Tutorial 15: FPS, CPU Usage, and Timers

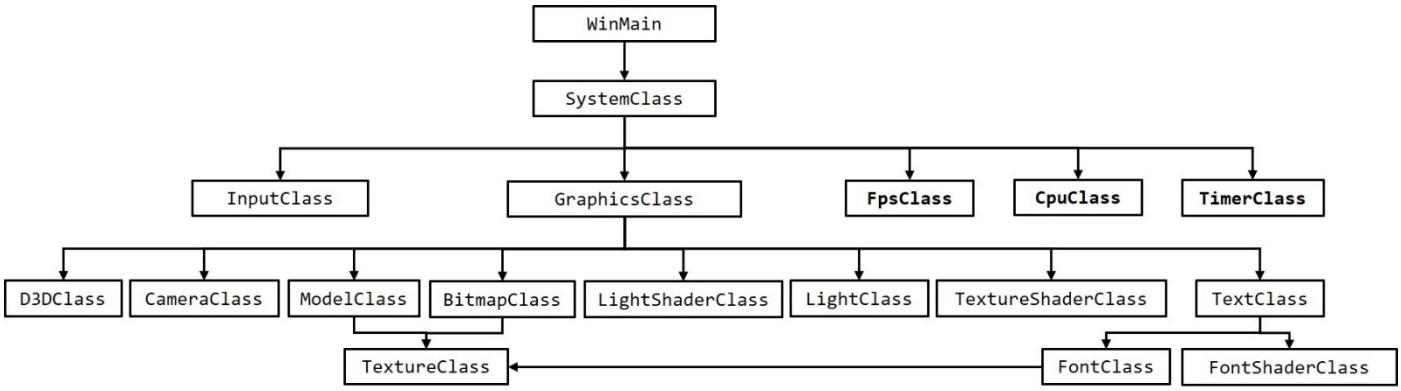
This tutorial will introduce three new classes that will encapsulate the functionality of a frames per second (FPS) counter, a CPU usage counter, and a high precision timer.

The first new class is the FpsClass. The FpsClass will handle recording the frames per second that the application is running at. Knowing how many frames are rendered every second gives us a good metric for measuring our application's performance. This is one of the industry standard metrics that is used to determine acceptable graphics render speed. It is also useful when implementing new features to see how they impact the frame speed. If the new feature cuts the frame speed in half then you can immediately realize you have a major problem by using just this simple counter. Keep in mind that the current standard fps speed for computers is 60 fps. Anything below 60 fps is considered to be performing poorly, and anything below 30 is very noticeable to the human eye. A general rule when coding is keep your fps maximized and if a properly implemented new feature makes a serious dent in that speed then it needs to be justified and at a minimum taken note of.

The second new class is the CpuClass. This will handle recording the cpu usage so that we can display the current percentage of CPU use to the screen. Knowing the CPU usage can be useful for debugging new changes to the code similar to how fps is used. It provides a simple and immediate metric for identifying recently implemented poor code or algorithms.

The final new class is the TimerClass. This is a high precision timer we can use for timing events and ensuring our application and its various components all synchronize to a common time frame.

Framework



We will start the tutorial by examining the three new classes.

Fpsclass.h

The FpsClass is simply a counter with a timer associated with it. It counts how many frames occur in a one second period and constantly updates that count.

```
////////////////////////////////////////////////////////////////////////////////
// Filename: fpsclass.h
////////////////////////////////////////////////////////////////////////////////
#ifndef _FPSCLASS_H_
#define _FPSCLASS_H_

//////////////
// LINKING //
//////////////
#pragma comment(lib, "winmm.lib")

//////////////
// INCLUDES //
//////////////
#include <windows.h>
#include <mmsystem.h>

////////////////////////////////////////////////////////////////////////////////
```

```
// Class name: FpsClass
////////////////////////////////////////////////////////////////////////////
class FpsClass
{
public:
	FpsClass();
	FpsClass(const FpsClass&);
	~FpsClass();

	void Initialize();
	void Frame();
	int GetFps();

private:
	int m_fps, m_count;
	unsigned long m_startTime;
};

#endif
```

