

## Synthesizing sound on flash, java, macos and ios platforms

Sound waves are continuous pressure changes in the air. In digital technology, we call these changes "pulses", and change count per second is called "sampling rate". The standard raw format for digital audio is "pulse code modulating" format (PCM), which is simply a queue of amplitude values which tells the hardware how powerful signals should it produce for output.

The human brain can process more than 100.000 changes per second. However, because of historical reasons the most common sampling rate in digital technology is 44100 samples per second.

So most of the time coders have to deal with this sampling rate, on every channel, and the usual bit depth is 16. This means that one pulse in the pulse stream can have  $2^{16}$  levels.

So one channel ( a mono stream ) uses 2 bytes per frame, and a stereo stream, which contains two channels, takes 4 bytes per frame.

That's all we have to know to start experimenting. We are going to make a simple sound generator, which generates a simple musical "A" tone. The musical "A" tone is a simple sinus wave, having 440 periods per second.

I separate the logic in two classes : the WaveGenerator, and the WavePlayer.

The WaveGenerator does nothing spectacular : it will generate a sinus wave on the given frequency sampled at the given sampling rate. The platform dependent magic will be in the WavePlayer class, it will contain the hardware initializer code, and the platform-specific instructions.

The scheme for WaveGenerator class is this :

```
field position    // actual position/angle
field previous   // previous position/angle

field increment  // position increment per step
field frequency  // tone frequency

method construct ( samplingrate, tonefrequency )
{
    increment = 2 * PI / samplingrate; // calculate increment
    frequency = tonefrequency;
}

method nextAmplitude ( )
{
    // rolling back position/angle
    if ( position > 2 * PI ) position -= 2 * PI;
    // passing back sinus of actual position multiplied by frequency
    return Math.sin( previous * frequency );
}
```

That's all, it's quite straightforward, let's check it out on all platforms :

flash

```
package
{
    public class WaveGenerator
    {
        private var position : Number;
        private var previous : Number;

        private var increment : Number;
        private var frequency : Number;

        public function WaveGenerator ( pRate : int , pFrequency : int )
        {
            position = 0;
            previous = 0;

            increment = 2 * Math.PI / pRate;
            frequency = pFrequency;
        }

        public function nextAmplitude ( ) : Number
        {
            previous = position;
            position += increment;

            if ( position > 2 * Math.PI ) position -= 2 * Math.PI;
            return Math.sin( previous * frequency );
        }
    }
}
```

java :

```
public class WaveGenerator
{
    private double position;
    private double previous;

    private double increment;
    private double frequency;

    public WaveGenerator ( int pRate , int pFrequency )
    {
        position = 0;
        previous = 0;

        increment = 2 * Math.PI / pRate;
        frequency = pFrequency;
    }
}
```

```

    }

    public double nextAmplitude ( )
    {
        previous = position;
        position += increment;

        if ( position > 2 * Math.PI ) position -= 2 * Math.PI;
        return Math.sin( previous * frequency );
    }
}

```

objective-c

@implementation WaveGenerator

```

- ( WaveGenerator* )
samplingRate : ( int ) pRate
toneFrequency: ( int ) pFrequency
{
    position = 0;
    previous = 0;

    increment = 2 * M_PI / pRate;
    frequency = pFrequency;

    return self;
}

- ( float )
nextAmplitude
{
    previous = position;
    position += increment;

    if ( position > 2 * M_PI ) position -= 2 * M_PI;
    return sinf( previous * frequency );
}

```

@end

Let's see the WavePlayer class. We are heading from the easiest platform to the hardest.

Flash

In flash you don't/can't do any low-level setup. It supports only 16 bit depth stereo sounds sampled in 44100 Hz, but you have to provide only floating numbers between -1 and 1 to the API for pulses.

The workflow is the following : Create a new instance of the Sound class. Add an event listener to this class, which listens for SAMPLE\_DATA events. These events will fire when the hardware runs out of buffer, and it needs more data. In the event handling function add more data to the buffer. Finally, you call soundInstance.play( ).

Event firing will never be accurate, so we mustn't hang on timing, we have to use a separate frame counter, or we can use the event's position field, which is the same. If we push 4410 frames to the buffer, the next event will come around  $4410/44100 = .1$  seconds.

As i said before, we fill up the buffer using float values from -1 to 1, and our WaveGenerator class does exactly this. So let's see the code :

```
#import "WaveGenerator.h"

package
{
    import flash.events.SampleDataEvent;
    import flash.media.Sound;

    public class WavePlayer
    {

        private var sound:Sound;
        private var generator:WaveGenerator;
        private var frameCount:int = 8192;

        public function WavePlayer( pGenerator : WaveGenerator )
        {

            sound = new Sound( );
            generator = pGenerator;
            sound.addEventListener
(SampleDataEvent.SAMPLE_DATA,onSampleData);
            sound.play();
        }

        private function onSampleData ( pEvent : SampleDataEvent ) : void
        {

            // trace( "onSampleData " + pEvent.position );

            for ( var index :int = 0 ; index < frameCount ; index ++ )
            {

                var amplitude : Number = generator.nextAmplitude();

                pEvent.data.writeFloat( amplitude );
                pEvent.data.writeFloat( amplitude );

            }

        }

    }
}
```

Java

In java we have to dive to a lower level. We will use the Java Media Framework.

