deWiTTERS Game Loop

The game loop is the heartbeat of every game, no game can run without it. But unfortunately for every new game programmer, there aren't any good articles on the internet who provide the proper information on this topic. But fear not, because you have just stumbled upon the one and only article that gives the game loop the attention it deserves. Thanks to my job as a game programmer, I come into contact with a lot of code for small mobile games. And it always amazes me how many game loop implementations are out there. You might wonder yourself how a simple thing like that can be written in different ways. Well, it can, and I will discuss the pros and cons of the most popular implementations, and give you the (in my opinion) best solution of implementing a game loop.

(Thanks to Kao Cardoso Félix this article is also available in Brazilian Portuguese, and thanks to Damian/MORT in Polish)

The Game Loop

Every game consists of a sequence of getting user input, updating the game state, handling AI, playing music and sound effects, and displaying the game. This sequence is handled through the game loop. Just like I said in the introduction, the game loop is the heartbeat of every game. In this article I will not go into details on any of the above mentioned tasks, but will concentrate on the game loop alone. That's also why I simplified the tasks to only 2 functions: updating the game and displaying it. Here is some example code of the game loop in it's most simplest form:

```
bool game_is_running = true;

while( game_is_running ) {
    update_game();
    display_game();
}
```

The problem with this simple loop is that it doesn't handle time, the game just runs. On slower hardware the game runs slower, and on faster hardware faster. Back in the old days when the speed of the hardware was known, this wasn't a problem, but nowadays there are so many hardware platforms out there, that we have to implement some sort of time handling. There are many ways to do this, and I'll discuss them in the following sections. First, let me explain 2 terms that are used throughout this article:

FPS

FPS is an abbreviation for Frames Per Second. In the context of the above implementation, it is the number of times display_game() is called per second.

Game Speed

Game Speed is the number of times the game state gets updated per second, or in other words, the

number of times update_game() is called per second.

FPS dependent on Constant Game Speed

Implementation

An easy solution to the timing issue is to just let the game run on a steady 25 frames per second. The code then looks like this:

```
const int FRAMES_PER_SECOND = 25;
const int SKIP_TICKS = 1000 / FRAMES_PER_SECOND;

DWORD next_game_tick = GetTickCount();
// GetTickCount() returns the current number of milliseconds
// that have elapsed since the system was started

int sleep_time = 0;

bool game_is_running = true;

while( game_is_running ) {
    update_game();
    display_game();

    next_game_tick += SKIP_TICKS;
    sleep_time = next_game_tick - GetTickCount();
    if( sleep_time >= 0 ) {
        Sleep( sleep_time );
    }
    else {
        // Shit, we are running behind!
    }
}
```

This is a solution with one huge benefit: it's simple! Since you know that update_game() gets called 25 times per second, writing your game code is quite straight forward. For example, implementing a replay function in this kind of game loop is easy. If no random values are used in the game, you can just log the input changes of the user and replay them later. On your testing hardware you can adapt the FRAMES_PER_SECOND to an ideal value, but what will happen on faster or slower hardware? Well, let's find out.

Slow hardware

If the hardware can handle the defined FPS, no problem. But the problems will start when the hardware can't handle it. The game will run slower. In the worst case the game has some heavy chunks where the game will run really slow, and some chunks where it runs normal. The timing becomes variable which can make your game unplayable.

Fast hardware

The game will have no problems on fast hardware, but you are wasting so many precious clock cycles. Running a game on 25 or 30 FPS when it could easily do 300 FPS… shame on you! You will lose a lot of visual appeal with this, especially with fast moving objects. On the other hand, with mobile devices, this

can be seen as a benefit. Not letting the game constantly run at it's edge could save some battery time.

Conclusion

Making the FPS dependent on a constant game speed is a solution that is quickly implemented and keeps the game code simple. But there are some problems: Defining a high FPS will pose problems on slower hardware, and defining a low FPS will waste visual appeal on fast hardware.

