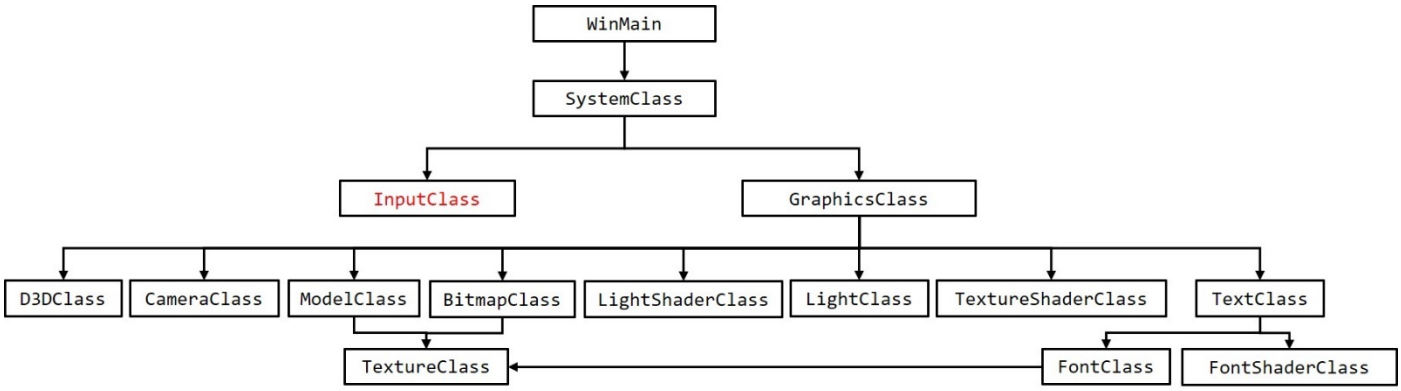
Tutorial: Direct Input & Sound

Direct Input

This tutorial will cover using Direct Input in DirectX 11. Direct Input is the high-speed method of interfacing with input devices that the DirectX API provides. In DirectX 11, the Direct Input portion of the API has not changed from previous versions, it is still version 8. However Direct Input was implemented very well in the first place (like direct sound) so there has not been any need to update it. Direct Input provides incredible speed over the regular windows input system. Any high-performance application (e.g. video games) that requires highly responsive input devices should be using Direct Input.

Framework



This tutorial will focus on how to implement Direct Input for **keyboard** and **mouse** devices. We will also use the TextClass to display the current location of the mouse pointer. As the previous tutorials already had an InputClass, we will just rewrite it using Direct Input instead of the Windows methods that were previously used.

Inputclass.h

You need to define the version of Direct Input you are using in the header or the compiler will generate annoying messages that it is defaulting to version 8.

```
#define DIRECTINPUT_VERSION 0x0800
```

The following two libraries need to be linked for Direct Input to work.

```
#pragma comment(lib, "dinput8.lib")
#pragma comment(lib, "dxguid.lib")
```

This is the required header for Direct Input.

```
#include <dinput.h>

class InputClass
{
public:
	InputClass();
	InputClass(const InputClass&);
	~InputClass();

	bool Initialize(HINSTANCE, HWND, int, int);
	void Shutdown();
	bool Frame();

	bool IsEscapePressed();
	void GetMouseLocation(int&, int&);

private:
	bool ReadKeyboard();
	bool ReadMouse();
```

```
        void ProcessInput();

private:
```

The first three private member variables are the interfaces to Direct Input, the keyboard device, and the mouse device.

```
        IDirectInput8* m_directInput;
        IDirectInputDevice8* m_keyboard;
        IDirectInputDevice8* m_mouse;
```

The next two private member variables are used for recording the current state of the keyboard and mouse devices.

```
        unsigned char m_keyboardState[256];
        DIMOUSESTATE m_mouseState;

        int m_screenWidth, m_screenHeight;
        int m_mouseX, m_mouseY;
};
```

Inputclass.cpp

```
#include "inputclass.h"
```

The class constructor initializes the Direct Input interface variables to null.

```
InputClass::InputClass()
{
        m_directInput = 0;
        m_keyboard = 0;
        m_mouse = 0;
}

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

InputClass::~InputClass()
{
}

bool InputClass::Initialize(HINSTANCE hinstance, HWND hwnd, int screenWidth, int screenHeight)
{
        HRESULT result;

        // Store the screen size which will be used for positioning the mouse cursor.
        m_screenWidth = screenWidth;
        m_screenHeight = screenHeight;

        // Initialize the location of the mouse on the screen.
        m_mouseX = 0;
        m_mouseY = 0;
```

