



GP Part I Extras: Application Timing

Frame Rate Timers

Faster computers can process more game loop iterations per second than slower computers. So Pentium IV™ systems will process more frames per second than Pentium III™ systems, which will in turn process more than Pentium II™. The GeForce™ 4 can finish drawing a frame faster than the GeForce™ III, which is faster than the GeForce™ II, and so on. Such variations in degree of hardware speed make timing game updates based on the frame rate (frames per second) an unwise choice. Certain games from the late eighties did not account for this and as a result, cannot be run on modern PCs. On a modern system, these games cannot be properly controlled since object velocities are simply too high given the rapid frame updates. Our preference is to use more scalable time units like seconds or some fraction thereof.

From a coding perspective, maintaining a game timer is quite simple. For example, most games will include timers that track the amount of time that has elapsed since the application started and how much time has elapsed since the last frame. These are both useful pieces of information to keep on hand for tasks like game physics, AI updates, animation, etc. Adding such a timer to the main application loop requires only a few lines of code. We will look at some of this code shortly.

Selecting a System Timer

In Windows, there are a number of timing mechanisms at our disposal. One of the easiest to use is the Windows scheduled event timer. We simply tell the system to send us WM_TIMER message at specified intervals and then process those messages in our message pump. However, the event based timer has poor performance and low resolution. In addition to the overhead of message handling, the WM_TIMER message is actually very low down on the list of Windows priorities. It actually has a lower priority than nearly all other Windows messages and is therefore unreliable.

If we set the timer to pulse 20 times per second, there is no guarantee that it actually will. If a WM_TIMER message is in the message loop waiting to be processed and another message comes along with a higher priority, the WM_TIMER message gets pushed to the back of the message queue. This inconsistency limits the usefulness of WM_TIMER for our purposes.

Windows provides two reliable timers that give good performance and suit all of our needs. Both are declared in the 'mmsystem.h', so this file should be included in your application if you want to use these timers. You must also link your application to the library winmm.lib.

timeGetTime()

This function is compatible with almost all PC's and provides an accuracy of between 10 and 50 milliseconds (1/1,000th of a second) depending on the target operating system. It takes no parameters and returns the current system time as a DWORD value. While this value by itself is not very meaningful, we can use it to tell how much time has passed since the last call to the function.

In other words, we can call `timeGetTime()`, store the value it returns and then on the next frame, call it again and subtract the first value from the second.



The result is the number of milliseconds that have passed since the last frame. This is a good timer; it is reliable and relatively fast, and is also available on very old PC's. Unfortunately, due to its relatively low resolution on certain operating systems (Windows NT, 2000 and XP) it is certainly a less than perfect solution.

The PerformanceCounter

The Performance Counter is a hardware counter that runs at 3.19 megahertz (MHz). This gives us a timer with measurement times in microseconds ($1/1,000,000^{\text{th}}$ of a second). Depending on your needs, this level of resolution may or may not be necessary. Our demo applications will favor the extra resolution.

Note: Earlier computers did not have Hardware Timers, so the previously discussed function `timeGetTime()` can be used to fall back on if a hardware timer is not present. All currently produced processors have hardware timers, so this is really not an issue if your game is going to be cutting edge, in which case it would not support running on a 386 or 486 anyway.

Two functions are available for using the Performance Counter. The first function we have to call is `QueryPerformanceFrequency`:

```
BOOL QueryPerformanceFrequency (LONGLONG *lpFrequency);
```

The single parameter is a pointer to a 64-bit integer that will receive the frequency of the timer. It will be measured in *ticks* per second. If this function returns a non-zero value, then a timer is available and `lpFrequency` will hold the timer's frequency. If this function returns 0, then it was unable to find a Hardware Timer on the current system. This function only needs to be called once, during our initialization routine.

Retrieving the timer frequency, even when we know that it runs at 3.19 MHz, makes our game future-proof. Should new hardware become available with a higher resolution timer, our game will still work as expected because our calculations will be relative to the timer's frequency.

`QueryPerformanceCounter` is the second function we will call -- once for every iteration of our game loop:

```
BOOL QueryPerformanceCounter (LONGLONG * lpTime);
```

This function also takes a pointer to a 64-bit integer as its only parameter and fills it with the current time. Just like the `timeGetTime()` function, this time is measured in ticks per second from some arbitrary starting point. Once again, while the raw result is not particularly meaningful to us, we can use it to determine how many ticks have passed since the last time the function was called. This value can then be converted into fractions of a second.