In java we have to ask for specific data lines ( input streams ) to specific audio mixers, and have to push data to them continuously, like in flash. To get a specific data line, we have to specify an audio format first, and AudioSystem class will give us a data line if our hardware is capable of playing sound in the provided format, or nothing, if not.

We use the most common format mentioned before : 16 bits, 44100 Hz sampling rate, 2 channels, 4 bytes per frame. In java, we have the possibility to set the integer type for the samples, byte ordering, and playing frame rate, which should be the same as the sampling rate.

```
AudioFormat format = new AudioFormat( AudioFormat.Encoding.PCM_SIGNED , 44100 ,  
16 , 2 , 4 , 44100 , true );
```

Then we can start playing the dataline, and providing frame data for the input stream.

To make playing more solid, i've created a fillBuffer function, which pre-creates buffer parts for every .1 seconds. We have to calculate buffer size and frame count before this.

```
bufferSize = (int)(44100 * 4 * .1 ); // .1 second length buffer  
frameCount = bufferSize / 4;
```

fillBuffer function will convert float values returned from WaveGenerator to signed 16-bit ( short ) values using Short.MAX\_VALUE :

```
short sample = (short) ( source.nextAmplitude() * Short.MAX_VALUE );
```

and pushes them into the bytearrayoutputstream in stereo :

```
stream.write( sample >> 8 );  
stream.write( sample );  
stream.write( sample >> 8 );  
stream.write( sample );
```

And that's all. Check it out :

```
import java.io.ByteArrayOutputStream;  
  
import javax.sound.sampled.AudioFormat;  
import javax.sound.sampled.AudioSystem;  
import javax.sound.sampled.SourceDataLine;  
  
public class WavePlayer  
{  
  
    private WaveGenerator source;  
    private int bufferSize;  
    private int frameCount;  
    private ByteArrayOutputStream stream;  
    private SourceDataLine dataLine;  
  
    public WavePlayer ( WaveGenerator pSource )  
    {
```

```

source = pSource;
stream = new ByteArrayOutputStream();

AudioFormat format = new AudioFormat( AudioFormat.Encoding.PCM_SIGNED ,
                                        44100 ,
                                        16 ,
                                        2 ,
                                        4 ,
                                        44100 ,
                                        true );

bufferSize = (int)(44100 * 4 * .1 );
frameCount = bufferSize / 4;

try
{
    dataLine = AudioSystem.getSourceDataLine(format);
    dataLine.open(format, bufferSize );
}
catch ( Exception e )
{
    e.printStackTrace();
}

fillBuffer( );
dataLine.start();

try
{
    while ( true )
    {
        fillBuffer( );
        dataLine.write(stream.toByteArray(), 0 ,stream.size());
    }
}
catch (Exception e)
{
    e.printStackTrace();
}
finally
{
    // plays what's left and and closes the audioChannel

    dataLine.drain();
    dataLine.close();
}
}

public void fillBuffer ( )
{
    stream.reset();

    for ( int index = 0 ; index < frameCount ; index++ )
    {
        short sample = (short) ( source.nextAmplitude() * Short.MAX_VALUE );

        try
        {
            stream.write( sample >> 8 );
            stream.write( sample );
            stream.write( sample >> 8 );
            stream.write( sample );
        }
    }
}

```

```

        catch ( Exception e )
        {
            e.printStackTrace();
        }
    }

}

public void play ( )
{
}
}

```

## Mac/iOs

And now for something completely different, and difficult. On mac/ios you have to use the CoreAudio framework. It is not inputstream-based like the flash/java solution, here you have to provide an arbitrary number of buffers for sound, and CoreAudio will rotate them, and notifies you in a standard C call if a buffer is emptied, and you have to fill up that buffer while it plays the other buffers, so it's really cpu friendly and effective, but quite difficult to handle.