This function call will initialize the interface to Direct Input. Once you have a Direct Input object you can initialize other input devices.

```
        // Initialize the main direct input interface.
        result = DirectInput8Create(hinstance, DIRECTINPUT_VERSION, IID_IDirectInput8,
                (void**)&m_directInput, NULL);
        if(FAILED(result))
        {
                return false;
        }
```

The first input device we will initialize will be the keyboard.

```
// Initialize the direct input interface for the keyboard.
result = m_directInput->CreateDevice(GUID_SysKeyboard, &m_keyboard, NULL);
if(FAILED(result))
{
	return false;
}

// Set the data format.    In this case since it is a keyboard we can use the predefined data format.
result = m_keyboard->SetDataFormat(&c_dfDIKeyboard);
if(FAILED(result))
{
	return false;
}
```

Setting the cooperative level of the keyboard is important in both what it does and how you use the device from that point forward. In this case we will set it to not share with other programs (DISCL_EXCLUSIVE). This way if you press a key only your application can see that input and no other application will have access to it. However if you want other applications to have access to keyboard input while your program is running you can set it to non-exclusive (DISCL_NONEXCLUSIVE). Now the print screen key works again and other running applications can be controlled by the keyboard and so forth. Just remember that if it is non-exclusive and you are running in a windowed mode then you will need to check for when the device loses focus and when it re-gains that focus so it can re-acquire the device for use again. This focus loss generally happens when other windows become the main focus over your window or your window is minimized.

```
// Set the cooperative level of the keyboard to not share with other programs.
result = m_keyboard->SetCooperativeLevel(hwnd, DISCL_FOREGROUND | DISCL_EXCLUSIVE);
if(FAILED(result))
{
	return false;
}
```

Once they keyboard is setup we then call Acquire to finally get access to the keyboard for use from this point forward.

```
// Now acquire the keyboard.
result = m_keyboard->Acquire();
if(FAILED(result))
{
	return false;
}
```

The next input device we setup is the mouse.

```
// Initialize the direct input interface for the mouse.
result = m_directInput->CreateDevice(GUID_SysMouse, &m_mouse, NULL);
if(FAILED(result))
{
	return false;
}

// Set the data format for the mouse using the pre-defined mouse data format.
result = m_mouse->SetDataFormat(&c_dfDIMouse);
if(FAILED(result))
{
	return false;
}
```

We use non-exclusive cooperative settings for the mouse. We will have to check for when it goes in and out of focus and re-acquire it each time.

```
// Set the cooperative level of the mouse to share with other programs.
result = m_mouse->SetCooperativeLevel(hwnd, DISCL_FOREGROUND | DISCL_NONEXCLUSIVE);
if(FAILED(result))
{
	return false;
}
```

Once the mouse is setup we acquire it so that we can begin using it.

```
	// Acquire the mouse.
	result = m_mouse->Acquire();
	if(FAILED(result))
	{
		return false;
	}

	return true;
}
```

The Shutdown function releases the two devices and the interface to Direct Input. Notice that the devices are always un-acquired first and then released.

```
void InputClass::Shutdown()
{
	// Release the mouse.
	if(m_mouse)
	{
		m_mouse->Unacquire();
		m_mouse->Release();
		m_mouse = 0;
	}

	// Release the keyboard.
	if(m_keyboard)
	{
		m_keyboard->Unacquire();
		m_keyboard->Release();
		m_keyboard = 0;
	}

	// Release the main interface to direct input.
	if(m_directInput)
	{
		m_directInput->Release();
		m_directInput = 0;
	}

	return;
}
```

The Frame function for the InputClass will read the current state of the devices into state buffers we setup. After the state of each device is read it then processes the changes.

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

	// Read the current state of the keyboard.
	result = ReadKeyboard();
	if(!result)
	{
		return false;
	}

	// Read the current state of the mouse.
	result = ReadMouse();
	if(!result)
	{
		return false;
	}

	// Process the changes in the mouse and keyboard.
```

```
	ProcessInput();

	return true;
}
```

ReadKeyboard will read the state of the keyboard into the m_keyboardState variable. The state will show any keys that are currently pressed or not pressed. If it fails reading the keyboard then it can be for one of five different reasons. The only two that we want to recover from are if the focus is lost or if it becomes un-acquired. If this is the case we call acquire each frame until we do get control back. The window may be minimized in which case Acquire will fail, but once the window comes to the foreground again then Acquire will succeed and we will be able to read the keyboard state. The other three error types we don't want to recover from in this tutorial.

