[**Kinect for Windows SDK开发入门(十二)：语音识别 上**](http://www.cnblogs.com/yangecnu/archive/2012/05/03/KinectSDK_SpeechRecognition_part1.html)

Kinect的麦克风阵列在Kinect设备的下方。这一阵列由4个独立的水平分布在Kinect下方的麦克风组成。虽然每一个麦克风都捕获相同的音频信号，但是组成阵列可以探测到声音的来源方向。使得能够用来识别从某一个特定的方向传来的声音。麦克风阵列捕获的音频数据流经过复杂的音频增强效果算法处理来移除不相关的背景噪音。所有这些复杂操作在Kinect硬件和Kinect SDK之间进行处理，这使得能够在一个大的空间范围内，即使人离麦克风一定的距离也能够进行语音命令的识别。

在Kinect第一次作为Xbox360的外设发布时，[骨骼追踪和语音识别是Kinect SDK最受开发者欢迎的特性](http://v.youku.com/v_show/id_XMjgwMTg5MDAw.html%20)，但是相比骨骼追踪，语音识别中麦克风阵列的强大功能有一点被忽视了。一部分原因归于Kinect中的令人兴奋的骨骼追踪系统，另一部分原因在于Xbox游戏操控面板以及Kinect体感游戏没有充分发挥Kinect音频处理的优点。

作为一个开始使用Kinect进行应用开发的开发者，Kinect上的麦克风阵列的出现使得基于Kinect应用程序的功能更加强大。虽然Kinect的视觉分析令人印象深刻，但是仍然不能很好的对马达进行控制。当我们从一种人机交互界面切换到另一种人机交互界面：如从命令行交互应用程序到标签页交互界面，再到鼠标图形用户界面或者触摸交互界面时，每一种交互界面都提供了各种最基本的更加容易实现的操作，这个操作就是选择。进一步，可以说，每一种交互界面都改进了我们对对象进行选择的能力。奇怪的是，Kinect破坏了这一趋势。

在Kinect应用程序中，选择操作是最复杂和难以掌握的行为之一。Xbox360中最初的选择操作是通过将手放到特定的位置，然后保持一段时间。在《舞林大会》游戏中，通过一个短暂的停顿加上滑动操作来对选择操作进行了一点改进。这一改进也被应用在了Xbox的操作面板中。另外的对选择进行改进的操作包括某种特定的手势，如将胳膊举起来。

这些问题，可以通过将语音识别指令和骨骼追踪系统结合起来产生一个复合的姿势来相对简单的解决：保持某一动作，然后通过语音执行。菜单的设计也可以通过首先展示菜单项，然后让用户说出菜单项的名称来进行选择-很多Xbox中的游戏已经使用了这种方式。可以预见，无论是程序开发者还是游戏公司，这种复合的解决方案在未来会越来越多的应用到新的交互方式中，而不用再像以前那样使用指然后点(point and click)这种方式来选择。

**1. 麦克风阵列**

安装完Microsoft Kinect SDK之后，语音识别的组件会自动安装。Kinect的麦克风阵列工作在一些语音识别的类库之上，这些类库是从Vista系统之时就有的。他们包括语音捕获DirectX多媒体对象(DirectX Media Object，DMO)以及语音识别API(Speech Recognition API,SAPI)。

在C#中，Kinect SDK提供了对语音捕获DMO的封装。语音捕获DMO最初是被设计用来给麦克风阵列提供API来支持一些功能如回声消除(acoustic echo cancellation,AEC)，自动增益控制(automatic gain control,AGC)和噪声抑制(noise suppression)。这些功能在SDK的音频控制类中可以找到。 Kinect SDK中音频处理对语音捕获DMO进行了简单封装，并专门针对Kinect传感器进行了性能优化。为了能够使用Kinect SDK进行语音识别，自动安装的类库包括:Speech Platform API, Speech Platform SDK和Kinect for Windows Runtime Language Pack。

