects and than it complicate the calculate and codes, I found a simple solution costing only some part of 1 frame and doing the same exact result.

In fact you just have to to wait some milliseconds to have a round count of frames, for example, your render function took 88ms and your physics frame is fixed to 10ms then you wait 2ms and calculate the number of physics frame to do (88+2)/10 -> 9 you proceed and render again, so simple!

Reply

100.  *Glenn Fiedler*
May 17, 2010 at 5:55 pm

this is actually very hard, i remember a paper about something pixar did for Monster's Inc where they ran a solver to solve motion for rigid bodies to pass through various points and end up at a specific configuration (e.g. one side up)

i would say that you should try to find this paper and read about how they do it

Reply

101.  *KIVagant*
May 17, 2010 at 3:12 pm

Hello.
Can you help me, please? I need to solve a simple problem.
I created a cube (JBox or Cube). The cube fly from same top point and then rolling on the plane. How can I make, that cube always stopped the same one side up?

Sorry for my bad English.

Reply

102. *Carlos Hernandez*
April 16, 2010 at 9:07 am

Nice article. I have developed a game for the iPhone, and the game renders and it is played beatifuly on an iPhone 3GS or 3rd generation iPod touch, but on older hardware it starts fine, but at some point on the game, especially when loading textures, and because of memory constraints, I need to start clearing the image cache, and that eventually lowers my fps, after a few seconds everything comes back to normal. But the thing is, that under those few seconds, some objects that were falling, suddenly they appear on the ground, and the point of the game is to kill this objects in mid air. So my question is, do you think that implementing the

on the ground, and the point of the game is to kill this objects in mid air. So my question is, do you think that implementing the rendering interpolation and/or having the calculations and rendering algorithm on different threads might eliminate this problem? I have implemented the fixed time step, but not the rendering interpolation. Here is the link of the game to give you an idea: http://www.softwarefactoryllc.com/airassault

Thanks.

Reply

-  *Glenn Fiedler*
  April 17, 2010 at 12:39 am

  you could move the texture loading and other stuff onto a separate thread so they don't hitch up the simulation – this is the best way to get around your problem. interpolation as described in this article won't help you. moving the sim off to a separate thread is another way to look at it, but i'd first try getting the loading async – cheers

  Reply

  - *Carlos Hernandez*
    April 20, 2010 at 10:53 am

    Thanks for your response. I tried all the solutions, but what really did the trick was clamping the dt values to 0.1. Now the game is super responsive in any situation on all devices, and even if it gets slow, at least the physics and rendering engines are in sync always.

    Thanks for a great article, and for taking the time to clear my doubts.

    Reply

103. *Jamie*
March 30, 2010 at 11:58 am

Excellent article. It is hard to find such clearly written material regarding game physics. Great job 🙂

Reply

-  *Glenn Fiedler*
  March 30, 2010 at 12:40 pm

  Thanks! Ironically, it's the very first article I wrote waaaaaay back in 2004… and it's still the most popular. Despite many attempts to improve my writing style. I should just give up. This article is always 2x more hits each day than any other =)

  Reply

104.  *ioio*
March 29, 2010 at 11:58 pm

at this point, when running the integrate function in the physics loop, is this call completely CPU independant ? I understand we are independant from rendering, but are we independant from cpu speed for simulation ?

Reply

-  *Glenn Fiedler*
  March 30, 2010 at 10:39 am

  I'm not really sure what you mean. Yes it's CPU independant, to a degree assuming you can simulate one physics step of dt in

I'm not really sure what you mean. Yes it's CPU independent, to a degree assuming you can simulate one physics step of dt in less than dt/2 in real-time, you should be fine. Otherwise you may get spiral of death given that you may have to simulate 1-1-2-1-1-2 etc. to speed up occasionally when the sim rate is lower than the render rate. Alternatively, you could move the sim into it's own thread or process to be completely asynchronous to render.