Fpsclass.cpp

```
////////////////////////////////////////////////////////////////////////////
// Filename: fpsclass.cpp
////////////////////////////////////////////////////////////////////////////
#include "fpsclass.h"


FpsClass::FpsClass()
{
}

FpsClass::FpsClass(const FpsClass& other)
{
}

FpsClass::~FpsClass()
{
}
```

The Initialize function sets all the counters to zero and starts the timer.

```
void FpsClass::Initialize()
{
	m_fps = 0;
	m_count = 0;
	m_startTime = timeGetTime();
	return;
}
```

The Frame function must be called each frame so that it can increment the frame count by 1. If it finds that one second has elapsed then it will store the frame count in the m_fps variable. It then resets the count and starts the timer again.

```
void FpsClass::Frame()
{
	m_count++;

	if(timeGetTime() >= (m_startTime + 1000))
	{
		m_fps = m_count;
		m_count = 0;

		m_startTime = timeGetTime();
	}
```

```
}
```

GetFps returns the frame per second speed for the last second that just passed. This function should be constantly queried so the latest fps can be displayed to the screen.

```
int FpsClass::GetFps()
{
	return m_fps;
}
```

Cpuclass.h

The CpuClass is used to determine the percentage of total cpu use that is occurring each second.

```
////////////////////////////////////////////////////////////////////////////////
// Filename: cpuclass.h
////////////////////////////////////////////////////////////////////////////////
#ifndef _CPUCLASS_H_
#define _CPUCLASS_H_
```

We use the pdh library to query the cpu usage.

```
//////////////
// LINKING //
//////////////
#pragma comment(lib, "pdh.lib")

//////////////
// INCLUDES //
//////////////
#include <pdh.h>

////////////////////////////////////////////////////////////////////////////////
// Class name: CpuClass
////////////////////////////////////////////////////////////////////////////////
class CpuClass
{
public:
	CpuClass();
	CpuClass(const CpuClass&);
	~CpuClass();

	void Initialize();
	void Shutdown();
	void Frame();
	int GetCpuPercentage();

private:
	bool m_canReadCpu;
	HQUERY m_queryHandle;
	HCOUNTER m_counterHandle;
	unsigned long m_lastSampleTime;
	long m_cpuUsage;
};

#endif
```

Cpuclass.cpp

```
////////////////////////////////////////////////////////////////////////////////
// Filename: cpuclass.cpp
////////////////////////////////////////////////////////////////////////////////
#include "cpuclass.h"
```

```
CpuClass::CpuClass()
{
}


CpuClass::CpuClass(const CpuClass& other)
{
}

CpuClass::~CpuClass()
{
}
```

The Initialize function will setup the handle for querying the cpu on its usage. The query setup here will combine the usage of all the cpus in the system and gives us back a total instead of each individual cpu's usage. If it can't get a query handle or poll the cpu usage for whatever reason it will set the m_canReadCpu flag to false and just keep the cpu usage at zero percent. Some cpus and operating systems privilege levels can cause this to fail. We also start the timer so we only sample the cpu usage once a second.

```
void CpuClass::Initialize()
{
	PDH_STATUS status;

	// Initialize the flag indicating whether this object can read the system cpu usage or not.
	m_canReadCpu = true;

	// Create a query object to poll cpu usage.
	status = PdhOpenQuery(NULL, 0, &m_queryHandle);
	if(status != ERROR_SUCCESS)
	{
		m_canReadCpu = false;
	}

	// Set query object to poll all cpus in the system.
	status = PdhAddCounter(m_queryHandle, TEXT("\\Processor(_Total)\\% processor time"), 0,
	&m_counterHandle);
	if(status != ERROR_SUCCESS)
	{
		m_canReadCpu = false;
	}

	m_lastSampleTime = GetTickCount();

	m_cpuUsage = 0;

	return;
}
```

The Shutdown function releases the handle we used to query the cpu usage.

```
void CpuClass::Shutdown()
{
	if(m_canReadCpu)
	{
		PdhCloseQuery(m_queryHandle);
	}

	return;
}
```