语音识别API能够简化操作系统自带的语音识别所需的类库。例如，如果你想通过普通的麦克风而不是Kinect麦克风阵列添加一些语音指令到桌面应用程序中去，可以使用也可以不使用Kinect SDK。

Kinect for windows 运行语言包是一系列的语言模型，用来在Kinect SDK和语音识别API组件之间进行互操作。就像Kinect骨骼识别需要大量的计算模型来提供决策树信息来分析节点位置那样，语音识别API也需要复杂的模型来辅助解释从Kinect麦克风阵列接收到的语言模型。Kinect语言包提供了这些模型来优化语音指令的识别。

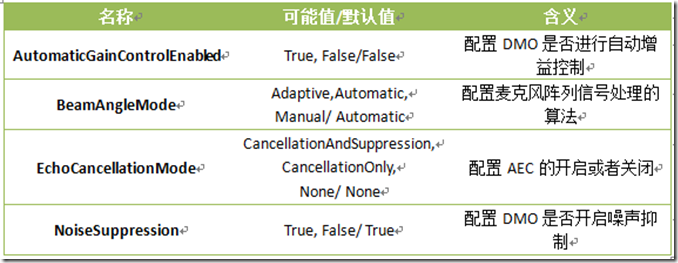
**1.1 MSR Kinect Audio**

Kinect中处理音频主要是通过KinectAudioSource这个对象来完成的。KinectAudioSource类的主要作用是从麦克风阵列中提取原始的或者经过处理的音频流。音频流可能会经过一系列的算法来处理以提高音频质量，这些处理包括:降噪、自动增益控制和回声消除。KinectAudioSource能够进行一些配置使得Kinect麦克风阵列可以以不同的模式进行工作。也能够用来探测从那个方向来的哪种音频信息最先达到麦克风以及用来强制麦克风阵列接受指定方向的音频信息。

本节尽量不会去介绍一些音频处理技术方面的较低层次的技术。但是为了使用KinectAudioSource,了解语音捕获以及语音传输中的一些术语可能会对熟悉KinectAudioSource中的一些属性和方法有所帮助。

* 回声消除(acoustic echo cancellation, AEC) 当用户的声音从麦克风返回时，就会产生回声。最简单的例子就是用户在打电话时能够听到自己的声音，这些声音有一些延迟，会在对方那里重复一段时间。回声消除通过提取发声者的声音模式，然后根据这一模式从麦克风接收到的音频中挑选出特定的音频来消除回声。
* 回声抑制(acoustic echo suppression, AES) 它是指通过一系列的算法来进一步消除AEC处理后所遗留的回声。
* 自动增益控制(acoustic gain control, AGS) 它涉及到一些算法用来使用户的声音的振幅与时间保持一致。例如当用户靠近或者或远离麦克风时，声音会出现变得响亮或更柔和，AGC通过算法使得这一过程变得更加明显。
* 波束成形(beamforming) 指的是模拟定向麦克风的算法技术。和只有一个麦克风不同，波速成形技术用于麦克风阵列中 （如Kinect 传感器上的麦克风阵列）使得麦克风阵列产生和使用多个固定麦克风的效果相同。
* 中心削波(center clipping) 用来移除在单向传输中经AEC处理后残留的小的回声。
* 帧尺寸(Frame Size) AEC算法处理PCM音频样本是是一帧一帧处理的。帧尺寸是样本中音频帧的大小。
* 获取增益边界(Gain Bounding) 该技术确保麦克风有正确的增益级别。如果增益过高，获取到的信号可能过于饱和，会被剪切掉。这种剪切有非线性的效果，会使得AEC算法失败。如果增益过低，信噪比会比较低，也会使得AEC算法失败或者执行的不好。
* 噪声填充(Noise Filling) 向中心削波移除了残留的回波信号后的部分信号中添加少量的噪音。和留下空白的沉默信号相比，这能够获得更好的用户体验。
* 噪声抑制 (NS) 用于从麦克风接收到的音频信号中剔除非言语声音。通过删除背景噪音，实际讲话者的声音能够被麦克风更清楚更明确的捕获到。
* Optibeam Kinect传感器从四个麦克风中能够获得11个波束。 这11个波束是逻辑结构，而四个通道是物理结构。Optibeam 是一种系统模式用来进行波束成形。
* 信噪比(Signal-to-Noise Ratio,SNR) 信号噪声比用来度量语音信号和总体背景噪声的比例，信噪比越高越好。
* 单通道(Single Channel) Kinect传感器有四个麦克风，因此支持4个通道，单通道是一种系统模式用来关闭波束成形。