Game Speed dependent on Variable FPS

Implementation

Another implementation of a game loop is to let it run as fast as possible, and let the FPS dictate the game speed. The game is updated with the time difference of the previous frame.

```
DWORD prev_frame_tick;
DWORD curr_frame_tick = GetTickCount();

bool game_is_running = true;
while( game_is_running ) {
    prev_frame_tick = curr_frame_tick;
    curr_frame_tick = GetTickCount();

    update_game( curr_frame_tick - prev_frame_tick );
    display_game();
}
```

The game code becomes a bit more complicated because we now have to consider the time difference in the update_game() function. But still, it's not that hard. At first sight this looks like the ideal solution to our problem. I have seen many smart programmers implement this kind of game loop. Some of them probably wished they would have read this article before they implemented their loop. I will show you in a minute that this loop can have serious problems on both slow and fast (yes, FAST!) hardware.

Slow Hardware

Slow hardware can sometimes cause certain delays at some points, where the game gets "heavy". This can definitely occur with a 3D game, where at a certain time too many polygons get shown. This drop in frame rate will affect the input response time, and therefore also the player's reaction time. The updating of the game will also feel the delay and the game state will be updated in big time-chunks. As a result the reaction time of the player, and also that of the AI, will slow down and can make a simple maneuver fail, or even impossible. For example, an obstacle that could be avoided with a normal FPS, can become impossible to avoid with a slow FPS. A more serious problem with slow hardware is that when using physics, your simulation can even explode!

Fast Hardware

You are probably wondering how the above game loop can go wrong on fast hardware. Unfortunately, it can, and to show you, let me first explain something about math on a computer. The memory space of a float or double value is limited, so some values cannot be represented. For example, 0.1 cannot be represented binary, and therefore is rounded when stored in a double. Let me show you using python:

```
>>> 0.1
0.10000000000000001
```

This itself is not dramatic, but the consequences are. Let's say you have a race-car that has a speed of 0.001 units per millisecond. After 10 seconds your race-car will have traveled a distance of 10.0. If you split this calculation up like a game would do, you have the following function using frames per second as input:

```
>>> def get_distance( fps ):
...     skip_ticks = 1000 / fps
...     total_ticks = 0
...     distance = 0.0
...     speed_per_tick = 0.001
...     while total_ticks < 10000:
...             distance += speed_per_tick * skip_ticks
...             total_ticks += skip_ticks
...     return distance
```

Now we can calculate the distance at 40 frames per second:

```
>>> get_distance( 40 )
10.000000000000075
```

Wait a minute... this is not 10.0??? What happened? Well, because we split up the calculation in 400 additions, a rounding error got big. I wonder what will happen at 100 frames per second...

```
>>> get_distance( 100 )
9.9999999999998312
```

What??? The error is even bigger!! Well, because we have more additions at 100 fps, the rounding error has more chance to get big. So the game will differ when running at 40 or 100 frames per second:

```
>>> get_distance( 40 ) - get_distance( 100 )
2.4336088699783431e-13
```

You might think that this difference is too small to be seen in the game itself. But the real problem will start when you use this incorrect value to do some more calculations. This way a small error can become big, and fuck up your game at high frame rates. Chances of that happening? Big enough to consider it! I have seen a game that used this kind of game loop, and which indeed gave trouble at high frame rates. After the programmer found out that the problem was hiding in the core of the game, only a lot of code rewriting could fix it.

Conclusion

This kind of game loop may seem very good at first sight, but don't be fooled. Both slow and fast hardware can cause serious problems for your game. And besides, the implementation of the game update function is harder than when you use a fixed frame rate, so why use it?