Just like the FpsClass we have to call the Frame function each frame. But to reduce the amount of querying we use a m_lastSampleTime variable to ensure we only sample once a second. So each second we ask the cpu for its usage and save that value in m_cpuUsage. More than this is not necessary.

```
void CpuClass::Frame()
```

```
{
	PDH_FMT_COUNTERVALUE value;

	if(m_canReadCpu)
	{
		if((m_lastSampleTime + 1000) < GetTickCount())
		{
			m_lastSampleTime = GetTickCount();

			PdhCollectQueryData(m_queryHandle);

			PdhGetFormattedCounterValue(m_counterHandle, PDH_FMT_LONG, NULL, &value);

			m_cpuUsage = value.longValue;
		}
	}

	return;
}
```

The GetCpuPercentage function returns the value of the current cpu usage to any calling function. Once again if it couldn't read the cpu for whatever reason we just set the usage to zero.

```
int CpuClass::GetCpuPercentage()
{
	int usage;

	if(m_canReadCpu)
	{
		usage = (int)m_cpuUsage;
	}
	else
	{
		usage = 0;
	}

	return usage;
}
```

Timerclass.h

The TimerClass is a high precision timer that measures the exact time between frames of execution. Its primary use is for synchronizing objects that require a standard time frame for movement. In this tutorial we won't have a use for it but we will implement it in the code so you can see how to apply it to your projects. The most common usage of the TimerClass is to use the frame time to figure out what percentage of a second has passed in the current frame and then move the objects by that percentage.

```
////////////////////////////////////////////////////////////////////////////////
// Filename: timerclass.h
////////////////////////////////////////////////////////////////////////////////
#ifndef _TIMERCLASS_H_
#define _TIMERCLASS_H_

//////////////
// INCLUDES //
//////////////
#include <windows.h>

////////////////////////////////////////////////////////////////////////////////
// Class name: TimerClass
////////////////////////////////////////////////////////////////////////////////
class TimerClass
{
public:
	TimerClass();
```

```
        TimerClass(const TimerClass&);
        ~TimerClass();

        bool Initialize();
        void Frame();

        float GetTime();

private:
        INT64 m_frequency;
        float m_ticksPerMs;
        INT64 m_startTime;
        float m_frameTime;
};

#endif
```

Timerclass.cpp

```
////////////////////////////////////////////////////////////////////////////////
// Filename: timerclass.cpp
////////////////////////////////////////////////////////////////////////////////
#include "timerclass.h"

TimerClass::TimerClass()
{
}

TimerClass::TimerClass(const TimerClass& other)
{
}

TimerClass::~TimerClass()
{
}
```

The Initialize function will first query the system to see if it supports high frequency timers. If it returns a frequency then we use that value to determine how many counter ticks will occur each millisecond. We can then use that value each frame to calculate the frame time. At the end of the Initialize function we query for the start time of this frame to start the timing.

```
bool TimerClass::Initialize()
{
        // Check to see if this system supports high performance timers.
        QueryPerformanceFrequency((LARGE_INTEGER*)&m_frequency);
        if(m_frequency == 0)
        {
                return false;
        }

        // Find out how many times the frequency counter ticks every millisecond.
        m_ticksPerMs = (float)(m_frequency / 1000);

        QueryPerformanceCounter((LARGE_INTEGER*)&m_startTime);

        return true;
}
```

The Frame function is called for every single loop of execution by the main program. This way we can calculate the difference of time between loops and determine the time it took to execute this frame. We query, calculate, and then store the time for this frame into m_frameTime so that it can be used by any calling object for synchronization. We then store the current time as the start of the next frame.

```
void TimerClass::Frame()
{
```

```
	INT64 currentTime;
	float timeDifference;

	QueryPerformanceCounter((LARGE_INTEGER*)& currentTime);

	timeDifference = (float)(currentTime - m_startTime);

	m_frameTime = timeDifference / m_ticksPerMs;

	m_startTime = currentTime;

	return;
}
```

GetTime returns the most recent frame time that was calculated.

```
float TimerClass::GetTime()
{
	return m_frameTime;
}
```

Systemclass.h

Now that we have looked at the three new classes we can examine how they fit into the framework. For all three new classes they will all be located under SystemClass.