The code snippet below queries the frequency of the clock and sets up a variable called `Time_Scale`. This code would be executed at application startup before we enter our main game loop. Later on in our game loop, the `Time_Scale` variable is multiplied by the elapsed time to

convert the elapsed time into fractions of a second. Note that if the hardware counter is not available, we use timeGetTime instead.



```
LONGLONG Frequency, Current_Time , Last_Time;

double Time_Elapsed , Time_Scale;

BOOL Counter_Available=false;

...
...

if(QueryPerformanceFrequency((LARGE_INTEGER *)&Frequency)
{
    Counter_Available = true;
    Time_Scale = 1.0 / Frequency;

    QueryPerformanceCounter((LARGE_INTEGER *)&Last_Time);
}
else
{
    Last_Time=timeGetTime();
    Time_Scale=0.001;
}
```

So if the Counter is available, then the Counter_Available variable is set to true and our game loop will know that it can use it. We divide 1.0 by the frequency of the timer that we recorded earlier to give us a multiplier that will later convert our elapsed time into seconds. We also store the current time in the Last_Time variable. This is so we have something to compare against our first time through the game loop. The game loop would have code in it that looks something like this:

```
if (Counter_Available)
    QueryPerformanceCounter((LARGE_INTEGER *) &Current_Time);
else
    Current_Time=timeGetTime();

Time_Elapsed = (Current_Time - Last_Time) * Time_Scale;

Last_Time = Current_Time;
```

That is all we have to do really. Time_Elapsed now stores the amount of time that has passed, in seconds. For example, a value of 0.1 equals 1/10th of a second.

We can now use Time_Elapsed in movement and rotation calculations. Assuming we are storing how much an object can move or rotate in units per second, we simply multiply these values by Time_Elapsed, (remember that this is really a division operation because Elapsed_Time is measured in Hertz -- 1/sec) Our code would look something like this:

```
Ships_Speed      = 10;    // ship travels 10 units in a second
Ships_TurnAngle  = 1.2;   // ship can turn 1.2 radians in a second
```



```
...
...

Game Loop
//First calculate the time that has elapsed since the last game loop
iteration
if (Counter_Available)
    QueryPerformanceCounter((LARGE_INTEGER *) &Current_Time);
else
    Current_Time=timeGetTime();

Time_Elapsed = (Current_Time - Last_Time) * Time_Scale;
Last_Time    = Current_Time;

//Now use the elapsed time to scale the position and orientation
//updates of our game objects
Ships_Position = Ships_Position + (Ships_Speed*Time_Elapsed);
Ships_Angle    = Ships_Angle + (Ships_TurnAngle*TimeElapsed);
```

The CTimer Class

We have implemented a simple timer class to wrap this functionality. After all, this really is an object that we will be re-using time and time again, and is thus ideally suited for being encapsulated in a self-contained class.

We will also implement some simple functionality to help smooth sporadic changes in elapsed time values. Should the frame rate take a sudden drop for some reason, perhaps due to an external process hogging CPU time for a fraction of a second, or perhaps due to the graphic complexity of the scene increasing dramatically for a brief period, we would like to avoid jerky results in our animations and scene updates. In this class, we will always keep an array of the last 50 elapsed times that have been fetched over the last 50 iterations of our game loop. The elapsed time returned by the class to our application will actually be the average of these times. Any temporary fluctuations in frame rate will be diluted by the other 49 values in the list. This also helps us more accurately describe a *consistent* application frame rate. You can experiment with this

In your demo projects, you will find the CTimer class declaration in the file 'CTimer.h'. It is shown below with a list of its member variables and methods. In addition to the constructor and the destructor, the class exposes only three public functions -- the interface for our class. The 'Tick' function should be called in every iteration of the game loop. It fetches the current time from the performance counter, subtracts this from the previous time (fetched in a previous call to this function) to calculate the elapsed time in seconds since the last time the function was called. We then use the GetTimeElapsed method to fetch this elapsed time value. This value is used to control the updating of the objects in our scene. This class also has a helpful function called GetFrameRate which will use the elapsed time value to calculate and return how many frames per second the