Constant Game Speed with Maximum FPS

Implementation

Our first solution, FPS dependent on Constant Game Speed, has a problem when running on slow hardware. Both the game speed and the framerate will drop in that case. A possible solution for this could be to keep updating the game at that rate, but reduce the rendering framerate. This can be done using following game loop:

```
const int TICKS_PER_SECOND = 50;
const int SKIP_TICKS = 1000 / TICKS_PER_SECOND;
const int MAX_FRAMESKIP = 10;

DWORD next_game_tick = GetTickCount();
int loops;

bool game_is_running = true;
while( game_is_running ) {

    loops = 0;
    while( GetTickCount() > next_game_tick && loops < MAX_FRAMESKIP) {
        update_game();

        next_game_tick += SKIP_TICKS;
        loops++;
    }

    display_game();
}
```

The game will be updated at a steady 50 times per second, and rendering is done as fast as possible. Remark that when rendering is done more than 50 times per second, some subsequent frames will be the same, so actual visual frames will be dispayed at a maximum of 50 frames per second. When running on slow hardware, the framerate can drop until the game update loop will reach MAX_FRAMESKIP. In practice this means that when our render FPS drops below 5 (= FRAMES_PER_SECOND / MAX_FRAMESKIP), the actual game will slow down.

Slow hardware

On slow hardware the frames per second will drop, but the game itself will hopefully run at the normal speed. If the hardware still can't handle this, the game itself will run slower and the framerate will not be smooth at all.

Fast hardware

The game will have no problems on fast hardware, but like the first solution, you are wasting so many precious clock cycles that can be used for a higher framerate. Finding the balance between a fast update rate and being able to run on slow hardware is crucial.

Conclusion

Using a constant game speed with a maximum FPS is a solution that is easy to implement and keeps the game code simple. But there are still some problems: Defining a high FPS can still pose problems on slow hardware (but not as severe as the first solution), and defining a low FPS will waste visual appeal on fast

hardware.

Constant Game Speed independent of Variable FPS

Implementation

Would it be possible to improve the above solution even further to run faster on slow hardware, and be visually more atractive on faster hardware? Well, lucky for us, this is possible. The game state itself doesn't need to be updated 60 times per second. Player input, AI and the updating of the game state have enough with 25 frames per second. So let's try to call the update_game() 25 times per second, no more, no less. The rendering, on the other hand, needs to be as fast as the hardware can handle. But a slow frame rate shouldn't interfere with the updating of the game. The way to achieve this is by using the following game loop:

```
    const int TICKS_PER_SECOND = 25;
    const int SKIP_TICKS = 1000 / TICKS_PER_SECOND;
    const int MAX_FRAMESKIP = 5;

    DWORD next_game_tick = GetTickCount();
    int loops;
    float interpolation;

    bool game_is_running = true;
    while( game_is_running ) {

        loops = 0;
        while( GetTickCount() > next_game_tick && loops < MAX_FRAMESKIP) {
            update_game();

            next_game_tick += SKIP_TICKS;
            loops++;
        }

        interpolation = float( GetTickCount() + SKIP_TICKS - next_game_tick )
                        / float( SKIP_TICKS );
        display_game( interpolation );
    }
```

With this kind of game loop, the implementation of update_game() will stay easy. But unfortunately, the display_game() function gets more complex. You will have to implement a prediction function that takes the interpolation as argument. But don't worry, this isn't hard, it just takes a bit more work. I'll explain below how this interpolation and prediction works, but first let me show you why it is needed.

The Need for Interpolation

The gamestate gets updated 25 times per second, so if you don't use interpolation in your rendering, frames will also be displayed at this speed. Remark that 25 fps isn't as slow as some people think, movies for example run at 24 frames per second. So 25 fps should be enough for a visually pleasing experience, but for fast moving objects, we can still see a improvement when doing more FPS. So what we can do is make fast movements more smooth in between the frames. And this is where interpolation and a prediction function can provide a solution.