Reply

105.  *Manuel Bua*
March 3, 2010 at 3:03 am

I forgot to mention in your sample code you are also interpolating the velocity, not only the screen position: obviously you intended to interpolate the screen state with properties like position, rotations and scaling, but this may lead other people to think they need to do the same with other forces, as well as trying to inject these lerp'ed values back in their physics engine.
I was reading Box2D forums and some code around and, sadly, most of them seem to misunderstand this very important point: too bad since it's a very well-written article on a too much often undervalued subject.

Reply

-  *Glenn Fiedler*
March 4, 2010 at 12:37 am

You are correct in that the velocity does not need to be interpolated. The code interpolates the entire physics state for the object but in fact the only interpolated quantity which is actually used is the position.

Reply

106.  *Manuel Bua*
February 22, 2010 at 4:30 pm

Oh well, i got it fixed, i wasn't interpolating the correct values due to a bad git commit..

Reply

107. *Anonymous*
February 21, 2010 at 5:14 pm

> **Glenn Fiedler :**
> to take it even further, you can run the main physics loop entirely on a separate thread which actually runs (in real time) at the number of frames per second the physics should run (say 100hz) and the render thread just reads latest physics state and interpolates, completely independent of physics
> this way you are never doing two or more physics updates per-update, so processing is more evenly distributed
> in addition, if you gather the input for you simulation on the physics thread, you sample the joystick or keyboard at regular 100hz intervals, which leads to better behavior of your physics
> consider this, "fix your timestep version 2.0″

Hi Glenn, thank you for such a beautiful article, nicely done!
I implemented a testbed with box2d and i have the renderer running in the main thread, while the physics and input are running on another one: i'm experimenting the better way of doing things and at this point i can say from my tests that a strict producer/consumer model (that is, a frameNeeded semaphore and a frameReady semaphore) make the physics thread fluctuate since it's need to wait for the render thread to signal a frameNeeded.
However, implementing it with a mutex protecting the "onRender" call (main thread) and the same mutex protecting the "onUpdate" call (in the other thread), adding "usleep" to avoid both threads concurring much for the lock seems to work flawlessly.
The thing i can't get to work is the render thread to do subframe interpolation: i mean, everything is fine if the update thread do the subframe interpolation/time aliasing lerp, but if it would work in the render thread as you said it may need less cpu time if vsync is on.
Anyway, let me know if i can send you my code for your own tests: i see you are planning a "fix your timestep v2" so i'm willing to help you out, although i'm running Ubuntu it shouldn't be a problem to make v2 cross-platform!

Reply

Reply

108. *Ilya*
February 17, 2010 at 4:00 pm

Many thanks for writing this wonderful article!

Thanks to it physical simulation in my game now runs much smoothly than ever before!

Reply

109.  *AngleWyrm*
January 31, 2010 at 12:08 pm

Could you please explain a little bit about why n steps of dt is often different from one step of n*dt?

Reply

-  *Glenn Fiedler*
  January 31, 2010 at 1:25 pm

  In general, physics simulation is too complicated to solve analytically so it is solved via numerical integration. Numerical integration is an approximation to the exact solution in which the accuracy of the result depends on the size of time-step. Typically numerical integration gives more accurate results for smaller time-steps. It follows that one step of n*dt is not the same as n steps of dt.

  Reply

110. *Mike M*
November 20, 2009 at 4:37 am

You are a gentleman and a scholar.
Many thanks for this guide!

Cheers!

Reply

111. *Quincy*
September 16, 2009 at 3:24 am

This may sound very dumb but what is state ?

Reply

-  *Glenn Fiedler*
  September 16, 2009 at 10:30 am

  state is the current attributes of a body in the simulation

  for example for a rigid body state is position, velocity, orientation and angular velocity

  for a car, state could have all of that information but also include the RPM of the engine, the current gear, the suspension lengths for each wheel, the speed of each wheel rotating and so on

  basically, state is the thing which you step forward by delta time (dt) each step of your simulation

  cheers

cheers

Reply

112.  *quano*
September 3, 2009 at 12:31 pm

Awsome article. I've implemented this in my game and it works splendidly!

