**[Kinect for Windows SDK开发入门(十)：手势识别 上：基本概念](http://www.cnblogs.com/yangecnu/archive/2012/04/21/KinectSDK_GesturesDetection_part1_BasicConception.html)**

像点击(clicks)是GUI平台的核心，轻点(taps)是触摸平台的核心那样，手势(gestures)是Kinect应用程序的核心。和图形用户界面中的数字交互不同，手势是现实生活中存在的动作。如果没有电脑我们就不需要鼠标，但是没了Kinect，手势依然存在。从另一方面讲，手势是日常生活中人与人之间相互交流的一部分。手势能够增强演讲的说服力，能够用来强调和传递情感。像挥手(waving)或者指点(pointing)这些手势都是某种无声的演讲。

Kinect应用程序的设计和开发者的任务就是将这些现实生活中存在的手势映射到计算机交互中去以传达人的想法。尝试从鼠标或触摸式的GUI设计移植基于手势的自然交互界面要做很多工作。借鉴过去30多年来对于这一概念的研究，以及从一些Kinect for Xbox的体感游戏中获取一些设计理念，计算机工程师和交互设计师一起为Kinect创建了一系列新的手势库。

本文将会介绍用户体验的一些知识，并讨论如何将手势应用到Kinect应用程序中。我们将展示Kinect如何作为自然交互界面(Natural User Interface)的人机交互模型的一部分。我们将讨论一些具体的使用Kinect来进行手势识别及交互的例子。更重要的是，将会展示一些已经作为Kinect手势识别库中的手势。

**1. 什么是手势**

在许多不同的学科中，手势(gesture)有着其独特的含义，可能这些含义之间有某些异同。在艺术领域，手势被用来传达舞蹈中最富表现力的部分，特别是在亚洲舞蹈艺术中，手势被作为某些宗教符号或者象征。在交互设计领域，在基于触摸的自然交互界面中手势和操控有很大区别。

以上这些说明手势在不同的学科领域都有自己独特的含义。在学术领域都试图对手势定义一个抽象的概念。在用户体验设计领域使用最广泛的关于手势的定义实在Eric Hulteen 和Gord Kurtenbach 1990年发表的一篇名为人机交互中的手势(Gestures in Human-Computer Communication),定义如下：”手势是身体的运动，他包含一些信息。挥手道别是一种手势。敲击键盘不是手势，因为用手指的运动去敲击按键没有被观察，也不重要，他只表达的键盘被按下这一动作。(A gesture is a motion of the body that contains information. Waving goodbye is a gesture. Pressing a key on a keyboard is not a gesture because the motion of a finger on its way to hitting a key is neither observed nor significant. All that matters is which key was pressed)”

这个定义既解释了什么是手势也解释了什么不是手势。像这样的下一个正式的定义通常有两个方面的困难，既要避免太具体也要避免太抽象。如果一个定义太具体-如，定义某项技术-可能会随着UI技术的变化会变得模糊不清。作为一种学术定义而不是以常见的用法为基础的定义，它也必须足够一般，并且符合或者说广大的研究机构先前已发表在HCI的研究成果及艺术中符号学。另一方面，定义过于宽泛，也会有有无关紧要的风险：如果一切都是一种姿态，那么就什么都不是了。

Eric Hulteen 和Gord Kurtenbach关于手势的定义的中心在于手势能够用来交流，手势的意义在于讲述而不是执行。

有趣的是将语言和行为引入到人机交互接口中来，这是一种彻底的变革。我们与计算机交互语音变为无声的语言(mute)：我们通过指点和手势而不是语言与计算设备进行沟通。当和计算机进行交互时，我们点击键盘按键或触摸屏幕。我们似乎更喜欢这种形式的静音通信即使当前的技术能够支持更简单的语音指令。我们没有操作(manipulation)的力量，和虚拟的对象而不是真实的物体进行交互，因而没有持久性。运动成为纯粹的手势。