```
bool InputClass::ReadKeyboard()
{
	HRESULT result;

	// Read the keyboard device.
	result = m_keyboard->GetDeviceState(sizeof(m_keyboardState), (LPVOID)&m_keyboardState);
	if(FAILED(result))
	{
		// If the keyboard lost focus or was not acquired then try to get control back.
		if((result == DIERR_INPUTLOST) || (result == DIERR_NOTACQUIRED))
		{
			m_keyboard->Acquire();
		}
		else
		{
			return false;
		}
	}

	return true;
}
```

ReadMouse will read the state of the mouse similar to how ReadKeyboard read in the state of the keyboard. However the state of the mouse is just changes in the position of the mouse from the last frame. So for example updates will look like the mouse has moved 5 units to the right, but it will not give you the actual position of the mouse on the screen. This delta information is very useful for different purposes and we can maintain the position of the mouse on the screen ourselves.

```
bool InputClass::ReadMouse()
{
	HRESULT result;

	// Read the mouse device.
	result = m_mouse->GetDeviceState(sizeof(DIMOUSESTATE), (LPVOID)&m_mouseState);
	if(FAILED(result))
	{
		// If the mouse lost focus or was not acquired then try to get control back.
		if((result == DIERR_INPUTLOST) || (result == DIERR_NOTACQUIRED))
		{
			m_mouse->Acquire();
		}
		else
		{
			return false;
		}
	}

	return true;
}
```

The ProcessInput function is where we deal with the changes that have happened in the input devices since the last frame. For this tutorial we will just do a simple mouse location update similar to how Windows keeps track of where the mouse cursor is. To do so we use the m_mouseX and m_mouseY variables that were initialized to zero and

simply add the changes in the mouse position to these two variables. This will maintain the position of the mouse based on the user moving the mouse around.

Note that we do check to make sure the mouse location never goes off the screen. Even if the user keeps moving the mouse to the left we will just keep the cursor at the zero position until they start moving it to the right again.

```
void InputClass::ProcessInput()
{
	// Update the location of the mouse cursor based on the change of the mouse location during the frame.
	m_mouseX += m_mouseState.lX;
	m_mouseY += m_mouseState.lY;

	// Ensure the mouse location doesn't exceed the screen width or height.
	if(m_mouseX < 0)  { m_mouseX = 0; }
	if(m_mouseY < 0)  { m_mouseY = 0; }

	if(m_mouseX > m_screenWidth)  { m_mouseX = m_screenWidth; }
	if(m_mouseY > m_screenHeight) { m_mouseY = m_screenHeight; }

	return;
}
```

I have added a function to the InputClass called IsEscapePressed. This shows how to utilize the keyboard to check if specific keys are currently being pressed. You can write other functions to check for any other keys that are of interest to your application.

```
bool InputClass::IsEscapePressed()
{
	// Do a bitwise and on the keyboard state to check if the escape key is currently being pressed.
	if(m_keyboardState[DIK_ESCAPE] & 0x80)
	{
		return true;
	}

	return false;
}
```

GetMouseLocation is a helper function we wrote which returns the location of the mouse. GraphicsClass can get this info and then use TextClass to render the mouse X and Y position to the screen.

```
void InputClass::GetMouseLocation(int& mouseX, int& mouseY)
{
	mouseX = m_mouseX;
	mouseY = m_mouseY;
	return;
}
```

Systemclass.cpp

I will just cover the functions that changed with the removal of the Windows Input system and addition of the DirectX Input system.

```
bool SystemClass::Initialize()
{
```

Initialization of the Input object is now different as it requires handles to the window, instance, and the screen size variables. It also returns a boolean value to indicate if it was successful or not in starting Direct Input.

```
	// Initialize the input object.
	result = m_Input->Initialize(m_hinstance, m_hwnd, screenWidth, screenHeight);
	if(!result)
	{
		MessageBox(m_hwnd, L"Could not initialize the input object.", L"Error", MB_OK);
		return false;
```

```
	}
}

void SystemClass::Shutdown()
{
```

Releasing the Input object now requires a Shutdown call previous to deleting the object.

```
	// Release the input object.
	if(m_Input)
	{
		m_Input->Shutdown();
		delete m_Input;
		m_Input = 0;
	}
}

void SystemClass::Run()
{
	while(!done)
	{
```

[Same as previous codes]

```
		// If windows signals to end the application then exit out.
		if(msg.message == WM_QUIT)
		{
			done = true;
		}
		else
		{
```

[Same as previous codes]

The check for the escape key in the Run function is now done slightly different by checking the return value of the helper function in the InputClass.