Interpolation and Prediction

Like I said the game code runs on it's own frames per second, so when you draw/render your frames, it is possible that it's in between 2 gameticks. Let's say you have just updated your gamestate for the 10Th time, and now you are going to render the scene. This render will be in between the 10Th and 11Th game update. So it is possible that the render is at about 10.3. The 'interpolation' value then holds 0.3. Take this example: I have a car that moves every game tick like this:

```
position = position + speed;
```

If in the 10Th gametick the position is 500, and the speed is 100, then in the 11Th gametick the position will be 600. So where will you place your car when you render it? You could just take the position of the last gametick (in this case 500). But a better way is to predict where the car would be at exact 10.3, and this happens like this:

```
view_position = position + (speed * interpolation)
```

The car will then be rendered at position 530. So basically the interpolation variable contains the value that is in between the previous gametick and the next one (previous = 0.0, next = 1.0). What you have to do then is make a "prediction" function where the car/camera/... would be placed at the render time. You can base this prediction function on the speed of the object, steering or rotation speed. It doesn't need to be complicated because we only use it to smooth things out in between the frames. It is indeed possible that an object gets rendered into another object right before a collision gets detected. But like we have seen before, the game is updated 25 frames per second, and so when this happens, the error is only shown for a fraction of a second, hardly noticeable to the human eye.

Slow Hardware

In most cases, update_game() will take far less time than display_game(). In fact, we can assume that even on slow hardware the update_game() function can run 25 times per second. So our game will handle player input and update the game state without much trouble, even if the game will only display 15 frames per second.

Fast Hardware

On fast hardware, the game will still run at a constant pace of 25 times per second, but the updating of the screen will be way faster than this. The interpolation/prediction method will create the visual appeal that the game is actually running at a high frame rate. The good thing is that you kind of cheat with your FPS. Because you don't update your game state every frame, only the visualization, your game will have a higher FPS than with the second method I described.

Conclusion

Making the game state independent of the FPS seems to be the best implementation for a game loop. However, you will have to implement a prediction function in display_game(), but this isn't that hard to achieve.

Overall Conclusion

A game loop has more to it than you think. We've reviewed 4 possible implementations, and it seems that there is one of them which you should definitely avoid, and that's the one where a variable FPS dictates the game speed. A constant frame rate can be a good and simple solution for mobile devices, but when you want to get everything the hardware has got, best use a game loop where the FPS is completely independent of the game speed, using a prediction function for high framerates. If you don't want to bother with a prediction function, you can work with a maximum frame rate, but finding the right game update rate for both slow and fast hardware can be tricky.

Now go and start coding that fantastic game you are thinking of!

Koen Witters

126 Comments

Older Comments

Mat

@koen Witters:

Are you interested in an Italian translation ?

October 12, 2015 at 03:38 Reply

Gray

Siiiiiiii

Yes, thank you.

May 2, 2016 at 07:23 Reply

uday

Nice post.
One the posts which helped me build my understanding of game loops and the considerations to be careful about.
Built a simple game based on my experience...
https://play.google.com/store/apps/details?id=com.uday.game.matchthrees

October 23, 2015 at 00:18 Reply

frank

Hi Koen!

I want to thank you very much for this article! It helped me a lot wrapping my heat around the topic.
I wonder if you would mind if I were posting a german translation of it on my website. This way your article could make the day of even more people.

October 23, 2015 at 15:45 Reply

neonwarge04

How do you know the value for MAX_FRAMESKIP = 5? This is magic. Do you have reason why this should be 5? Is this tied to TICKS_PER_SECOND? What would MAX_FRAMESKIP be if TICKS_PER_SECOND = 60?

Thank you!

December 16, 2015 at 02:02 Reply

J.A. Rubio

Great article! It does clear a great deal of doubts regarding time management in game loops.

I wonder how much of these applies to small 2D games, in particular the interpolation part. Since 2D animations are pre-made, I can't see the benefits of interpolation.

December 16, 2015 at 03:17 Reply

Falconerd

The interpolation described is for positions, I believe. Though, I'm sure you could figure out a way for animations to handle interpolation as well using some kind programmatic animations with sprite sheets.

January 6, 2016 at 04:08 Reply

Farid

Thanks for the great article but here is the two things that concerns me,
1. Frame rates more that 60 fps cant be handled by monitor so why we allow the loop to render as fast as possible?
2. there is no sleep() in your solution, doesn't this cause high usage of CPU?