基于Eric Hulteen 和Gord Kurtenbach的定义，我们都明白什么是 UI 操作 ——暂时不是一种手势 ——理解什么是手势以及手势表示"重大"行为或者符号仍然有很大的困难。移动交互的含义是什么？手势进行沟通和语言进行沟通的最明显不同是什么？我们做手势的象征意义往往很抽象简单。

在人机交互领域，手势通常被作为传达一些简单的指令而不是交流某些事实、描述问题或者陈述想法。使用手势操作电脑通常是命令式的，这通常不是人们使用手势的目的。例如，挥手(wave)这一动作，在现实世界中通常是打招呼的一种方式，但是这种打招呼的方式在人机交互中却不太常用。通常第一次写程序通常会显示“hello”,但我们对和电脑打招呼并不感兴趣。

但是，在一个繁忙的餐馆，挥手这一手势可能就有不同的含义了。当向服务员招收时，可能是要引起服务员注意，需要他们提供服务。在计算机中，要引起计算机注意有时候也有其特殊意义，比如，计算机休眠时，一般都会敲击键盘或者移动鼠标来唤醒，以提醒计算机“注意”。当使用Kinect时，可以使用更加直观的方式，就行少数派报告阿汤哥那样，抬起双手，或者简单的朝计算机挥挥手，计算机就会从休眠状态唤醒。

在人机交互领域，手势通常有一些含义，表示有意让某些事情发生。手势是一种指令。当通过鼠标或者触控板去点击UI界面上的按钮时，我们希望按钮会触发其背后的事件。通常，按钮上会有一个标签来指示按钮的功能如：开始、取消、打开、关闭。我们的手势操作就是想要实现这些事件。

上面的定义中的第一点可以得出，手势的另一个特点是比较随意(arbitrary)。手势有限定的领域，那么在该领域之外没有任何意义。令人惊讶的是除了指向(pointing)和耸肩(shurg)，人类学家没有发现任何东西我们可以称之为一种通用的手势。然而，在计算机的UI中，指向(pointing)通常被认为是直接操作因为它牵涉到跟踪，同时耸肩的含义太微妙而不好辨识。因此，我们想要使用的任何Kinect手势必须基于应用程序的用户 和应用程序的设计和开发者之间就某种手势代表的含义达成一致。

因为手势是任意的(arbitrary)所以他们也是基于约定的(conventional)。应用程序的设计者必须告诉用户正在使用的手势的意义，或者是这些手势是约定俗称大家都知道的。此外，这些约定不是基于语言文化，而是对已确定的技术规则。我们知道如何使用鼠标 （行为学习） 并不是因为这是我们已经从我们的文化导入的东西，而是因为这是基于特定的图形用户界面的跨文化约定。同样地，我们知道如何点击或滑动智能手机，不是因为这些都是文化的约定，而是因为这些都是跨文化自然用户界面项约定。有趣的是，我们在一定程度上知道如何点击平板电脑，因为我们以前学习了如何使用鼠标单击。技术约定之间可以相互转化，这是因为语言和手势可以通过不同的语言和文化之间来转换。

然而，手势的这种任意性和基于约定的特性也带来了误解性（misunderstanding），这是在设计任何用户界面，尤其是像Kinect这样的没有任何预先设定好的操作约定的用户界面时需要关注的风险。就像在有些国家，点头表示否定摇头表示可能。手势，或者任何身体的运动，都有可能产生误解。

总之，在人机交互领域，手势是：

* 表达一种简单的命令
* 天生有随意性
* 基于某种协定
* 可能被误解

注意：实际的直接操作（manipulation）不是手势。

**2. 自然交互界面(NUI)**

讨论手势而不讨论自然用户界面显然不完整。自然用户界面是一系列技术的合计，他包括：语音识别，多点触控以及类似Kinect的动感交互界面，他和Windows和Macs操作系统中鼠标和键盘交互这种很常见图形交互界面不同。就像图像交互界面和之前的命名行交互界面不同那样。