[Same as previous codes]

We include the three new classes here.

```
#include "fpsclass.h"
#include "cpuclass.h"
#include "timerclass.h"

////////////////////////////////////////////////////////////////////////////////
// Class name: SystemClass
////////////////////////////////////////////////////////////////////////////////
class SystemClass
{
```

[Same as previous codes]

private:

[Same as previous codes]

We create a new object for each of the three new classes.

```
	FpsClass* m_Fps;
	CpuClass* m_Cpu;
	TimerClass* m_Timer;
};
```

Systemclass.cpp

We will cover just the changes to this class since the font tutorial.

```
////////////////////////////////////////////////////////////////////////////////
// Filename: systemclass.cpp
////////////////////////////////////////////////////////////////////////////////
#include "systemclass.h"
```

```
SystemClass::SystemClass()
{
```

[Same as previous codes]

Initialize the three new objects to null in the class constructor.

```
	m_Fps = 0;
	m_Cpu = 0;
	m_Timer = 0;
}

bool SystemClass::Initialize()
{
```

[Same as previous codes]

Create and initialize the FpsClass.

```
	// Create the fps object.
	m_Fps = new FpsClass;
	if(!m_Fps)
	{
		return false;
	}

	// Initialize the fps object.
	m_Fps->Initialize();
```

Create and initialize the CpuClass.

```
	// Create the cpu object.
	m_Cpu = new CpuClass;
	if(!m_Cpu)
	{
		return false;
	}

	// Initialize the cpu object.
	m_Cpu->Initialize();
```

Create and initialize the TimerClass.

```
	// Create the timer object.
	m_Timer = new TimerClass;
	if(!m_Timer)
	{
		return false;
	}

	// Initialize the timer object.
	result = m_Timer->Initialize();
	if(!result)
	{
		MessageBox(m_hwnd, L"Could not initialize the Timer object.", L"Error", MB_OK);
		return false;
	}

	return true;
}


void SystemClass::Shutdown()
{
```

[Same as previous codes]

Release the three new class objects here in the Shutdown function.

```
	// Release the timer object.
	if(m_Timer)
	{
		delete m_Timer;
		m_Timer = 0;
	}

	// Release the cpu object.
	if(m_Cpu)
	{
		m_Cpu->Shutdown();
		delete m_Cpu;
		m_Cpu = 0;
	}

	// Release the fps object.
	if(m_Fps)
	{
		delete m_Fps;
		m_Fps = 0;
	}

	// Shutdown the window.
	ShutdownWindows();

	return;
}
```

The final change is the Frame function. Each of the new classes needs to call its own Frame function for each frame of execution the application goes through. Once the Frame for each has been called we can now query for the updated data in each and send it into the GraphicsClass for use.

```
bool SystemClass::Frame()
{
	bool result;

	// Update the system stats.
	m_Timer->Frame();
	m_Fps->Frame();
	m_Cpu->Frame();

	// Do the input frame processing.
	result = m_Input->Frame();
	if(!result)
	{
		return false;
	}

	// Do the frame processing for the graphics object.
	result = m_Graphics->Frame(m_Fps->GetFps(), m_Cpu->GetCpuPercentage(), m_Timer->GetTime());
	if(!result)
	{
		return false;
	}

	return true;
}
```

Graphicsclass.h

[Same as previous codes]

We disable vsync for this tutorial so the application will run as fast as possible.

```
const bool VSYNC_ENABLED = false;
```

[Same as previous codes]

```
class GraphicsClass
{
public:
```

[Same as previous codes]

```
	bool Frame(int, int, float);
	bool Render(float);
```

[Same as previous codes]

```
};
```

Graphicsclass.cpp

[Same as previous codes]

The Frame function now takes in the fps, cpu, and timer counts. The fps and cpu count are set in the TextClass so they can be rendered to the screen.

```
bool GraphicsClass::Frame(int fps, int cpu, float frameTime)
{
```