KinectAudioSource类提供一些对音频捕获多方面的较高层次的控制，虽然它并没有提供DMO中的所有功能。KinectAudioSource中用来调整音频处理的各种属性被称之为功能(features)属性。下表中列出了可以调整的功能属性。Kinect SDK早期的Beta版本视图提供了DMO中的所有功能以使得能够有更加强大的控制能力，但是这也极大的增加了复杂度。SDK的正式版本提取了DMO中的所有可能的配置然后将其封装为特征属性使得我们不用关心底层的配置细节。对于没有接触过这些底层配置属性的开发者来说，这是一种巨大的解脱。

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201205/201205030902091379.png)

EchoCancellationMode是一个隐藏在不起眼的名称后面神奇的技术之一。他可能的设置如下表。为了适应AEC，需要给EchoCancellationSpeakerIndex属性赋一个int值来指定那一个用户的噪音需要控制。SDK会自动执行活动麦克风的发现和初始化。

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201205/201205030902104510.png)

BeamAngleMode对底层的DMO系统模式和麦克风阵列属性进行了抽象封装。在DMO级别上，他决定了是由DMO还是应用程序进行波束成形。在这一基础上Kinect for Windows SDK提供了额外的一系列算法来进行波束成行。通常，可以将该属性设置为Adaptive，将这些复杂的操作交给SDK进行处理。下表展示了每一个可设置值的属性。

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201205/201205030902117051.png)

自适应波速成形(Adaptive beamforming)能够发挥Kinect传感器的特性优势，根据骨骼追踪所找到的游戏者，从而找出正确的声音源。和骨骼追踪一样，Kinect的波束成形特性也能够使用手动模式，允许应用程序来设定要探测声音的方向。要使用Kinect传感器作为定向的麦克风，需要将波束角度模式设定为Manual然后设置KinectAudioSource的ManualBeamAngle属性。

**1.2 语音识别**

语音识别可以分为两类：对特定命令的识别(recognition of command)和对自由形式的语音的识别(recognition of free-form dictation)。自由形式的语音识别需要训练软件来识别特定的声音以提高识别精度。一般做法是让讲话人大声的朗读一系列的语料来使得软件能够识别讲话人声音的特征模式，然后根据这一特征模式来进行识别。

而命令识别(Command recognition)则应用了不同的策略来提高识别精度。和必须识别说话人声音不同，命令识别限制了说话人所讲的词汇的范围。基于这一有限的范围，命令识别可以不需要熟悉讲话人的语音模式就可以推断出说话人想要说的内容。

考虑到Kinect的特性，使用Kinect进行自由形式的语音识别没有多大意义。Kinect SDK的设计初衷是让大家能够简单容易的使用，因此，SDK提供了Microsoft.Speech类库来原生支持语音命令的识别。Microsoft.Speech类库是Microsoft语音识别技术的服务器版本。如果你想使用System.Speech类库中的语音识别能力，可以使用Windows操作系统内建的桌面版语音识别来通过Kinect的麦克风来建立一个自由语音识别系统。但是通过将Kinect的麦克风和System.Speech类库组合开发的自由语音识别系统的识别效果可能不会太好。这是因为Kinect for windows运行时语言包，能够适应从开放空间中的声音，而不是从麦克风发出的声音，这些语言模型在 System.Speech 中不能够使用。