application is currently achieving. For convenience, passing in a string allows us to get back a line of text describing the frames per second currently being achieved, so that the application can output this string to the user.

```
class CTimer
{
public:
    //-----
    // Constructors & Destructors for This Class
    //-----
    CTimer();
    virtual ~CTimer();

    //-----
    // Public Functions For This Class
    //-----
    void          Tick( float fLockFPS = 0.0f );
    float         GetTimeElapsed() const;
    unsigned long GetFrameRate( LPTSTR lpszString = NULL ) const;

private:
    //-----
    // Private Variables For This Class
    //-----

    bool          m_PerfHardware;
    float         m_TimeScale;
    float         m_TimeElapsed;
    __int64       m_CurrentTime;
    __int64       m_LastTime;
    __int64       m_PerfFreq;

    float         m_FrameTime[MAX_SAMPLE_COUNT];
    ULONG         m_SampleCount;

    unsigned long m_FrameRate;
    unsigned long m_FPSFrameCount;
    float         m_FPSTimeElapsed;
};
```

Let us now take a look at the member variables and describe their purpose.

bool m_PerfHardware

This Boolean variable is set to either true or false in the constructor of the timer object depending on whether a hardware performance counter is available on the current system or not. If this is set to TRUE, then the Tick function will use QueryPerformanceCounter to fetch the elapsed time. If set to FALSE, the Tick function will use the lower resolution timeGetTime function as a fall back option.

float m_TimeScale

This value is used to convert the result of time query functions into seconds. It is calculated in the constructor by dividing 1 by the frequency of the timer.



**float m_TimeElapsed**

This member stores the current time in seconds that has elapsed since the previous call to the Tick function. The setting of this variable is the fundamental purpose of the Tick function. Once the Tick function has executed, this value can be fetched by the application using the GetTimeElapsed method. It can be used directly for scene updates and task execution.

__int64 m_CurrentTime

This member contains the raw value (time) returned by the QueryPerformanceCounter function. This value will later have the previous time subtracted from it and will be scaled by the m_TimeScale for conversion into seconds of elapsed time (since the previous call to the Tick function).

__int64 m_LastTime

At the end of the Tick function, this variable will be assigned the value in m_CurrentTime. The next time the Tick function is called, it will be subtracted from the new m_CurrentTime to determine the elapsed time between calls.

__int64 m_PerfFreq

This value will be assigned in the constructor when the timer is first created. It will contain the frequency of the timer that we are using. This is needed to calculate m_TimeScale so that we can convert the elapsed time into seconds. It is retrieved from the system using the QueryPerformanceFrequency function.

float m_FrameTime[MAX_SAMPLE_COUNT]

This float array will be used to store MAX_SAMPLE_COUNT elapsed time values (in seconds) that we calculated in the last MAX_SAMPLE_COUNT calls to the Tick function. MAX_SAMPLE_COUNT is defined as 50 in our demos, but you can change this. At the end of the Tick function, the m_TimeElapsed value will be set to the average of these 50 values. This should help smooth any sporadic fluctuations in frame rate that occurred in the previous 50 iterations of our game loop.

unsigned long m_SampleCount;

This variable stores the current number of time samples stored in the m_FrameTime array. Whilst you might think this would always be equal to 50 (MAX_SAMPLE_COUNT), this will only be true once the first 50 iterations of our game loop have been executed. When our application is in its 10th iteration for example, there will be only 10 actual time samples in the m_FrameTime array.

unsigned long m_Framerate;

This value is also calculated in the Tick function and stores how many frames per second the application is currently achieving. Whilst this is not to be used to update the objects in our scene in any way, it can be useful for diagnostic purposes.

unsigned long m_FPSFrameCount;

This variable is used in the Tick function to keep track of how many times the function has been called in any given second. It is incremented each time the Tick function is called. Once the end of the current second has been reached, m_FPSFrameCount will contain the total number of frames that occurred in that second. It can then be stored in the m_Framerate variable.



float m_FPSTimeElapsed;

This member is used in connection with the previous two values to record how much time has elapsed since the start of the current second we are processing. Once this time has reached 1.0, the second has ended and we can check the m_FPSFrameCount variable to see how many frames were achieved in this second. The result is stored in m_FrameRate so that it can be retrieved by the application for diagnostic purposes.