```
			// Check if the user pressed escape and wants to quit.
			if(m_Input->IsEscapePressed() == true)
			{
				done = true;
			}
		}
	}

	return;
}

bool SystemClass::Frame()
{
	bool result;
	int mouseX, mouseY;
```

During the Frame function we call the Input object's own Frame function to update the states of the keyboard and mouse. This call can fail so we need to check the return value.

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

After the input device updates have been read, we update the GraphicsClass with the location of the mouse so it can render that in text on the screen.

```
	// Get the location of the mouse from the input object,
	m_Input->GetMouseLocation(mouseX, mouseY);

	// Do the frame processing for the graphics object.
	result = m_Graphics->Frame(mouseX, mouseY);
	if(!result)
	{
		return false;
	}

	return true;
}
```

We have removed the Windows keyboard reads from the MessageHandler function. Direct Input handles all of this for us now.

```
LRESULT CALLBACK SystemClass::MessageHandler(HWND hwnd, UINT umsg, WPARAM wparam, LPARAM
	lparam)
{
	return DefWindowProc(hwnd, umsg, wparam, lparam);
}
```

Graphicsclass.h

```
class GraphicsClass
{
```

The Frame function now takes two integers for the mouse position updates each frame.

```
public:
	bool Frame(int, int);
};
```

Graphicsclass.cpp

The Frame function now takes in the mouse X and Y position and then has the TextClass object update the text strings that will write the position onto the screen.

```
bool GraphicsClass::Frame(int mouseX, int mouseY)
{
	// Set the location of the mouse.
	result = m_Text->SetMousePosition(mouseX, mouseY, m_D3D->GetDeviceContext());
	if (!result)
	{
		return false;
	}
}
```

Textclass.h

```
class TextClass
{
```

TextClass now has a new function for setting the location of the mouse.

```
public:
	bool SetMousePosition(int, int, ID3D11DeviceContext*);
};
```

Textclass.cpp

We now have a new function in the TextClass which converts the mouse X and Y position into two strings and then updates the two sentences so that the position of the mouse can be rendered to the screen.

```
bool TextClass::SetMousePosition(int mouseX, int mouseY, ID3D11DeviceContext* deviceContext)
{
	char tempString[16];
	char mouseString[16];
	bool result;

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

	// Setup the mouseX string.
	strcpy_s(mouseString, "Mouse X: ");
	strcat_s(mouseString, tempString);

	// Update the sentence vertex buffer with the new string information.
	result = UpdateSentence(m_sentence1, mouseString, 20, 20, 1.0f, 1.0f, 1.0f, deviceContext);
	if(!result)
	{
		return false;
	}

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

	// Setup the mouseY string.
	strcpy_s(mouseString, "Mouse Y: ");
	strcat_s(mouseString, tempString);

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

	return true;
}
```

Summary

As you can see setting up Direct Input in DirectX 11 is simple and it gives us high speed access to information from the input devices.

Exercise

1. Use the information from the 2D Rendering tutorial and combine it with this one to create your own mouse cursor that moves with the mouse movement.
2. Implement a function that reads the keyboard buffer and displays what you type on the screen.

Direct Sound

This tutorial will cover the basics of using **Direct Sound** in DirectX 11 as well as how to load and play **.wav** audio files. This tutorial is based on the code in the previous DirectX 11 tutorials. We will cover a couple basics about Direct Sound in DirectX 11 as well as a bit about sound formats before we start the code part of the tutorial.

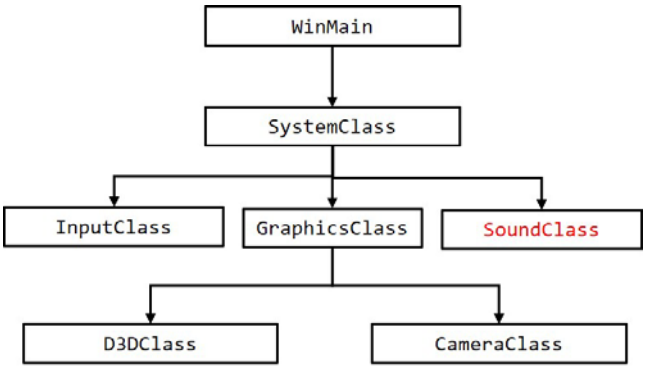
The first thing you will notice is that in DirectX 11 the Direct Sound API is still the same one from DirectX 8. The only major difference is that hardware sound mixing is generally not available on the latest Windows operating systems. The reason being is for security and operating system consistency all hardware calls now have to go through a security layer. The older sound cards used to have DMA (direct memory access) which was very fast but doesn't work with this new Windows security model. So all sound mixing is now done at the software level and hence no hardware acceleration is directly available to this API.

The nice thing about Direct Sound is that you can play any audio format you want. In this tutorial, the .wav audio format is only covered, but you can replace the .wav code with .mp3 or anything you prefer. You can even use your own audio format if you have created one. Direct Sound is so easy to use you just create a sound buffer with the play back format you would like and then copy your audio format into the buffer's format and then it is ready to play. You can see why so many applications use Direct Sound due to its pure simplicity.

Note that Direct Sound does use two different kinds of buffers which are **primary** and **secondary** buffers. The primary buffer is the main sound memory buffer on your default sound card, USB headset, and so forth. Secondary buffers are buffers you create in memory and load your sounds into. When you play a secondary buffer the Direct Sound API takes care of mixing that sound into the primary buffer which then plays the sound. If you play multiple secondary buffers at the same time it will mix them together and play them in the primary buffer. Also note that all buffers are circular so you can set them to repeat indefinitely.

Framework

To start the tutorial we will first look at the updated frame work. The only new class is the SoundClass which contains all the DirectSound and .wav format functionality. We have removed the other classes to simplify this tutorial.



Soundclass.h

The SoundClass encapsulates the DirectSound functionality as well as the .wav audio loading and playing capabilities.