January 14, 2016 at 08:01 Reply

Loolz

I tried to implement the last method (the one involving interpolation) of game loop but for some reason when I used a counter variable to actually get the amount of the ticks happening in a second it gave me a strange result. I get 25 and 26 alternating. So for the first second it might be 25 and then it will be 26 and then 25 and so on. Shouldn't it always be 25?

May 29, 2016 at 18:57 Reply

Wyatt Baldree

Just wanted to say thank you so much for your explanation of the game loops! This is by far the most clear explanation of interpolation that I have read.

June 4, 2016 at 19:34 Reply

Matan

Hi!

I realize this is a seven years old post, but I'll comment anyway…

The second line of the first game loop reads

const int SKIP_TICKS = 1000 / FRAMES_PER_SECOND;

Other loops have a similar line.

In an explanatory article I find this vague. Why are the SKIP_TICKS computed this way? Why 1000? Isn't the dimensional units of SKIP_TICKS ticks?

I think the line might be clearer if it were

const int TICKS_PER_SECOND = 1000;
const int SECONDS_PER_FRAME = 1.0 / FRAMES_PER_SECOND;
const int SKIP_TICKS = TICKS_PER_SECOND * SECONDS_PER_FRAME;

This explains where the 1000 comes from, and it "encodes" the dimensional unit analysis in the names, so we see the in the last line that the dimensional units of SKIP_TICKS is ticks per frame. Replacing division with multiplication in the computing line makes the dimensional unit analysis easier to work out, I think.

Granted, it is a few lines more, but if the purpose is to explain the idea, I think this serves well.

Thanks for the great post! I love it!

July 7, 2016 at 09:09 Reply

Jonas

Oh my gosh, this is just fantastic!
I thought much about how I can do the best gameloop and your last game loop with interpolation is a really great idea – THX dude^-^

PS: In your last slow hardware section didn't you mean 25FPS instead of 15FPS or do I overlook something?

January 7, 2017 at 05:45 Reply

Koen Witters

I used to to mobile development on PocketPC, before those platforms became really powerful. On those things, we got away with 15 FPS and it still looked good.

January 8, 2017 at 05:11 Reply

Andres

Hey,

Great post, it was really informative. Where do you think the sprite animations should live within this sort of interpolated gameloop? It seems to me like it fits within the draw method as only the draw knows of the delta time which would then drive the animation frame ticks. It is a bit odd, though, because it is an "update" within the draw method.

February 21, 2017 at 22:53 Reply

Jason

Super limited experience, but a normal sprite animation wouldn't be interpolated, just the position of the object. There are spline animations where the movement is defined using a series of bones and such that can be interpolated between positions, though.

Anyway, for standard animations, you just have an animation speed and the update function switches which image a sprite animation is on. The render function just blits whichever image is the current one (possibly skipping or re-rendering). It's important that the update function handle the actual animation instead of the rendering function because different parts of animations can change how the object behaves, eg, it can grow if a weapon extends forward, changing its collision box, or in certain frames it's only vulnerable from certain sides. This only really matters when the FPS is lower than the gamerate divided by the animation speed (slower animations are less likely to skip, etc), so this might be purely academic. But interpolation is impossible without intimate knowledge of each

animation and, at that point, you can “just” add more frames to every animation, but any more than the gamerate would be difficult to keep track of because it’d be separate from the update.

Again, super limited experience (read: zero), but that’s how it works in my head.

March 17, 2017 at 21:52 Reply

Ivan

Koen Witters , Hello! I am interesting in this article. So what about game loops when we have multicores CPU? It seem to use threads. Is’n it? Maybe it will be better?

December 3, 2017 at 10:38 Reply

Ivan

Update:
I think that if we have even 2 cores we can do update() in 1st and draw() in 2nd core.