The Constructor

This constructor initializes the member variables to their default values and determines whether or not the high performance counter is available. We call QueryPerformanceFrequency to fetch the frequency of the high performance counter on the current system. If the function returns non-zero then a performance counter is available and its frequency is stored in the m_PerfFreq member variable and we set m_PerfHardware to TRUE. We then query the current time of the performance counter and store it in m_LastTime for use in the first call to Tick. Finally, we calculate m_TimeScale by dividing 1.0 by the frequency of the performance counter.

```
CTimer::CTimer()
{
    // Query performance hardware and setup time scaling values
    if (QueryPerformanceFrequency((LARGE_INTEGER *)&m_PerfFreq))
    {
        m_PerfHardware          = TRUE;
        QueryPerformanceCounter((LARGE_INTEGER *) &m_LastTime);
        m_TimeScale              = 1.0f / m_PerfFreq;
    }
    else
    {
        // no performance counter, read in using timeGetTime
        m_PerfHardware          = FALSE;
        m_LastTime              = timeGetTime();
        m_TimeScale              = 0.001f;
    } // End If No Hardware

    // Clear any needed values
    m_SampleCount               = 0;
    m_FrameRate                 = 0;
    m_FPSFrameCount             = 0;
    m_FPSTimeElapsed            = 0.0f;
}
```

If the performance counter is not available on the current system then we will set the m_PerfHardware flag to FALSE. The Tick function will then know to use timeGetTime instead of the higher resolution QueryPerformanceCounter alternative. We then query the current time and store it in m_LastTime and set the m_TimeScale variable to 0.001 (for millisecond resolution, compatible with the timeGetTime return value).

The Destructor

As this class does not allocate any dynamic memory, the destructor does not have any work to do.

```
CTimer::~~CTimer() {}
```




The Tick Function

The Tick method is called any time we would like the timer updated. In our demos this is done once per game loop iteration, before we update the scene. Here the current time is updated and the elapsed time calculated from the previous iteration.

The function takes a single float parameter which serves as an optional frame rate lock. This can be useful if you want to see how your game performs at different frame rates. Our demo applications will not use this feature, and will prefer to squeeze out as many frames as possible. The default value for the fLockFPS parameter (0.0) allows the application to proceed at the maximum speed.

First the function fetches the current time using either QueryPerformanceCounter or timeGetTime, depending on whether the high performance counter is available. Once we have the current time of the counter, we subtract from this the time that was fetched from the counter in the previous call to the function (or that was set in the constructor if this is the first time the Tick function has been called) to calculate the elapsed time between calls. These steps are shown below:

```
void CTimer::Tick( float fLockFPS )
{
    float fTimeElapsed;

    // Is performance hardware available?
    if ( m_PerfHardware )
    {
        // Query high-resolution performance hardware
        QueryPerformanceCounter((LARGE_INTEGER *)&m_CurrentTime);
    }
    else
    {
        // Fall back to less accurate timer
        m_CurrentTime = timeGetTime();
    } // End If no hardware available

    // Calculate elapsed time in seconds
    fTimeElapsed = (m_CurrentTime - m_LastTime) * m_TimeScale;
```

The next section of code is activated only if we chose to lock the frame rate to a given number of frames per second. Here we do a brute force while loop to chew up time, but a sleep function could also be used. The loop repeatedly queries the time of the counter and calculates the elapsed time in seconds. Only when the elapsed time reaches the user-specified frame rate ceiling is the while loop exited. If the caller requested 30 frames per second, then we only need to update our timer every $1.0/30 = 0.033333$ seconds. Therefore, if the elapsed time between this frame and the last frame is say, 0.011111, then this while loop would chew up time until fTimeElapsed was greater than or equal to 0.033333.

```
    // Should we lock the frame rate ?
    if ( fLockFPS > 0.0f )
    {
        while ( fTimeElapsed < (1.0f / fLockFPS))
        {
```



```
// Is performance hardware available?
if ( m_PerfHardware )
{
    // Query high-resolution performance hardware
    QueryPerformanceCounter((LARGE_INTEGER*)&m_CurrentTime);
}
else
{
    // Fall back to less accurate timer
    m_CurrentTime = timeGetTime();
} // End If no hardware available

// Calculate elapsed time in seconds
fTimeElapsed = (m_CurrentTime - m_LastTime) * m_TimeScale;

} // End While
} // End If
```

Our next step is to store the current time of the timer object (calculated in the last call to this function) in the `m_LastTime` member. The next time this function is called it can be subtracted from the current time (at the start of the function) to determine the difference in the timer's time between calls to the function.

```
// Save current frame time
m_LastTime = m_CurrentTime;
```

The next section of code is where we add our elapsed time to an array of the 50 most recent time values that have been calculated (in previous calls to the function). We only add our elapsed time if it is not a wild time (such as over a whole second for example) which is more than likely being caused by a menu bar being selected or something like that. So, provided the elapsed time is a sensible one, we insert it at the head of our frame time array by moving the first 49 elements in the array [0] to [48] into elements [1] to [49] (nudging them up one). The oldest time will be bumped off the end of the array, leaving room at the head of our array (element [0]) to place our new time.