Reply

-  *Glenn Fiedler*
  September 3, 2009 at 3:12 pm

  cool, glad you like it – cheers!

  Reply

113. *sigzegv*
August 23, 2009 at 12:54 pm

Great article, thank you very much.

Just to be sure (i am new in game development), if I understand, any code writen inside the 'while(accumulator >= dt)' is totally independant from framerate, is that right ?
So is this the good section to implement any game mechanics that have nothing to do with rendering ?

Reply

-  *Glenn Fiedler*
  August 23, 2009 at 1:06 pm

  precisely, the game simulation update goes inside the while loop – anything which is per-render frame goes outside – eg. perhaps you have some visual only particle effects, these could be updated at render frequency if you wish

  Reply

114. *Mc Jman*
July 28, 2009 at 2:31 am

Sorry, but I'm having a bit of trouble implementing this in obj-c
I ported the code from the download example you provided and my version compiles and returns results, but the values seem way off.

One method in particular has me wondering….

```
float acceleration(const State &state, float t)
{
const float k = 10;
const float b = 1;
return – k*state.x – b*state.v;
}
```

Why isn't the value of "t" used in this function?

Thanks

Reply

-  *Glenn Fiedler*
  July 29, 2009 at 10:15 am

  the value t is not used in that function, it is just passed in in case you want an acceleration that is a function of time, eg. sin(t) – in this case, acceleration is just a function of distance from the center point and velocity (it is a simple damped spring)

  cheers

  Reply

115. *Daniel SL*
April 7, 2009 at 6:32 am

You're fuckin fantastiiiiiic! Haha

This article really save.. Only 2 words: THANK YOU!

Greetings from Argentina,
Daniel SL

Reply

116. *Sirius*
February 5, 2009 at 12:08 pm

Ah, works perfectly!
Also, thank you for your advice on fast forwarding, makes sense too. 🙂

Thank you very much for your help~!

Cheers~!

Reply

117. *Vorn*
February 3, 2009 at 9:02 pm

I am curious: how does one handle interpolation of objects that do not exist in one frame or the other? for instance, a bullet only exists in the frames between being fired and hitting its target. Does one simply not display these objects? Or would something more involved be preferable?

Reply

-  *Glenn Fiedler*
  February 3, 2009 at 11:12 pm

  if an object is created, you just set its previous position same as the current, eg. dont interpolate

  if the object is deleted, then obviously you dont need to interpolate it because its no longer there

  of course, to be smooth about it fade-in fade-out works fine, during which time you'd interpolate normally

  cheers

  Reply

118.  *Glenn Fiedler*
January 19, 2009 at 6:54 pm

in this case there is no need to extrapolate, since the rendering is performed synchronously with the simulation in one process – of course, if you had a server process running the simulation and a client process viewing it, then yeah it would make sense to have support for extrapolation

Reply

119.  *Fredrik*
January 12, 2009 at 3:01 am

Hi,

nice article! Although I've been using these techniques for a while, I never thought of it this way. Rather, these are classic client/server game design (client doing rendering, server doing physics and stuff). The interesting part however, that one that threw me a bit off, you interpolate between last and current frame (effectively adding a frame of latency). Wouldn't make more sense to extrapolate (sometimes called prediction) from the current frame onto the next, and compensate when the frame is finished? The reason for this is two-fold, one, as you describe, to smooth the animation, and two (which is more important if you're a network jock like me) to compensate for packet loss.

Reply

120. *Khu*
December 30, 2008 at 1:36 pm

Thank you so much! This is awesome! =) I gotta read up on that linear interpolation thingy though.

Reply

121.  *Glenn Fiedler*
December 12, 2008 at 4:16 pm

yes thats exactly what i'm doing, interpolating between the previous frame and the current – effectively adding one frame of latency

cheers

Reply

122. *space*
December 12, 2008 at 3:43 pm

I see now, that the spiral I mentioned can be avoided by making sure the timestep is big enough. Of course there will be a lower limit of how slow the computer can be then, but you can't expect your games to work on a 286 anyway 🙂

I also wondered about the interpolation.