自然交互界面自然在哪儿呢？早期自然交互界面的发起者认为交互界面的设计应该对用户非常直观，使用用户先天就会的行为来进行交互操作。他的目标是不需要操作由图标和菜单构成的基于GUI 的应用程序界面，因为这种界面通常具有陡峭的学习曲线。相反，理想化的状态是，用户应该能够走到应用程序前面，就能够开始使用它。在过去的几年里随着触摸功能的智能手机和平板电脑的流行，逐渐取代了键盘鼠标，当我们看到孩子们开始走到任何触摸屏设备面前，用手去触摸它，期待它的响应，在这一点上看这一理念已经实现。

虽然自然用户界面的自然性似乎是直接操作的最佳写照，当使用手指来进行触摸交互时，先天自然和后天学习行为之间的对立被打破。一些手势，如轻触屏幕，在某种意义上就是先天就会的动作。其他的动作比如说双击，获得点击然后拖拉等，没有先天就会。而且随着不同的设备制造商开始支持不同触摸手势，为了使得相同的手势在不同的触摸平台上有相同的意义和行为，为某些手势定义一些约定显得更加重要。

自然用户界面（NUI）的自然性更多的是一种相对自然的概念。对于NUI的更现代的理解受Bill Buxton所影响。他认为NUI界面的设计充分利用了用户预先就会的技能，用户和UI进行交互感到很自然，使得他们甚至忘了是从哪里学到这些和UI进行交互所需的技能的。换句话说，第一次操作时，我们不记得我们曾经学过这些知识。例如，轻点(tap)这个手势早平板电脑和手机中使用的很频繁，这个技能是从我们之前在传统的人机交互界面上使用鼠标来指向并点击某一个界面上的元素学来的。点击（click）和轻点（tap）的最主要区别在于，点击需要鼠标，对于触摸屏，不需要额外的设备，只需要用手指轻轻触摸一下屏幕就可以了。

这引出了自然用户界面的另一个特点。用户和计算机之间的交互看起来不需要任何媒介，这种相互作用的媒介是不可见的。例如，在语音识别界面中，人机交互是通过具有复杂电子过滤去噪的麦克风实现的，其内部有解析发音语义单元的各种算法，将这些语义传递给其它软件来进行将特定的短语解释为命令，并将该命令映射到某种软件功能操作。但是，内部的这一切，对用户是不可见的。当用户对计算机发出这样的命令，"嘿，注意我"，她会认为计算机会像类似大多数人的本能那样的响应这个命令。

自然用户界面的 依赖于先验知识和不需要媒介的交互这两个特征是每一种NUI界面的共同特征，其他方面如触摸，语音和动态交互界面则因设备的不同而各异。目前，大多数关于NUI的设计都是基于多点触控体验的。这就是为什么前面对于手势的标准定义是那样定义的。它是将多点触摸的场景进行修改并将手势和操作区分开来。

关于手势（gesture）和操作（manipulation）的争论也存在于语音交互界面中，命令等同于手势，语音等同于直接操作，在动态交互界面中，将手或者身体追踪展示在可视化界面上手和身体的运动等同于直接操作。自由形式的运动像挥手这一动作就属于手势。

但是Kinect还有第三种交互界面，他和触摸和语音交互不同。那就上一篇文章中所讲的姿势（pose），姿势是身体的某一部分和其他部分之间的一种静态关系，他不是运动的。Kinect中的姿势和日常生活中的姿势是一样的，例如，左臂伸出45度表示将当前的窗口变为活动的交互窗体，右臂伸出45度或者135度表示垂直滚动工具栏。

另外，交互方式可以从一种类型的交互界面转换到另外一种交互界面。以按钮为例，按钮其实就是一个符号，这是一个先验的图形用户界面。从最基本的功能来讲，按钮就是一个通过鼠标点击在一个可视化元素的文字或者图像上触发一些命令的工具。在过去15年，按钮被作为人机交互界面的一个集成部分，被转换到多点触摸界面，以及Kinect用户界面中来。