```
#ifndef _SOUNDCLASS_H_
#define _SOUNDCLASS_H_
```

The following libraries and headers are required for DirectSound to compile properly.

```
#pragma comment(lib, "dsound.lib")
#pragma comment(lib, "dxguid.lib")
#pragma comment(lib, "winmm.lib")

#include <windows.h>
#include <mmsystem.h>
#include <dsound.h>
#include <stdio.h>
```

```
class SoundClass
{
private:
```

The WaveHeaderType structure used here is for the .wav file format. When loading in .wav files we first read in the header to determine the required information for loading in the .wav audio data. If you are using a different format you will want to replace this header with the one required for your audio format.

```
	struct WaveHeaderType
	{
		char chunkId[4];
		unsigned long chunkSize;
		char format[4];
		char subChunkId[4];
		unsigned long subChunkSize;
		unsigned short audioFormat;
		unsigned short numChannels;
		unsigned long sampleRate;
		unsigned long bytesPerSecond;
		unsigned short blockAlign;
		unsigned short bitsPerSample;
		char dataChunkId[4];
		unsigned long dataSize;
	};

public:
	SoundClass();
	SoundClass(const SoundClass&);
	~SoundClass();
```

Initialize and Shutdown will handle everything needed for this tutorial. The Initialize function will initialize DirectSound and load in the .wav audio file and then play it once. Shutdown will release the .wav file and shutdown DirectSound.

```
	bool Initialize(HWND);
	void Shutdown();

private:
	bool InitializeDirectSound(HWND);
	void ShutdownDirectSound();

	bool LoadWaveFile(char*, IDirectSoundBuffer8**);
	void ShutdownWaveFile(IDirectSoundBuffer8**);

	bool PlayWaveFile();

private:
	IDirectSound8* m_DirectSound;
	IDirectSoundBuffer* m_primaryBuffer;
```

Note that we only have one secondary buffer as this tutorial only loads in one sound.

```
	IDirectSoundBuffer8* m_secondaryBuffer1;
};

#endif
```

Soundclass.cpp

```
#include "soundclass.h"
```

Use the class constructor to initialize the private member variables that are used inside the sound class.

```
SoundClass::SoundClass()
{
```

```
	m_DirectSound = 0;
	m_primaryBuffer = 0;
	m_secondaryBuffer1 = 0;
}

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

SoundClass::~SoundClass()
{
}

bool SoundClass::Initialize(HWND hwnd)
{
	bool result;
```

First initialize the DirectSound API as well as the primary buffer. Once that is initialized then the LoadWaveFile function can be called which will load in the .wav audio file and initialize the secondary buffer with the audio information from the .wav file. After loading is complete then PlayWaveFile is called which then plays the .wav file once.

```
	// Initialize direct sound and the primary sound buffer.
	result = InitializeDirectSound(hwnd);
	if(!result)
	{
		return false;
	}

	// Load a wave audio file onto a secondary buffer.
	result = LoadWaveFile("../Engine/data/sound01.wav", &m_secondaryBuffer1);
	if(!result)
	{
		return false;
	}

	// Play the wave file now that it has been loaded.
	result = PlayWaveFile();
	if(!result)
	{
		return false;
	}

	return true;
}
```

The Shutdown function first releases the secondary buffer which held the .wav file audio data using the ShutdownWaveFile function. Once that completes this function then called ShutdownDirectSound which releases the primary buffer and the DirectSound interface.