Microsoft.Speech类库的语音识别功能是通过SpeechRecognitionEngine对象提供的。SpeechRecognitionEngine类是语音识别的核心，它负责从Kinect传感器获取处理后的音频数据流，然后分析和解译这些数据流，然后匹配出最合适的语音命令。引擎给基本发声单元一定的权重，如果判断出发声包含特定待识别的命令，就通过事件进行进一步处理，如果不包含，直接丢掉这部分音频数据流。

我们需要告诉SpeechRecognitionEngine从一个特定的称之为语法(grammars)的对象中进行查找。Grammar对象由一系列的单个单词或者词语组成。如果我们不关心短语的部分内容，可以使用语法对象中的通配符。例如，我们可能不会在意命令包含短语"an" apple或者"the" apple，语法中的通配符告诉识别引擎这两者都是可以接受的。此外，我们还可以添加一个称之为Choices的对象到语法中来。选择类(Choices)是通配符类(Wildcard)的一种，它可以包含多个值。但与通配符不同的是，我们可以指定可接受的值的顺序。例如如果我们想要识别“Give me some fruit”我们不关心fruit单词之前的内容，但是我们想将fruit替换为其它的值，如apple，orange或者banana等值。这个语法可以通过下面的代码来实现。Microsoft.Speech类库中提供了一个GrammarBuilder类来建立语法(grammars)。

var choices = new Choices();

choices.Add("fruit");

choices.Add("apple");

choices.Add("orange");

choices.Add("banana");

var grammarBuilder = new GrammarBuilder();

grammarBuilder.Append("give");

grammarBuilder.Append("me");

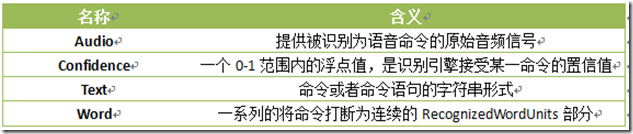
grammarBuilder.AppendWildcard();

grammarBuilder.Append(choices);

var grammar = new Grammar(grammarBuilder);

语法中的单词不区分大小写，但是出于一致性考虑，要么都用大写，要么都用小写。

语音识别引擎使用LoadGrammar方法将Grammars对象加载进来。语音识别引擎能够加载而且通常是加载多个语法对象。识别引擎有3个事件：SpeechHypothesized，SpeechRecognized和SpeechRecognitionRejected。 SpeechHypothesized事件是识别引擎在决定接受或者拒绝用户命令之前解释用户说话的内容。SpeechRecognitionRejected用来处理识别命令失败时需要执行的操作。SpeechRecognized是最重要的事件，他在引擎决定接受用户的语音命令时触发。该事件触发时，通过SpeechRecognizedEventArgs对象参数传递一些数据。SpeechRecognizedEventArgs类有一个Result属性，该属性描述如下：

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201205/201205030902117641.png)

实例化SpeechRecognitionEngine对象需要执行一系列特定的步骤。首先，需要设置识别引擎的ID编号。当安装了服务器版本的Microsoft语音库时，名为Microsoft Lightweight Speech Recognizier的识别引擎有一个为SR\_MS\_ZXX\_Lightweight\_v10.0的ID值（这个值根据你所安装的语音库的不同而不同）。当安装了Kinect for Windows运行时语音库时，第二个ID为Server Speech Recognition Language-Kinect(en-US)的语音库可以使用。这是Kinect中我们可以使用的第二个识别语音库。下一步SpeechRecognitionEngine需要指定正确的识别语音库。由于第二个语音识别库的ID可能会在以后有所改变，我们需要使用模式匹配来找到这一ID。最后，语音识别引擎需要进行配置，以接收来自KinectAudioSource对象的音频数据流。下面是执行以上过程的样板代码片段。

var source = new KinectAudioSource();

Func<RecognizerInfo, bool> matchingFunc = r =>

{

String value;

r.AdditionalInfo.TryGetValue("Kinect", out value);

return "True".Equals(value, StringComparison.InvariantCultureIgnoreCase)

&& "en-US".Equals(r.Culture.Name, StringComparison.InvariantCultureIgnoreCase);