自然用户界面设计师所追求的是的是自然，按钮恰好提供了这一点。但是按钮在每一种用户界面中的转换都面临着一些挑战。

图形用户界面中按钮的一个通常的特征是他提供了一个悬浮状态来指示用户光标已经悬停在的按钮上方的正确位置。这种悬浮状态将点（click）这个动作离散开来。悬浮状态可以为按钮提供一些额外的信息。当将按钮移植到触摸屏交互界面时，按钮不能提供悬浮状态。触摸屏界面只能响应触摸。因此，和电脑上的图像用户界面相比，按钮只能提供“击”（click）操作，而没有“点”（point）的能力。

当将按钮移植到基于Kinect的用户界面上时，按钮的行为就变得更加特殊了。基于Kinect的图形界面中，按钮的行为和触摸界面中的刚好相反，他只提供了悬浮（hover）的“点”（point）的能力，没有“击”（click）的能力。按钮这种更令用户体验设计者感到沮丧的弱点，在过去的几年里，迫使设计者不断的对Kinect界面上的按钮进行改进，以提供更多巧妙的方式来点击视觉元素。这些改进包括：悬停在按钮上一段时间、将手掌向外推(笨拙地模仿点击一个按钮的行为）等。

虽然触摸界面也有手势，但Kinect 界面有些互动不是手势，不过软件的开发和设计者倾向于以 Kinect 手势操作作为交互界面。这似乎是因为使用手势作为物理操作是 Kinect 应用程序的最大的特点。与此相反的是，触摸界面的突出特点是直接操作。虽然可能不准确，人们通常将自然交互界面划分为三类：语音交互界面，触摸交互界面和手势交互界面。

然而，在关于Kinect的相关介绍文档中，你会发现有时候姿势(pose)和操作(manipulation)都被描述为手势。这些都没有错。要记住的是，当我们讨论Kinect中的一些术语，如挥手(wave),滑动(swipe),我们会作为纯粹的手势，而姿势和操控只有在隐喻意义上才称之为手势。

以上的讨论都很重要，因为我们会进一步设计Kinect互动的语意，我们将最终移除从其他图形界面上借鉴过来的关于按钮的语意，然后尝试建立基于Kinect的先验的语意。挥手(wave)这是Kinect中纯粹的手势，是最早的这种尝试。乔治亚技术研究所的研究人员正在利用 Kinect 来解释美国手语。相反，其他研究人员，正在利用 Kinect 解释身体语言——另一种预先形成的手势和姿势的沟通。诸如此类的研究可以视为对于NUI的第二层研究。这些逐渐接近了最初NUI人机交互的原始的梦想，不只是看不见，而且NUI能够自适应以理解我们的行为，而不是迫使我们了解我们和电脑的人机交互。

**3. 手势从哪里来**

在手势交互界面中，纯粹的手势，姿势和追踪以及他们之间的组合构成了交互的基本术语。对于Kinect来说，目前可以使用的有8个通用的手势：挥手(wave)，悬浮按钮(hover button),磁吸按钮(magnet button),推按钮(push button)，磁吸幻灯片(magnetic slide)，通用暂停(universal pause)，垂直滚动条(vertical scrolling)和滑动(swipping)。其中的一些术语是微软自己引入的，有一些是游戏代理商设计的，还有一些是Kinect for PC开发人员为了开发应用而引入的。

很少情况下会为人际交互界面术语进行定制。通常要将这8种手势区分开来，并在一些应用中通用也不常见。相似的情况在web术语和手机手势中设计新的界面时也会遇到，其中只有部分的设计能够变成标准。在网页设计领域，走马灯和光标动画流行一时，并在一片鄙夷声中迅速消失。在手机设计领域由于苹果公司在触摸屏领域的早期地位这种术语得到了很好的规范。苹果引入了一些触摸手势术语，如轻点（tap）,点住不放(tap and hold)，滑动swipe及pinch。