I understand the reason to interpolate and not extrapolate, but it doesn't make sense though, since the position you are getting is a previous one.

To do the interpolation for the correct position, you should use the current position and the NEXT position, meaning you would have to always calculate the next step, a frame before you actually render it.

Maybe that is what you are already doing, and I just didn't catch that in your code. I don't know.

I was also thinking, that your clamp the deltaTime to some max, it would mean that you couldn't use it for realtime multiplayer

I was also thinking, that your clamp the deltaTime to some max, it would mean that you couldn't use it for realtime multiplayer games, since one computer could get behind. But I guess it could under all circumstances, since the timers might not run the same on both. I guess in such cases you would need some time signal from the server to keep it accurate. But maybe these differences would be so small that it wouldn't be important.

By the way, thanks for the article, it was useful for me.

Reply

123.  *Glenn Fiedler*
December 8, 2008 at 6:10 pm

yes i actually prefer the separate thread approach myself, and it has better CPU patterns than the approach in this article, avoiding having to simulate 2 or more sim frames per render frame in the common case, spreading it out properly with frame sleeps — it also avoids the "input aliasing" that you get with this approach, which is subtle, but there

or even better — put the simulation in a different process and connect a "view client" to it, using a simple network model with interpolation/extrapolation

ps. the reason i interpolate previous two frames is to always be accurate, you could extrapolate from the current frame if you want, but this causes discontinuities if the velocity changes abruptly (eg. a bounce…)

cheers

Reply

124. *Netherby*
December 8, 2008 at 7:30 am

Hehe, well if it takes longer than your time step to simulate that amount of time, you're in a bit of trouble! You probably need to increase the size of your step..

These are some very good articles!

Though I'm slightly baffled by the interpolation used.. You're dropping back one step then interpolating by alpha towards the most recent known data.. Is that really the best way to handle it? I guess it's the only accurate data..

Any way I was thinking that having the rendering done in a separate thread would be the best solution, but it adds complexity and makes things harder to debug.. So I was considering going with the single threaded approach for initial revisions and then moving the rendering to another thread.. But do you think this would just cause more pain in the long run than just starting with that approach?

Reply

125. *Ape-Inago*
November 30, 2008 at 8:43 pm

if you've already got code that can interpolotate a timestamp… if your simulation can't keep up, you'll atleast know how much it isn't keeping up by the accumulator being > than 1/60th (or whatever rate you are intending to get)… and can get a rough interpolotation since the last time.

maybe that didn't make much sense, but it worked decently enough for me.

Reply

126.  *Glenn Fiedler*
November 27, 2008 at 12:22 pm

sure thats actually correct, but the premise of this article is that you can actually afford to simulate one timestep in <= the actual time it is simulating

it is simulating

so in your case, you should simply use a variable framerate, or optimize your code down so it fits in budget

cheers

Reply

127. *space*
November 27, 2008 at 1:31 am

I've discovered a problem with this method.

If the computer cannot keep up, meaning it's been longer since last frame, then you have to calculate several physics states for that frame.

This will mean that this frame will take even longer to calculate for the computer, and thus when you get to the next frame, even more time has passed, and you will have to do even more calculations for that frame, which will take even longer.

It's an evil spiral, that means once it just gets slow enough once, for the computer to not be able to keep up, then the accumulation will just keep growing bigger every frame.

Reply

128. *Soj*
November 26, 2008 at 11:47 pm

It seems to me that with this method there will always be a remainder. It's unlikely that the time would happen to fit with a multiple of your dt, so you will always be doing interpolation.

You write: "We can use this remainder value to get a blending factor between the current physics state and the previous state simply by dividing by dt"

If you want to interpolate the next point, you should be getting the the blending factor between the current and the next state, not the previous. So to do this interpolation, you have to also calculate the next step, even though you are not yet taking it.