};

RecognizerInfo ri = SpeechRecognitionEngine.InstalledRecognizers().Where(matchingFunc).FirstOrDefault();

var sre = new SpeechRecognitionEngine(ri.Id);

KinectSensor.KinectSensors[0].Start();

Stream s = source.Start();

sre.SetInputToAudioStream(s,new SpeechAudioFormatInfo(EncodingFormat.Pcm, 16000, 16, 1, 32000, 2, null));

sre.Recognize();

SetInputToAudioStream方法的第二个参数用来设置从Kinect获取的音频数据流的格式。在上面的代码中，我们设置音频编码格式为Pulse Code Modulation(PCM),每秒接收16000个采样，每个样本占16位，只有1个通道，每秒中产生32000字节数据，块对齐值设置为2。

Grammars加载到语音识别引擎后，引擎必须启动后才能进行识别，启动引擎有几种模式，可以使用同步或者异步模式启动。另外也可以识别一次或者继续识别从KinectAudioSource传来的多条语音命令。下表列出了开始语音识别的可选方法。

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201205/201205030902123562.png)

在文将使用一些简单的例子来展示如何使用KinectAudioSource和SpeechRecognitionEngine类。

**2. 获取音频数据**

虽然 KinectAudioSource 类的最主要作用是为语音识别引擎提供音频数据流，但是它也可以用于其他目的。他还能够用来录制 wav 文件。下面的示例将使用KinectAudioSource来开发一个音频录音机。使用这个项目作为录音机，读者可以修改Kinect sdk中KinectAudioSource的各个参数的默认值的来了解这些参数是如何控制音频数据流的产生。

**2.1 使用音频数据流**

虽然使用的是Kinect的音频相关类，而不是视觉元素类，但是建立一个Kinect音频项目的过程大致是类似的。

1. 创建一个名为KinectAudioRecorder的WPF应用项目。

2. 添加对Microsoft.Kinect.dll和Microsoft.Speech.dll的引用。

3. 在MainWindows中添加名为Play，Record和Stop三个按钮。

4. 将主窗体的名称改为“Audio Recorder”

在VS的设计视图中，界面看起来应该如下:

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201205/201205030902121120.png)

令人遗憾的是，C#没有一个方法能够直接写入wav文件。为了能够帮助我们生成wav文件，我们使用下面自定义的RecorderHelper类，该类中有一个称之为WAVFORMATEX的结构，他是C++中对象转换过来的，用来方便我们对音频数据进行处理。该类中也有一个称之为IsRecording的属性来使得我们可以停止录制。类的基本结构，以及WAVFORMATEX的结构和属性如下。我们也需要初始化一个私有名为buffer字节数组用来缓存我们从Kinect接收到的音频数据流。

class RecorderHelper

{

static byte[] buffer = new byte[4096];

static bool \_isRecording;

public static bool IsRecording

{

get

{

return \_isRecording;

}

set

{

\_isRecording = value;

}

}

struct WAVEFORMATEX

{

public ushort wFormatTag;

public ushort nChannels;

public uint nSamplesPerSec;

public uint nAvgBytesPerSec;

public ushort nBlockAlign;

public ushort wBitsPerSample;

public ushort cbSize;

}

}

为了完成这个帮助类，我们还需要添加三个方法:WriteString,WriteWavHeader和WriteWavFile方法。WriteWavFile方法如下，方法接受KinectAudioSource和FileStream对象，从KinectAudioSource对象中我们可以获取音频数据，我们使用FileStream来写入数据。方法开始写入一个假的头文件，然后读取Kinect中的音频数据流，然后填充FileStream对象，直到\_isRecoding属性被设置为false。然后检查已经写入到文件中的数据流大小，用这个值来改写之前写入的文件头。

public static void WriteWavFile(KinectAudioSource source, FileStream fileStream)

{

var size = 0;

//write wav header placeholder

WriteWavHeader(fileStream, size);

using (var audioStream = source.Start())