We also increment the sample count because if this function has not been called at least 50 times, the sample count array will not be full yet. We need to know how many are stored there to perform the averaging of times shown in a moment.

```
// Filter out values wildly different from current average
if ( fabsf(fTimeElapsed - m_TimeElapsed) < 1.0f )
{
    // Wrap FIFO frame time buffer.
    memmove(&m_FrameTime[1],m_FrameTime, (MAX_SAMPLE_COUNT-1)*sizeof(float));
    m_FrameTime[ 0 ] = fTimeElapsed;

    if ( m_SampleCount < MAX_SAMPLE_COUNT ) m_SampleCount++;
} // End if
```

The next thing our timer class does is calculate the frame rate by calculating when a second



boundary has been reached. It records the number of times this function was called within that second. Whilst we do not need to know the frame rate as such to move our objects correctly, this is useful diagnostic information that the application may want to display to the user.

`m_FPSFrameCount` starts at 0 initially and is incremented each time the function is called. `m_FPSTimeElapsed` also starts at 0 and has the elapsed time added to it with each call to this function. When `m_FPSTimeElapsed` reaches a value ≥ 1.0 , it means one second has passed and `m_FPSFrameCount` will contain the total number of times this function was executed in the previous second. When this is the case, we record the frame count in the `m_FrameRate` member variable and reset `m_FPSFrameCount` and `m_FPSTimeElapsed` to zero so that we can start recording the frame rate all over again for the next second of time.

```
// Calculate Frame Rate
m_FPSFrameCount++;
m_FPSTimeElapsed += m_TimeElapsed;
if ( m_FPSTimeElapsed > 1.0f)
{
    m_FrameRate      = m_FPSFrameCount;
    m_FPSFrameCount   = 0;
    m_FPSTimeElapsed = 0.0f;
} // End If Second Elapsed
```

Our next important job is to calculate the elapsed time in seconds. Since the application will use this value to update objects, we would like it to be smooth. This is where our time sample array comes into play. The following code sums the elements in the `m_FrameTime` array and averages the result using the number of elements in the array. The result is stored in the `m_TimeElapsed` member variable.

```
// Count up the new average elapsed time
m_TimeElapsed = 0.0f;
for ( ULONG i = 0; i < m_SampleCount; i++ )
    m_TimeElapsed += m_FrameTime[ i ];

if ( m_SampleCount > 0 ) m_TimeElapsed /= m_SampleCount;
}
```

At this point the internal variables of the `CTimer` object have been updated and both the elapsed time and the frame rate values are available to the application through the two member functions described next.

The GetTimeElapsed Function

This function should be called after the `Tick` function has been called for each iteration of the game loop. It simply returns the averaged elapsed time that was calculated and stored in the `Tick` function. This value will be used by our demo applications to scale movements and rotations of objects in the scene.

```
float CTimer::GetTimeElapsed() const
{
    return m_TimeElapsed;
}
```



The GetFrameRate Function

This function returns the current frame rate and optionally fills a formatted string with this information for output. The ‘_itot’ function is used to place the integer frame rate value into the string in base 10 format. The letters FPS are pre-pended for strings that look like “25 FPS”.

```
unsigned long CTimer::GetFrameRate( LPTSTR lpszString ) const
{
    // Fill string buffer ?
    if ( lpszString )
    {
        // Copy frame rate value into string
        _itot( m_FrameRate, lpszString, 10 );

        // Append with FPS
        strcat( lpszString, _T(" FPS") );

    } // End if build FPS string

    return m_FrameRate;
}
```

Conclusion

This simple timer system should serve us well for the remainder of this course. Feel free to use it however you see fit -- you can probably think of ways to streamline it a bit or tweak it for your own purposes. There is much more to learn about game timing as we move on in this series. For example, task scheduling and time-slice management are very interesting related subjects and we will examine these later in the curriculum. For now however, you should be in good shape with this nice little reusable class.