Reply

129. *Marcelo*
November 11, 2008 at 7:53 pm

Exactly, I am getting this exact problem right now.
What was hard for me to "get" was the part that, on the last code snippet, we're not rendering something that is the latest physics step we know of, we're rendering 1 dt BEFORE that!

That was kinda weird for me at first, but it makes a little more sense now. As soon as I can figure out how to get/set all of the "state" of the simulation (I am using a framework, Box2D, so it encapsulates the state), I'll try that and see how it goes. Thanks again!

Cheers,
Marcelo.

Reply

130.  *Glenn Fiedler*
November 10, 2008 at 11:29 pm

yes, you are stepping forward at nice physics intervals like 1/60th of a second, but rendering at another rate – since you have no way to get the exact position at the real time of render, you just interpolate between the two nearest physics steps to get the approximate position at the render time t, so that your rendering is smooth

position at the render time t, so that your rendering is smooth

if you don't do this, you get temporal aliasing – eg. jittery movement

cheers

Reply

131. *Marcelo*
November 10, 2008 at 11:21 am

I am a programmer and wannabe-game-developer, and am currently trying to tackle a project that is much larger than my skills. Just 'cause. 🙂

Being that so, I have to thank you very much on this article, it's been really helpful. It's also taken me several hours to wrap my head around the "final touch". I've been able to "free the physics" and, of course, experienced the problem you mentioned.

So here's how I understand it: In "free the physics", we are always rendering the latest physics step we have, even though we're a bit ahead of it in time. On transitioning to "the final touch", though, we decide to render something that is between the latest physics step and the step BEFORE it. That would mean, as I understand, we're further away from "the truth" (the state as it would be in current time) than we were before deciding to interpolate, but that means better animation.

Do I understand it correctly?

Thank you again for this,
Marcelo.

Reply

132.  *Glenn Fiedler*
October 4, 2008 at 10:57 am

damian, yes i agree the absolutely correct pattern is a producer/consumer model with the simulation running in another thread or process from the renderer – i'll be writing a new article about this at some point i hope

Reply

133. *Bob*
August 11, 2008 at 5:50 pm

For the Windows alt-tab issue, the other solution is to run the physics code even when the renderering code isn't happening. This should free up the CPU time being used for graphics, and given the technique in the article wouldn't need to run perfectly smoothly anyway.

This is especially important in multiplayer games where you can't just automatically pause the game.

Reply

134.  *Dunge*
August 11, 2008 at 4:46 pm

I was reading your article and tough it was perfect, but I just though "integrate" meant the physics update/game logic/input pooling and I though it was working this this, but after checking the source code, integrate is really a mathematical integration, and I don't quite understand what it is doing there. I also don't quite understand the constants used in the function, and what are x and b are supposed to mean in your State structure..

Reply

-  *admin*
  August 1, 2010 at 9:01 am

  Basically integrate just means to advance the simulation state forward. You can try it with something simpler like euler integration, eg. x += dxdt * dt if you like. cheers

  Reply

135. *David*
May 21, 2008 at 6:49 pm

Hi Glenn,

Thank you for this article, as a beginner in game development this is a great help and I will definitely make my game loop like this.

I also like the way you separate your window code from your game code, it makes everything so much more clean.

I got a question though, when I run your example I can still see the dot stuttering from the moment it goes left for the second time and the stuttering stays until it slows down in the middle.
Is that normal? Am I being too picky? Do you have the same effect when you run it or your computer?

Anyway, again, thanks a lot for this.

Regards

David

Reply

136. *Macguffin*
May 5, 2008 at 11:41 am

Hi Glen,

That's an excellent article. Thanks!

Reply

137.  *Cyde Weys*
March 10, 2008 at 9:00 pm

Thanks for the great article. You have the optimal solution here that I really wish I had known about when I was screwing around with physics simulation Java applets back in high school. The animation was just never quite right.