{

//chunk audio stream to file

while (audioStream.Read(buffer, 0, buffer.Length) > 0 && \_isRecording)

{

fileStream.Write(buffer, 0, buffer.Length);

size += buffer.Length;

}

}

//write real wav header

long prePosition = fileStream.Position;

fileStream.Seek(0, SeekOrigin.Begin);

WriteWavHeader(fileStream, size);

fileStream.Seek(prePosition, SeekOrigin.Begin);

fileStream.Flush();

}

public static void WriteWavHeader(Stream stream, int dataLength)

{

using (MemoryStream memStream = new MemoryStream(64))

{

int cbFormat = 18;

WAVEFORMATEX format = new WAVEFORMATEX()

{

wFormatTag = 1,

nChannels = 1,

nSamplesPerSec = 16000,

nAvgBytesPerSec = 32000,

nBlockAlign = 2,

wBitsPerSample = 16,

cbSize = 0

};

using (var bw = new BinaryWriter(memStream))

{

WriteString(memStream, "RIFF");

bw.Write(dataLength + cbFormat + 4);

WriteString(memStream, "WAVE");

WriteString(memStream, "fmt ");

bw.Write(cbFormat);

bw.Write(format.wFormatTag);

bw.Write(format.nChannels);

bw.Write(format.nSamplesPerSec);

bw.Write(format.nAvgBytesPerSec);

bw.Write(format.nBlockAlign);

bw.Write(format.wBitsPerSample);

bw.Write(format.cbSize);

WriteString(memStream, "data");

bw.Write(dataLength);

memStream.WriteTo(stream);

}

}

}

static void WriteString(Stream stream, string s)

{

byte[] bytes = Encoding.ASCII.GetBytes(s);

stream.Write(bytes, 0, bytes.Length);

}

使用该帮助方法，我们可以开始建立和配置KinectAudioSource对象。首先添加一个私有的\_isPlaying 布尔值来保存是否我们想要播放录制的wav文件。这能够帮助我们避免录音和播放功能同事发生。除此之外，还添加了一个MediaPlayer对象用来播放录制好的wav文件。\_recodingFileName用来保存最近录制好的音频文件的名称。代码如下所示，我们添加了几个属性来关闭和开启这三个按钮，他们是：IsPlaying，IsRecording，IsPlayingEnabled，IsRecordingEnabled和IsStopEnabled。为了使得这些对象可以被绑定，我们使MainWindows对象实现INotifyPropertyChanged接口，然后添加一个NotifyPropertyChanged事件以及一个OnNotifyPropertyChanged帮助方法。

在设置各种属性的逻辑中，先判断IsRecording属性，如果为false，再设置IsPlayingEnabled属性。同样的先判断IsPlaying属性为是否false，然后在设置IsRecordingEnabled属性。前端的XAML代码如下:

<Window x:Class="KinectRecordAudio.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

Title="Audio Recorder" Height="226" Width="405">

<Grid Width="369" Height="170">

<Button Content="Play" Height="44" HorizontalAlignment="Left" Margin="12,13,0,0" Name="button1" VerticalAlignment="Top" Width="114" Click="button1\_Click" IsEnabled="{Binding IsPlayingEnabled}" FontSize="18"></Button>

<Button Content="Record" Height="44" HorizontalAlignment="Left" Margin="132,13,0,0" Name="button2" VerticalAlignment="Top" Width="110" Click="button2\_Click" IsEnabled="{Binding IsRecordingEnabled}" FontSize="18"/>

<Button Content="Stop" Height="44" HorizontalAlignment="Left" Margin="248,13,0,0" Name="button3" VerticalAlignment="Top" Width="107" Click="button3\_Click" IsEnabled="{Binding IsStopEnabled}" FontSize="18"/>