[Same as previous codes]

```
	// Set the frames per second.
	result = m_Text->SetFps(fps, m_D3D->GetDeviceContext());
	if(!result)
	{
		return false;
	}

	// Set the cpu usage.
	result = m_Text->SetCpu(cpu, m_D3D->GetDeviceContext());
	if(!result)
	{
		return false;
	}

	// Render the graphics scene.
	result = Render(rotation);
	if (!result)
	{
		return false;
	}

	return true;
}
```

Textclass.h

```
class TextClass
{
```

[Same as previous codes]

```
public:
```

[Same as previous codes]

We now have two new functions for setting the fps count and the cpu usage.

```
bool SetFps(int, ID3D11DeviceContext*);
bool SetCpu(int, ID3D11DeviceContext*);
```

[Same as previous codes]

};

Textclass.cpp

[Same as previous codes]

The SetFps function takes the fps integer value given to it and then converts it to a string. Once the fps count is in a string format it gets concatenated to another string so it has a prefix indicating that it is the fps speed. After that it is stored in the sentence structure for rendering. The SetFps function also sets the color of the fps string to green if above 60 fps, yellow if below 60 fps, and red if below 30 fps.

```
bool TextClass::SetFps(int fps, ID3D11DeviceContext* deviceContext)
{
	char tempString[16];
	char fpsString[16];
	float red, green, blue;
	bool result;


	// Truncate the fps to below 10,000.
	if(fps > 9999)
	{
		fps = 9999;
	}

	// Convert the fps integer to string format.
	_itoa_s(fps, tempString, 10);

	// Setup the fps string.
	strcpy_s(fpsString, "Fps: ");
	strcat_s(fpsString, tempString);

	// If fps is 60 or above set the fps color to green.
	if(fps >= 60)
	{
		red = 0.0f;
		green = 1.0f;
		blue = 0.0f;
	}

	// If fps is below 60 set the fps color to yellow.
	if(fps < 60)
	{
		red = 1.0f;
		green = 1.0f;
		blue = 0.0f;
	}

	// If fps is below 30 set the fps color to red.
	if(fps < 30)
	{
		red = 1.0f;
		green = 0.0f;
		blue = 0.0f;
	}
```

```
	// Update the sentence vertex buffer with the new string information.
	result = UpdateSentence(m_sentence1, fpsString, 20, 20, red, green, blue, deviceContext);
	if(!result)
	{
		return false;
	}

	return true;
}
```

The SetCpu function is similar to the SetFps function. It takes the cpu value and converts it to a string which is then stored in the sentence structure and rendered.

```
bool TextClass::SetCpu(int cpu, ID3D11DeviceContext* deviceContext)
{
	char tempString[16];
	char cpuString[16];
	bool result;


	// Convert the cpu integer to string format.
	_itoa_s(cpu, tempString, 10);

	// Setup the cpu string.
	strcpy_s(cpuString, "Cpu: ");
	strcat_s(cpuString, tempString);
	strcat_s(cpuString, "%");

	// Update the sentence vertex buffer with the new string information.
	result = UpdateSentence(m_sentence2, cpuString, 20, 40, 0.0f, 1.0f, 0.0f, deviceContext);
	if(!result)
	{
		return false;
	}

	return true;
}
```

Summary

Now we can see the FPS and CPU usage while rendering our scenes. As well we now have precision timers which we use to ensure translation and rotation of objects is consistent regardless of the frame speed the application is running at.

HW 8 (Due 05/28, 5 pt)

1. Render several high polygonal 3D objects (more than 100,000 meshes in total, different in shape, and not simple primitives). All models should be properly textured. Use all three lightings: ambient, diffuse, specular.

2. Display the following rendering information on the screen: FPS (frames per second), CPU usage (%), Objects (total number), Polygons (total number), Screen size (pixels)

3. Using Direct Input, add a camera navigation through the 3D world space using a combination of key and mouse inputs. For example, the WASD keys can translate the camera while the mouse can change a view orientation.

Submit a ZIP file of VS project and source codes to **ClassNet**.