December 3, 2017 at 10:52 Reply

Rafbeam

I made a program to calculate update and display rate:

```
#include
#include
#include

const int TICKS_PER_SECOND = 25;
const int SKIP_TICKS = 1000 / TICKS_PER_SECOND; //CLOCK_TICKS_PER_SECOND / TICKS_PER_SECOND
const int MAX_FRAMESKIP = 5; //sqrt(TICKS_PER_SECOND);

int main()
{
clock_t timer = clock();
clock_t dT = clock();
clock_t uT = clock();
int skip = 0;
while(true)
{
skip = 0;
while(clock() > timer && skip < MAX_FRAMESKIP)
{
//UPDATE
timer += SKIP_TICKS;
```

```
skip++;
std::cout<<"Update rate: "<<clock() – uT<<"ms PER SEC: "<<1000/(clock() – (double)uT)<<"\n";
uT = clock();
}
//DISPLAY
if(GetAsyncKeyState(VK_SPACE))system("pause");
std::cout<<"Display rate: "<<clock() – dT<<"ms FPS: "<<1000/(clock() – (double)dT)<<"\n";
dT = clock();
}
return 0;
}
```

Is it good? Update rate is not constant, it's alternating beetwen 47 and 31, with a slight advantage of 47. So real tick rate is (47+31)/2 = 39 1000/39 = 25.5, so with this adventage of 47 it will be 25 tps.

Should it work like that? Should tick rate be constant? I think it shouldn't, because we can't "half-update" game state or "half-draw" single frame.

December 14, 2017 at 01:24 Reply

Rafbeam

I made a program to calculate update and display rate:

```
#include
#include
#include

const int TICKS_PER_SECOND = 25;
const int SKIP_TICKS = 1000 / TICKS_PER_SECOND; //CLOCK_TICKS_PER_SECOND / TICKS_PER_SECOND
const int MAX_FRAMESKIP = 5; // ?sqrt(TICKS_PER_SECOND);? ?FOR SAFETY?

int main()
{
clock_t timer = clock();
clock_t dT = clock();
clock_t uT = clock();
int skip = 0;
while(true)
{
skip = 0;
while(clock() > timer && skip < MAX_FRAMESKIP)
{
//UPDATE
timer += SKIP_TICKS;
skip++;
std::cout<<"Update rate: "<<clock() – uT<<"ms PER SEC: "<<1000/(clock() – (double)uT)<<"\n";
uT = clock();
}
//DISPLAY
if(GetAsyncKeyState(VK_SPACE))system("pause");
```

```
std::cout<<"Display rate: "<<clock() – dT<<"ms FPS: "<<1000/(clock() – (double)dT)<<"\n";
dT = clock();
}
return 0;
}
```

Is it good? Update rate is not constant, it's alternating beetwen 47 and 31, with a slight advantage of 47. So real tick rate is (47+31)/2 = 39 1000/39 = 25.5, so with this adventage of 47 it will be 25 tps.

Should it work like that? Should tick rate be constant? I think it shouldn't, because we can't "half-update" game state or "half-draw" single frame.

December 14, 2017 at 01:24 Reply

Rafbeam

Okay! So MAX_FRAMESKIP is 20% of TICKS_PER_SECOND! And it’s preventing program from stopping drawing!

December 14, 2017 at 11:45 Reply

Luhan M.

I know this is an old post, but could you answer me what is the actual difference of a game slowing down and the drop of frame rate?

February 5, 2018 at 07:09 Reply

Koen Witters

Frame rate is just the number of frames per second shown on the screen. So dropping from 60 frames to 30 frames can mean a car in a racing game drives as fast as before.

On the other hand, if the game slows down without the frame rate dropping, this means the car drives half as fast as normal, but the screen is still updated at 60 frames per second.

February 5, 2018 at 09:12 Reply

Luhan M.

Sorry if I misunderstood you, but, how the rendering keep 60 FPS while the game can’t keep up with update()? Because as I understood, if it can’t catch up with the actual frame, it will slow down, wouldn’t that affect the rendering fps too?

February 5, 2018 at 15:31 Reply

Barnack

Why not just leave the amount of steps per second as an option to the user for the “FPS dependent on Constant Game Speed” implementation? If the user has a faster pc he is free to set the game speed to 200 steps per second 😉

June 9, 2018 at 08:55 Reply

Leave a Reply

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

Comment

Name*

Email*

Website

Post Comment

Sign me up for the newsletter