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204212305298864.png)

交互术语形成规范有几个障碍。第一个就是为了获得利益而避免标准化。在90年代后期的浏览器大战中，尽管各大厂商在口头上说标准化协议很重要，但是在浏览器开发上依旧不停的开发自己的HTML版本，以吸引开发者使用他们的技术。设备制造商可以利用市场占有率的优势来锁定消费者，通过在他们的手机上实现自己定义语意的触屏，来推行自己的手势操作。这些都是不自然的行为，因为不同厂商对于同一手势的语意都不同，并且他们看起来不自然，使用不同厂商的产品需要再学习。

另一种形成规范化的障碍是上下文手势的专利。例如，苹果公司不能对“滑动”(swipe)操作申请专利，但是它可以对“滑动解锁手机”这个手势申请专利，这使得其他公司需要使用这一技术或者设计理念时要么给苹果公司支付专利费，要么将苹果告上法庭以避免专利费，或则干脆不使用这一上下文手势。如果不使用这一上下文手势，那么产品就破坏了之前我们学习到使用很自然的方式滑动解锁手机，音乐播放器，平板电脑等这一约定了。

最后一个障碍是，设计一个手势很困难。手势术语会面对一些App Store中手机应用程序和YouTube中视频应用所遇到的一些问题：人们要么会要么不会。手势需要思考如何定义的简单使得人们能够去用，这就是长尾理论留下来的问题。

那么什么样的手势术语才是好的呢。如果一个手势易于使用，那么他就被认为是设计良好的。在交互设计中，易用性有两个方面：可用(affordance)和反馈(feedback)。反馈就是说用户知道当前正在进行的操作。在网页中,点击按钮会看到按钮有一点偏移,这就表示交互成功。鼠标按键按下时的声音在某种意义上也是一种反馈,他表示鼠标在工作。对于Winodw Phone Metro风格的界面上的磁贴，开发这认为这些按钮应该足够大，以容下大面积的触摸区域，但是他们也认为过大的触摸区域会使得用户触摸到区域外面也会触发注册的事件。另外，状态信息或者确认对话框会在应用程序中弹出以提示用户发生了一些事情。在 Xbox 的仪表板中，使用Kinect传感器产生的光标悬停在的热点上开始动画播放。

如果说反馈发生在操作进行中或者之后，那么可用性(affordance)就发生在操作之前了。可用性就是一种提示或者引导，告诉用户某一个可视化元素是可以交互的，指示用户该元素的用处。在GUI交互界面中，按钮是能够最好的完成这些理念的元素。按钮通过文字或者图标提示来执行一些函数操作。GUI界面上的按钮通过悬浮状态可以提示用户其用途。最好的可用性-可能有点绕圈-就是约定俗成。用户知道某一个可视化元素的用途，因为之前在其他应用中使用过类似的可视化控件，或者是在其他设备中执行过类似的操作。但是，这一点对于基于Kinect的手势交互界面来说有点困难，因为一切都是新的。

通常的做法就是使用借用其他类型交互界面中的约定。在触摸交互界面中，一个轻点(tap)手势和通常的鼠标点击是等同的。响应轻点事件的两个可视化元素，图标和按钮，也被设计的和传统的GUI界面上的图标和按钮一样，来达到提示用户该元素的作用这一目的。Kinect也使用按钮和图标来使得用户能够更加容易使用。因为Kinect基本上是基于”点”（pointing）而原生不支持“击”(clicking)。在此之前，软件界面设计者和开发者的花费了很多精力来对手势交互界面进行定制以实现“击”这一动作。

和触摸交互界面不一样，手势交互界面可以从社会中人的一般手势中借用一些手势操作。这就使得挥手(wave)成为Kinect应用程序的经典手势。因为这一姿势和现实生活中的姿势有象征性联系使得非常容易理解和使用。轨迹追踪，虽然在技术上不是手势，但是他是另一个在现实生活中和指点有联系的术语。当在电视机或者显示器前挥动手时，好的Kinect应用程序应该能够追踪到手的运动，并显示一个光标随着手一起运动。当我们在现实生活中指点物体时，Kinect中的手部追踪显示的手形图标的反馈使得程序更加易用。