First we have to define the audio format. It's a simple structure, with additional options compared to java, let it speak for itself :

```

audioFormat = (AudioStreamBasicDescription)
{
    .mFormatID = kAudioFormatLinearPCM,
    .mFormatFlags = 0 | kAudioFormatFlagIsPacked | kAudioFormatFlagIsSignedInteger |
kAudioFormatFlagsNativeEndian ,
    .mSampleRate = 44100,
    .mBitsPerChannel = 16,
    .mChannelsPerFrame = 2,
    .mBytesPerFrame = 4,
    .mFramesPerPacket = 1,
    .mBytesPerPacket = 4,
};

```

bufferSize and frameCount are calculated in the same way

```

bufferSize = audioFormat.mSampleRate * audioFormat.mBytesPerPacket * .1;
frameCount = bufferSize / audioFormat.mBytesPerFrame;

```

this is followed by audio session initialization, and setting up the audio queue output :

```

AudioQueueNewOutput(
    &audioFormat,           // audio format
    updateQueue,          // callback function on buffer update request
    self,                 // custom parameter, passing ourself
    NULL,                 // timing comes from the audioqueue's thread
    NULL,                 // default loop mode
    0,                    // flags
    &audioQueue           // audio queue which stores the created session
);

```

updateQueue has to be a plain C function, and since we started from an objective-c class, we have to place it over the @implementation part. It contains nothing but a simple callback to the objective c function

```

- ( void )
updateQueue      : ( AudioQueueRef ) pAudioQueue
updateBuffer     : ( AudioQueueBufferRef ) pLastBuffer

```

where we can handle buffer fillup and rotation.

After setting up the queue, we fill up the starting buffers, and call `AudioQueueStart(audioQueue, NULL);` and the process starts.

Let's check out the `updateQueue:updateBuffer:` method.

Like in java, we convert the floating point values to signed 16-bit integers.

```
short sampleValue = [ waveSource nextAmplitude ] * INT16_MAX;
```

And at the end, we push the buffer back to the queue.

```
AudioQueueEnqueueBuffer( pAudioQueue , pLastBuffer , 0 , NULL );
```

Check out the whole class :

```

#import "WavePlayer.h"

// plain c callback function called by AudioQueue when last buffer is empty
void updateQueue ( void* inData , AudioQueueRef inAQ , AudioQueueBufferRef
inBuffer )
{
    WavePlayer* staticApp = (WavePlayer*) inData;
    [staticApp updateQueue:inAQ updateBuffer:inBuffer];
}

@implementation WavePlayer

- ( WavePlayer* )
waveSource : ( WaveGenerator* ) pSource
{
    // construct
    audioFormat = (AudioStreamBasicDescription)
    {
        .mFormatID = kAudioFormatLinearPCM,
        .mFormatFlags = 0 | kAudioFormatFlagIsPacked |
kAudioFormatFlagIsSignedInteger | kAudioFormatFlagsNativeEndian ,
        .mSampleRate = 44100,
        .mBitsPerChannel = 16,
        .mChannelsPerFrame = 2,
        .mBytesPerFrame = 4,
        .mFramesPerPacket = 1,
        .mBytesPerPacket = 4,
    };

    // assign
    waveSource = pSource;
    bufferSize = audioFormat.mSampleRate * audioFormat.mBytesPerPacket * .1;
    frameCount = bufferSize / audioFormat.mBytesPerFrame;

    // start
    AudioSessionInitialize(NULL, NULL, NULL, NULL);
}

```

```

        UInt32 category = kAudioSessionCategory_MediaPlayback; // plays through
sleep lock and silent switch
        AudioSessionSetProperty(
            kAudioSessionProperty_AudioCategory,
            sizeof(category),
            &category
        );
        AudioSessionSetActive(true);

        // create output and assign to audioQueue

        AudioQueueNewOutput(
            &audioFormat, // audio format
            updateQueue, // callback function on buffer update request
            self, // custom parameter, passing ourself
            NULL, // timing comes from the audioqueue's
thread
            NULL, // default loop mode
            0, // flags
session
            &audioQueue // audio queue which stores the created
        );

        // allocate and fill up buffers
        for (int i = 0; i < 4; ++i)
        {
            AudioQueueAllocateBuffer(
                audioQueue,
                bufferSize,
                &frameBuffers[i]
            );

            updateQueue(
                self,
                audioQueue,
                frameBuffers[i]
            );
        }

        return self;
    }

- ( void )
play
{
    AudioQueueStart(audioQueue, NULL);
}

- ( void )
updateQueue : ( AudioQueueRef ) pAudioQueue
updateBuffer : ( AudioQueueBufferRef ) pLastBuffer
{
    // NSLog( @"SynthesizerAppDelegate updateQueue" );

    int channelCount = 0;
    short* buffer = pLastBuffer->mAudioData;

    for ( int i = 0; i < frameCount ; i ++ )
    {
        short sampleValue = [ waveSource nextAmplitude ] * INT16_MAX;

        buffer[ channelCount++ ] = sampleValue;
        buffer[ channelCount++ ] = sampleValue;
    }
}

```

```
        }  
        pLastBuffer->mAudioDataByteSize = bufferSize;  
        AudioQueueEnqueueBuffer( pAudioQueue , pLastBuffer , 0 ,  
NULL );  
    }
```

@end

And that's all folks!

2010. 07. 30.