Reply

138. *Naveen Nattam*
January 9, 2008 at 11:31 am

I just wanted to give a huge thanks to you Glenn. Your method helped me out incredibly! I have way more control over my physics simulation in my app now! Thanks again!

Reply

139. *Darklawl*
September 28, 2007 at 4:38 pm

September 28, 2007 at 4:58 pm

Great article which nicely describes a solution for a “common” problem!

Reply

140. *Gregoris jose*
July 1, 2007 at 5:15 pm

Hi cheers

I cannot make him to work still, but I think that I am very close.

At the moment, I don’t use QueryPerformanceCounter .
I wanted to ask to you.
How much can this affect in the problem?.

At the moment I has a undersampling problem.
My fps fluctuates between 29 and 35.

Advance thank you

KIKO

Reply

141.  *Glenn Fiedler*
July 1, 2007 at 5:34 pm

chances are the precision of your timer is probably too low – upgrade to QueryPerformanceCounter and see how it goes

Reply

142.  *Glenn Fiedler*
June 27, 2007 at 8:30 pm

good point, fixed!

Reply

143. *Cazy*
June 27, 2007 at 12:03 pm

What confused me is that the actual current time is currentTime + acumulator, so it’s outside the (previousTime, currentTime) interval. But after thinking it trough I realised that what you’re doing is shift the fraction alpha back one inteval. This is complelty legal, as the game loop uses a fixed time step. So CurrentState could be vweyt well named NextState and PreviousState could be named CurrentStep.

Thank you for your time, reading your article and replies was very informative. One other question though (feel free to ignore it): why do you initialize currentTime to 0? Isn’t that going to give a wrong deltaTime the first time the game loop is entered?

Best Regards, Cazy.

Reply

144.  *Glenn Fiedler*
June 26, 2007 at 10:49 pm

interpolation is between two known values, so its “perfect” – just lagged by up to one frame

interpolation is between two known values, so its "perfect" – just lagged by up to one frame

extrapolation is jerky and is predicting ahead, thus gets it wrong, and has to snap and pop to correct

trust me you want interpolation

take the example source code and change it to extrapolation, and let me know how it goes! – maybe it'll be good for you, i'm conservative so i prefer interpolation

cheers

Reply

145. *Cazy*
June 26, 2007 at 1:46 pm

I am very impressed by the quality of the article too. I've read a few more on the subject but none was this clean and eloquent.

One comment though. The interpolation function in your code looks like this:

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

My calculations (and intuition) say it should be:

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

Am I wrong? Can you double-check your math?

Best regards, Cazy.

Reply

146.  *Glenn Fiedler*
June 21, 2007 at 10:38 pm

i think you should try a simple example first, and get your head around it — its not hard! but becomes complex when introducing middleware and "its way of doing things"

here is what i think you need to do with newton, always update it at 60fps (~16.7 ms per frame), but interpolate between according to render rate, as described in the article above

so newton is always creating state for you on integer frame numbers, exactly 1/60th of a second apart, while your renderer is not always doing this, it is working on arbitrary dt, — so you interpolate between the two nearest integer frame samples to find the position/orientation of your body at the render time

cheers

Reply

147. *Jose Gregoris*
June 20, 2007 at 7:51 am

Hi Glenn

Thank you for their attention.
I have a question more.
Suppose you, that I update the physics with a dt=0.016 that is the minimum that NewtonGD requires.
The range is of (1/60.. 1/1000) for NewtonGD.
I would update the physics X times before the render, you suppose 16 times.

With a smaller dt=0.01 I would be update the physics more times, you suppose 25 times

With a smaller dt=0.01 I would be update the physics more times, you suppose 25 times.

My question is:

Should I have a buffer the position/orientation coming out of newton's callbacks of variable size?
For example: 16 for dt=0.016 or 25 for dt=0.01.
Is this correct?.
if it is this way.
How I do make to manage after the update of the physics the render?