目前，现实生活中的易用性手势在Kinect交互界面中用的比较少，大部分的易用性都是从传统的GUI界面上的可用性移植过来的。随着时间的改变，这一点会得到改善。在触摸屏设备上新的手势通过在传统的已经建立的约定中添加手指来形成。两指轻点和一指轻点有些不同，使用两个手指或者多个手指进行滑动有其独特的含义。最终，触摸手势全部由手指完成。另一方面，真正的手势用户界面，有一个近乎无限的语意库，使得我们可以基于现实生活中相关联的手势进行改进。

本文接下来从理论到实现，讨论如何实现手势识别，并展示了Kinect中八中基本手势中的挥手(wave)手势的识别。

**4. 实现手势识别**

Microsoft Kinect SDK并没有包含手势识别引擎。因此需要开发者来定义和手势识别。从SDK的Beta版放出以来，一些第三方开发者创建的手势引擎已初见端倪。但是，微软没有将他们作为标准的引擎。看来这可能还要等微软将手势识别引擎添加到SDK中来，或者指明可替代的解决方案。本节对手势识别技术进行了简单介绍，希望能够帮助开发者在标准的手势识别引擎出来之前，可以自己动手开发手势识别引擎。

手势识别相对来说可以简单也可以很复杂，这取决与要识别的手势。有三种基本的方法可以用来识别手势：基于算法，基于神经网络和基于手势样本库。每一种方法都有其优缺点。开发者具体采用那种方法取决与待识别的手势、项目需求，开发时间以及开发水平。基于算法的手势识别相对简单容易实现，基于神经网络和手势样本库则有些复杂。

**4.1 基于算法的手势识别**

算法是解决软件开发中几乎所有问题的最基本方法。使用算法的基本流程就是定义处理规则和条件，这些处理规则和条件必须符合处理结果的要求。在手势识别中，这种算法的结果要求是一个二值型对象，某一手势要么符合预定的手势要么不符合。使用算法来识别手势是最基本的方法，因为对于有一点编程能力的开发这来说，手势识别的代码易于理解，编写，维护和调试。

但是，最简单直接的方法也有其缺点。算法的简单性限制了其能识别到的手势的类别。对于挥手(wave)识别较好的算法不能够识别扔(throw)和摆(swing)动作。前者动作相对简单和规整，后者则更加细微且多变。可能能够写一个识别摆动(swing)的算法，但是代码可能比较费解和脆弱。

算法还有一个内在的扩展性问题。虽然一些代码可以重用，但是每一种手势必须使用定制的算法来进行识别。随着新的手势识别算法加入类库，类库的大小会迅速增加。这就对程序的性能产生影响，因为需要使用很多算法来对某一个手势进行识别以判断该手势的类型。

最后，每一个手势识别算法需要不同的参数，例如时间间隔和阈值。尤其是在依据流程识别特定的手势的时候这一点显得尤其明显。开发者需要不断测试和实验以为每一种算法确定合适的参数值。这本身是一个有挑战也很乏味的工作。然而每一种手势的识别有着自己特殊的问题。

**4.2 基于神经网络的手势识别**

当用户在做手势时，手势的形式并不总是足够清晰到能够判断用户的意图。例如跳跃手势，跳跃手势就是用户短暂的跳起来，脚离开地面。这个定义不能够提供足够的信息来识别这一动作。