```
void SoundClass::Shutdown()
{
	// Release the secondary buffer.
	ShutdownWaveFile(&m_secondaryBuffer1);

	// Shutdown the Direct Sound API.
	ShutdownDirectSound();

	return;
}
```

InitializeDirectSound handles getting an interface pointer to DirectSound and the default primary sound buffer. Note that you can query the system for all the sound devices and then grab the pointer to the primary sound buffer for a specific device, however I've kept this tutorial simple and just grabbed the pointer to the primary buffer of the default sound device.

```
bool SoundClass::InitializeDirectSound(HWND hwnd)
{
	HRESULT result;
	DSBUFFERDESC bufferDesc;
	WAVEFORMATEX waveFormat;

	// Initialize the direct sound interface pointer for the default sound device.
	result = DirectSoundCreate8(NULL, &m_DirectSound, NULL);
	if(FAILED(result))
	{
		return false;
	}

	// Set the cooperative level to priority so the format of the primary sound buffer can be modified.
	result = m_DirectSound->SetCooperativeLevel(hwnd, DSSCL_PRIORITY);
	if(FAILED(result))
	{
		return false;
	}
```

We have to setup the description of how we want to access the primary buffer. The dwFlags are the important part of this structure. In this case we just want to setup a primary buffer description with the capability of adjusting its volume. There are other capabilities you can grab but we are keeping it simple for now.

```
	// Setup the primary buffer description.
	bufferDesc.dwSize = sizeof(DSBUFFERDESC);
	bufferDesc.dwFlags = DSBCAPS_PRIMARYBUFFER | DSBCAPS_CTRLVOLUME;
	bufferDesc.dwBufferBytes = 0;
	bufferDesc.dwReserved = 0;
	bufferDesc.lpwfxFormat = NULL;
	bufferDesc.guid3DAlgorithm = GUID_NULL;

	// Get control of the primary sound buffer on the default sound device.
	result = m_DirectSound->CreateSoundBuffer(&bufferDesc, &m_primaryBuffer, NULL);
	if(FAILED(result))
	{
		return false;
	}
```

Now that we have control of the primary buffer on the default sound device we want to change its format to our desired audio file format. Here we have decided we want high quality sound so we will set it to uncompressed CD audio quality.

```
	// Setup the format of the primary sound bufffer.
	// In this case it is a .WAV file recorded at 44,100 samples per second in 16-bit stereo (cd audio format).
	waveFormat.wFormatTag = WAVE_FORMAT_PCM;
	waveFormat.nSamplesPerSec = 44100;
	waveFormat.wBitsPerSample = 16;
	waveFormat.nChannels = 2;
	waveFormat.nBlockAlign = (waveFormat.wBitsPerSample / 8) * waveFormat.nChannels;
	waveFormat.nAvgBytesPerSec = waveFormat.nSamplesPerSec * waveFormat.nBlockAlign;
	waveFormat.cbSize = 0;

	// Set the primary buffer to be the wave format specified.
	result = m_primaryBuffer->SetFormat(&waveFormat);
	if(FAILED(result))
	{
		return false;
	}

	return true;
}
```

The ShutdownDirectSound function handles releasing the primary buffer and DirectSound interfaces.

```
void SoundClass::ShutdownDirectSound()
{
	// Release the primary sound buffer pointer.
	if(m_primaryBuffer)
	{
		m_primaryBuffer->Release();
		m_primaryBuffer = 0;
	}

	// Release the direct sound interface pointer.
	if(m_DirectSound)
	{
		m_DirectSound->Release();
		m_DirectSound = 0;
	}

	return;
}
```

The LoadWaveFile function is what handles loading in a .wav audio file and then copies the data onto a new secondary buffer. If you are looking to do different formats you would replace this function or write a similar one.

```
bool SoundClass::LoadWaveFile(char* filename, IDirectSoundBuffer8** secondaryBuffer)
{
	int error;
	FILE* filePtr;
	unsigned int count;
	WaveHeaderType waveFileHeader;
	WAVEFORMATEX waveFormat;
	DSBUFFERDESC bufferDesc;
	HRESULT result;
	IDirectSoundBuffer* tempBuffer;
	unsigned char* waveData;
	unsigned char *bufferPtr;
	unsigned long bufferSize;
```

To start first open the .wav file and read in the header of the file. The header will contain all the information about the audio file so we can use that to create a secondary buffer to accommodate the audio data. The audio file header also tells us where the data begins and how big it is. You will notice we check for all the needed tags to ensure the audio file is not corrupt and is the proper wave file format containing RIFF, WAVE, fmt, data, and WAVE_FORMAT_PCM tags. We also do a couple other checks to ensure it is a 44.1KHz stereo 16bit audio file. If it is mono, 22.1 KHZ, 8bit, or anything else then it will fail ensuring we are only loading the exact format we want.