I implement the solution with 2 variables (currentMatrix previousMatrix), but I have the problem of the stuttering

I am really confused

Advance thank

Reply

148. *Leigh McRae*
February 21, 2007 at 8:33 pm

Very nice article. I was using this method for a game and it actually caused a very subtle bug. When playing very large in game cut scenes, the audio would become out of sync with the animations. I tracked the problem down to delta time accumulating error. The solution instead was to keep a simulation time and a real time. Then the loop is something like:

```
while ( (time() – simTime)>=dt )
{
simTime += dt
}
```

You still get error but it's less. It also works with a crappy timer 🙂

Leigh

Reply

-  *admin*
August 1, 2010 at 8:59 am

  This is a suitable implementation if you don't need a fixed delta time, but for general 100% repeatability and same results each time run, I generally prefer the full fixed dt myself. cheers

  Reply

149.  *Romz*
February 11, 2007 at 5:53 pm

Everything is now working perfectly, and the interpolation part makes a noticable difference ! Thanks a lot 🙂

Reply

150.  *Glenn Fiedler*
February 10, 2007 at 12:13 pm

use a higher resolution timer, 1ms resolution is next to useless, and is the root cause of your problem

use QueryPerformanceCounter directly!

Reply

151.  *Romz*
[February 9, 2007 at 5:15 pm]

Hello, great article! I'v implemented your technique but I'd like your advice about (mega)oversampling.

My project doesn't do much things so the framerate is insanely high with my PC (3000fps), and the timer of my engine (Ogre) gives me a deltaTime of 0, because it's has a precision of 1 milisecond, so 1/3000 = 0.00033s = 0.33ms (that's why I got 0).

I already had this problem with the SDL timer and I guess a QueryPerformanceCounter or any other high precision timer have the same limitation. So I thought putting some ugly Sleep() in the code to obtain a valid deltaTime. How do you think it's possible to do something about this ?

And even I if want to fix the display framerate, the problem is the same (adding 0 to any accumulator, so it's never time to update).

Thank you !

Reply

152. *Matt*
January 16, 2007 at 3:01 am

I thought I could get by without fixed timestep, however some testers were having issues with the game I'm working on – which were fixed by a fixed timestep. Thanks for writing the article, it saved me tons of time messing around.

Reply

153. *Manu*
December 29, 2006 at 7:36 pm

Thank you very much!
This solves all my problems 😀 (really!)

Reply

154. *Paul*
November 28, 2006 at 10:04 am

Excellent article and very helpful to my current situation. Thanks for taking the time to explain this in terms that non-mathematician games programmers can have a half chance of understanding.

Reply

155.  *Ruud van Gaal*
November 21, 2006 at 10:06 am

Given the last interpolation trick with 'alpha' blending between states, you'd think that for physics-intensive programs it may be important to really catch up with realtime. I.e. after the while(accumulator>=dt) loop finishes, do a new deltaTime calculation with a new time() call and re-enter the while loop if deltaTime is bigger than some threshold.
I've noticed with my car simulator that the physics can take up quite a bit of CPU time, making the last alpha blending call a bit useless if you're already straying away from the actual time.

Reply

156. *pTymN*
October 13, 2006 at 9:44 am

Also, fixed internal framerates help when you do networked games.

Reply

157. *JC*
September 29, 2006 at 8:17 pm

Thanks so much for posting this! After trying to figure this out after many hours, I was able to quickly come up to speed thanks to this article. Excellent work!

Reply

158. *Phillip*
September 15, 2006 at 11:05 am

Excellent articles on game physics. I'm starting to do some research on network-based state machines/physics and they were quite helpful.

Reply

159. *Andrew Ladenberger*
September 14, 2006 at 12:00 pm

Great article! Best article I've read on this important subject. Very nice and clean implementation too.

Do not worry about your difficulties in Mathematics. I can assure you mine are still greater
-Albert Einstein

Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Website

Post Comment

Articles

- Game Physics
  - Integration Basics