</Grid>

</Window>

MainWindow构造函数中实例化一个MediaPlayer对象，将其存储到\_mediaPlayer变量中。因为Media Player对象在其自己的线程中，我们需要捕获播放完成的时间，来重置所有按钮的状态。另外，我们使用WPF中的技巧来使得我们的MainWindow绑定IsPlayingEnabled以及属性。我们将MainPage的DataContext属性设置给自己。这是提高代码可读性的一个捷径，虽然典型的做法是将这些绑定属性放置到各自单独的类中。

public MainWindow()

{

InitializeComponent();

this.Loaded += delegate { KinectSensor.KinectSensors[0].Start(); };

\_mplayer = new MediaPlayer();

\_mplayer.MediaEnded += delegate { \_mplayer.Close(); IsPlaying = false; };

this.DataContext = this;

}

现在，我们准备好了实例化KinectAudioSource类，并将其传递给之前创建的RecordHelper类。为了安全性，我们给RecordKinectAudio方法添加了锁。在加锁之前我们将IsRunning属性设置为true，方法结束后，将该属性设置回false。

private object lockObj = new object();

private void RecordKinectAudio()

{

lock (lockObj)

{

IsRecording = true;

var source = CreateAudioSource();

var time = DateTime.Now.ToString("hhmmss");

\_recordingFileName = time + ".wav";

using (var fileStream =

new FileStream(\_recordingFileName, FileMode.Create))

{

RecorderHelper.WriteWavFile(source, fileStream);

}

IsRecording = false;

}

}

private KinectAudioSource CreateAudioSource()

{

var source = KinectSensor.KinectSensors[0].AudioSource;

source.BeamAngleMode = BeamAngleMode.Adaptive;

source.NoiseSuppression = \_isNoiseSuppressionOn;

source.AutomaticGainControlEnabled = \_isAutomaticGainOn;

if (IsAECOn)

{

source.EchoCancellationMode = EchoCancellationMode.CancellationOnly;

source.AutomaticGainControlEnabled = false;

IsAutomaticGainOn = false;

source.EchoCancellationSpeakerIndex = 0;

}

return source;

}

为了保证不对之前录制好的音频文件进行再次写入，我们在每个音频文件录制完了之后，使用当前的时间位文件名创建一个新的音频文件。最后一步就是录制和播放按钮调用的方法。UI界面上的按钮调用Play\_Click和Record\_Click方法。这些方法只是调用实际对象的Play和Record方法。需要注意的是下面的Record方法，重新开启了一个新的线程来执行RecordKinectAudio方法。

private void Play()

{

IsPlaying = true;

\_mplayer.Open(new Uri(\_recordingFileName, UriKind.Relative));

\_mplayer.Play();

}

private void Record()

{

Thread thread = new Thread(new ThreadStart(RecordKinectAudio));

thread.Priority = ThreadPriority.Highest;

thread.Start();

}

private void Stop()

{

KinectSensor.KinectSensors[0].AudioSource.Stop();

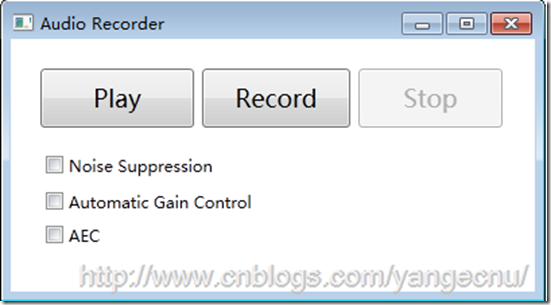
IsRecording = false;

}

现在就可以使用Kinect来录制音频文件了。运行程序，点击录制按钮，然后在房间里走动，以测试Kinect从不同距离录制的声音效果。在CreateAudioSource方法中，麦克风阵列配置为使用自适应波束成形，因此当你说话时，会跟随录制。点击停止录制按钮结束录制。然后点击播放，就可以播放刚才录制的音频文件了。

**2.2 对音频数据进行处理**

我们可以将之前的例子进行扩展，添加一些属性配置。在本节中，我们添加对噪音抑制和自动增益属性开关的控制。下图显示了修改后的程序界面。

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201205/201205030902132822.png)

bool \_isNoiseSuppressionOn;

bool \_isAutomaticGainOn;

bool \_isAECOn;