```
	// Open the wave file in binary.
	error = fopen_s(&filePtr, filename, "rb");
	if(error != 0)
	{
		return false;
	}

	// Read in the wave file header.
	count = fread(&waveFileHeader, sizeof(waveFileHeader), 1, filePtr);
	if(count != 1)
	{
		return false;
	}

	// Check that the chunk ID is the RIFF format.
	if((waveFileHeader.chunkId[0] != 'R') || (waveFileHeader.chunkId[1] != 'I') ||
	   (waveFileHeader.chunkId[2] != 'F') || (waveFileHeader.chunkId[3] != 'F'))
	{
		return false;
	}

	// Check that the file format is the WAVE format.
	if((waveFileHeader.format[0] != 'W') || (waveFileHeader.format[1] != 'A') ||
```

```
    (waveFileHeader.format[2] != 'V') || (waveFileHeader.format[3] != 'E'))
{
        return false;
}

// Check that the sub chunk ID is the fmt format.
if((waveFileHeader.subChunkId[0] != 'f') || (waveFileHeader.subChunkId[1] != 'm') ||
    (waveFileHeader.subChunkId[2] != 't') || (waveFileHeader.subChunkId[3] != ' '))
{
        return false;
}

// Check that the audio format is WAVE_FORMAT_PCM.
if(waveFileHeader.audioFormat != WAVE_FORMAT_PCM)
{
        return false;
}

// Check that the wave file was recorded in stereo format.
if(waveFileHeader.numChannels != 2)
{
        return false;
}

// Check that the wave file was recorded at a sample rate of 44.1 KHz.
if(waveFileHeader.sampleRate != 44100)
{
        return false;
}

// Ensure that the wave file was recorded in 16 bit format.
if(waveFileHeader.bitsPerSample != 16)
{
        return false;
}

// Check for the data chunk header.
if((waveFileHeader.dataChunkId[0] != 'd') || (waveFileHeader.dataChunkId[1] != 'a') ||
    (waveFileHeader.dataChunkId[2] != 't') || (waveFileHeader.dataChunkId[3] != 'a'))
{
        return false;
}
```

Now that the wave header file has been verified we can setup the secondary buffer we will load the audio data onto. We have to first set the wave format and buffer description of the secondary buffer similar to how we did for the primary buffer. There are some changes though since this is secondary and not primary in terms of the dwFlags and dwBufferBytes.

```
// Set the wave format of secondary buffer that this wave file will be loaded onto.
waveFormat.wFormatTag = WAVE_FORMAT_PCM;
waveFormat.nSamplesPerSec = 44100;
waveFormat.wBitsPerSample = 16;
waveFormat.nChannels = 2;
waveFormat.nBlockAlign = (waveFormat.wBitsPerSample / 8) * waveFormat.nChannels;
waveFormat.nAvgBytesPerSec = waveFormat.nSamplesPerSec * waveFormat.nBlockAlign;
waveFormat.cbSize = 0;

// Set the buffer description of the secondary sound buffer that the wave file will be loaded onto.
bufferDesc.dwSize = sizeof(DSBUFFERDESC);
bufferDesc.dwFlags = DSBCAPS_CTRLVOLUME;
bufferDesc.dwBufferBytes = waveFileHeader.dataSize;
bufferDesc.dwReserved = 0;
bufferDesc.lpwfxFormat = &waveFormat;
bufferDesc.guid3DAlgorithm = GUID_NULL;
```

Now the way to create a secondary buffer is fairly strange. First step is that you create a temporary IDirectSoundBuffer with the sound buffer description you setup for the secondary buffer. If this succeeds then you can use that temporary buffer to create a IDirectSoundBuffer8 secondary buffer by calling QueryInterface with the IID_IDirectSoundBuffer8 parameter. If this succeeds then you can release the temporary buffer and the secondary buffer is ready for use.

```
// Create a temporary sound buffer with the specific buffer settings.
result = m_DirectSound->CreateSoundBuffer(&bufferDesc, &tempBuffer, NULL);
if(FAILED(result))
{
	return false;
}

// Test the buffer format against the direct sound 8 interface and create the secondary buffer.
result = tempBuffer->QueryInterface(IID_IDirectSoundBuffer8, (void**)&*secondaryBuffer);
if(FAILED(result))
{
	return false;
}

// Release the temporary buffer.
tempBuffer->Release();
tempBuffer = 0;
```

Now that the secondary buffer is ready we can load in the wave data from the audio file. First, we load it into a memory buffer so we can check and modify the data if we need to. Once the data is in memory you then lock the secondary buffer, copy the data to it using a memcpy, and then unlock it. This secondary buffer is now ready for use. Note that locking the secondary buffer can actually take in two pointers and two positions to write to. This is because it is a circular buffer and if you start by writing to the middle of it you will need the size of the buffer from that point so that you don't write outside the bounds of it. This is useful for streaming audio and such. In this tutorial we create a buffer that is the same size as the audio file and write from the beginning to make things simple.