使用之前OnPropertyChanged帮助方法，我们能够创建一个绑定属性。代码如下：

public bool IsNoiseSuppressionOn

{

get

{

return \_isNoiseSuppressionOn;

}

set

{

if (\_isNoiseSuppressionOn != value)

{

\_isNoiseSuppressionOn = value;

OnPropertyChanged("IsNoiseSuppressionOn");

}

}

}

public bool IsAutomaticGainOn

{

get

{

return \_isAutomaticGainOn;

}

set

{

if (\_isAutomaticGainOn != value)

{

\_isAutomaticGainOn = value;

OnPropertyChanged("IsAutomaticGainOn");

}

}

}

逻辑部分代码写好后，在界面上添加几个Checkbox来控制.

<CheckBox Content="Noise Suppression" Height="16" HorizontalAlignment="Left" Margin="16,77,0,0" VerticalAlignment="Top" Width="142" IsChecked="{Binding IsNoiseSuppressionOn}" />

<CheckBox Content="Automatic Gain Control" Height="16" HorizontalAlignment="Left" Margin="16,104,0,0" VerticalAlignment="Top" IsChecked="{Binding IsAutomaticGainOn}"/>

<CheckBox Content="AEC" Height="44" HorizontalAlignment="Left" IsChecked="{Binding IsAECOn}" Margin="16,129,0,0" VerticalAlignment="Top" />

最后在CreateAudioSource方法中绑定这些值.

private KinectAudioSource CreateAudioSource()

{

var source = KinectSensor.KinectSensors[0].AudioSource;

source.BeamAngleMode = BeamAngleMode.Adaptive;

source.NoiseSuppression = \_isNoiseSuppressionOn;

source.AutomaticGainControlEnabled = \_isAutomaticGainOn;

return source;

}

运行程序，然后选中这几个选项，试着录音，可以看到噪声抑制在提高录音效果的作用了。当你在房间里来回走动，并提高或者降低声音时，自动增益控制的效果会更加明显。其它属性配置的效果可能比较细微，没有这两个效果明显。读者可以自己添加几个CheckBox进行测试。

**2.3 去除回声**

回声消除不单是KinectAudioSource类的一个属性，他还是Kinect的核心技术之一。要测试这一特性可能会比之前的测试要复杂。

要测试AEC的效果，需要在界面上添加一个CheckBox。然后创建一个IsAECOn属性来设置和保存这个设置的值。最后将CheckBox的IsCheck属性绑定到IsAECOn属性上。

我们将会在CreateAudioSource方法中配置这一属性。代码如下：

if (IsAECOn)

{

source.EchoCancellationMode = EchoCancellationMode.CancellationOnly;

source.AutomaticGainControlEnabled = false;

IsAutomaticGainOn = false;

source.EchoCancellationSpeakerIndex = 0;

}

上面的代码中SystemMode属性必须设置为Optibeam或者echo cancelation。需要关闭自动增益控制，因为在AEC下面不会起作用。另外，如果AutomaticGainOn属性不为false的话，需要将其设置为false。这样界面上的UI展现逻辑就有一点从冲突。AEC配置接下来需要找到我们使用的麦克风，以及正在使用麦克风的讲话人，以使得AEC算法能够从数据流中提取到需要的音频数据。运行程序，你可以播放之前录制的音频文件来测试这一特征。

**3. 结语**

本文简要介绍了Kinect中语音识别的基本概念，以及一些语音处理方面的术语。在此基础上以一个使用Kinect麦克风阵列来进行音频录制的例子展现了Kinect中音频处理的核心对象及其配置，限于篇幅原因，本文介绍到这里，所有代码点击[此处](http://files.cnblogs.com/yangecnu/KinectSDK_SpeechRecognition_part1_KinectRecordAudio.rar)下载。下一篇文章将继续介绍Kinect中的语音识别，并以两个小例子来展示语音识别中方向识别和语音命令识别，敬请期待！