```
// Move to the beginning of the wave data which starts at the end of the data chunk header.
fseek(filePtr, sizeof(WaveHeaderType), SEEK_SET);

// Create a temporary buffer to hold the wave file data.
waveData = new unsigned char[waveFileHeader.dataSize];
if(!waveData)
{
	return false;
}

// Read in the wave file data into the newly created buffer.
count = fread(waveData, 1, waveFileHeader.dataSize, filePtr);
if(count != waveFileHeader.dataSize)
{
	return false;
}

// Close the file once done reading.
error = fclose(filePtr);
if(error != 0)
{
	return false;
}

// Lock the secondary buffer to write wave data into it.
result = (*secondaryBuffer)->Lock(0, waveFileHeader.dataSize, (void**)&bufferPtr, (DWORD*)&bufferSize,
NULL, 0, 0);
if(FAILED(result))
{
	return false;
}

// Copy the wave data into the buffer.
memcpy(bufferPtr, waveData, waveFileHeader.dataSize);
```

```
	// Unlock the secondary buffer after the data has been written to it.
	result = (*secondaryBuffer)->Unlock((void*)bufferPtr, bufferSize, NULL, 0);
	if(FAILED(result))
	{
		return false;
	}

	// Release the wave data since it was copied into the secondary buffer.
	delete [] waveData;
	waveData = 0;

	return true;
}
```

ShutdownWaveFile just does a release of the secondary buffer.

```
void SoundClass::ShutdownWaveFile(IDirectSoundBuffer8** secondaryBuffer)
{
	// Release the secondary sound buffer.
	if(*secondaryBuffer)
	{
		(*secondaryBuffer)->Release();
		*secondaryBuffer = 0;
	}

	return;
}
```

The PlayWaveFile function will play the audio file stored in the secondary buffer. The moment you use the Play function it will automatically mix the audio into the primary buffer and start it playing if it wasn't already. Also note that we set the position to start playing at the beginning of the secondary sound buffer otherwise it will continue from where it last stopped playing. And since we set the capabilities of the buffer to allow us to control the sound we set the volume to maximum here.

```
bool SoundClass::PlayWaveFile()
{
	HRESULT result;

	// Set position at the beginning of the sound buffer.
	result = m_secondaryBuffer1->SetCurrentPosition(0);
	if(FAILED(result))
	{
		return false;
	}

	// Set volume of the buffer to 100%.
	result = m_secondaryBuffer1->SetVolume(DSBVOLUME_MAX);
	if(FAILED(result))
	{
		return false;
	}

	// Play the contents of the secondary sound buffer.
	result = m_secondaryBuffer1->Play(0, 0, 0);
	if(FAILED(result))
	{
		return false;
	}

	return true;
}
```

Systemclass.h

Here we include the new SoundClass header file.

```
#include "soundclass.h"

class SystemClass
{
```

We create a new private variable for the SoundClass object.

```
private:
	SoundClass* m_Sound;
};
```

Systemclass.cpp

We will just cover the functions that have changed since the previous tutorial.

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

Initialize the new SoundClass object to null in the class constructor.

```
	m_Sound = 0;
}

bool SystemClass::Initialize()
{
```

Here is where we create the SoundClass object and then initialize it for use. Note that in this tutorial the initialization will also start the wave file playing.

```
	// Create the sound object.
	m_Sound = new SoundClass;
	if(!m_Sound)
	{
		return false;
	}

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

void SystemClass::Shutdown()
{
```

In the SystemClass::Shutdown we also shutdown the SoundClass object and release it.

```
	// Release the sound object.
	if(m_Sound)
	{
		m_Sound->Shutdown();
		delete m_Sound;
		m_Sound = 0;
	}

}
```

Summary

The engine now supports the basics of Direct Sound. It currently just plays a single wave file once you start the program.

Exercise

1. Replace the sound01.wav file with your own 44.1KHz 16bit 2channel audio wave file and run the program again.
2. Rewrite the program to load two wave files and play them simultaneously.
3. Change the wave to loop instead of playing just once.

[Original script: www.rastertek.com]