咋一看，这个动作似乎足够简单，使得可以使用算法来进行识别。首先，考虑到有很多种不同形式的跳跃：基本跳跃（basic jumping）、 跨栏(hurdling)、 跳远(long jumping)、 跳跃(hopping)，等等。但是这里有一个大的问题就是，由于受到Kinect视场区域的限制，不可能总是能够探测到地板的位置，这使得脚部何时离开地板很难确定。想象一下，用户在膝盖到下蹲点处弯下，然后跳起来。手势识别引擎应该认为这是一个手势还是多个手势：下蹲或 下蹲跳起或者是跳起？如果用户在蹲下的时间和跳跃的时间相比过长，那么这一手势可能应被识别为下蹲而不是跳跃。

看到这些，最开始对跳跃的定义就会变得模糊。这一姿势很难定义清楚，使得不能够通过定义一些算法来进行识别，同时这些算法由于需要定义过多的规则和条件而变得难以管理和不稳定。使用对或错的二值策略来识别用户手势的算法太简单和不够健壮，不能够很好的识别出类似跳跃，下蹲等动作。

神经网络的组织和判断是基于统计和概率的，因此使得像识别手势这些过程变得容易控制。基于什么网络的手势识别引擎对于下蹲然后跳跃动作，80%的概率判断为跳跃，10%会判定为下蹲。

除了能够识别复杂和精细的手势，神经网络方法还能解决基于算法手势识别存在的扩展性问题。神经网络包含很多神经元，每一个神经元是一个好的算法，能够用来判断手势的细微部分的运动。在神经网络中，许多手势可以共享神经元。但是每一中手势识别有着独特的神经元的组合。而且，神经元具有高效的数据结构来处理信息。这使得在识别手势时具有很高的效率。

使用基于神经网络进行手势识别的缺点是方法本身复杂。虽然神经网络以及在计算机科学中对其的应用已经有了好几十年，建立一个好的神经网络对于大多数程序员来说还是有一些困难的。大多数开发者可能对数据结构中的图和树比较熟悉，而对神经网络中尺度和模糊逻辑的实现可能一点都不了解。这种缺乏建立神经网络的经验是一个巨大的困难，即使能够成功的构建一个神经网络，程序的调试相当困难。

和基于算法的方法相比，神经网络依赖大量的参数来能得到精确的结果。参数的个数随着神经元的个数增长。每一个神经元可以用来识别多个手势，每一个神经远的参数的变化都会影响其他节点的识别结果。配置和调整这些参数是一项艺术，需要经验，并没有特定的规则可循。然而，当神经网络配对机器学习过程中手动调整参数，随着时间的推移，系统的识别精度会随之提高。

**4.3 基于样本的识别**

基于样本或者基于模版的手势识别系统能够将人的手势和已知的手势相匹配。用户的手势在模板库中已经规范化了，使得能够用来计算手势的匹配精度。有两种样本识别方法，一种是存储一系列的点，另一种方法是使用类似的Kinect SDK中的骨骼追踪系统。在后面的那个方法中，系统中包含一系列骨骼数据和景深帧数据，能够使用统计方法对产生的影像帧数据进行匹配以识别出已知的帧数据来。

这种手势识别方法高度依赖于机器学习。识别引擎会记录，处理，和重用当前帧数据，所以随着时间的推移，手势识别精度会逐步提高。系统能够更好的识别出你想要表达的具体手势。这种方法能够比较容易的识别出新的手势，而且较其他两种方法能够更好的处理比较复杂的手势。但是建立这样一个系统也不容易。首先，系统依赖于大量的样本数据。数据越多，识别精度越高。所以系统需要大量的存储资源和CPU时间的来进行查找和匹配。其次系统需要不同高度，不同胖瘦，不同穿着（穿着会影响景深数据提取身体轮廓）的样本来进行某一个手势。

**5.识别常见的手势**

选择手势识别的方法通常是依赖于项目的需要。如果项目只需要识别几个简单的手势，那么使用基于算法或者基于神经网络的手势识别就足够了。对于其他类型的项目，如果有兴趣的话可以投入时间来建立可复用的手势识别引擎，或者使用一些人家已经写好的识别算法，接下来本文介绍几种常用的手势，并演示如何使用算法的方法来识别他们，手势识别的另外两种方法由于其复杂性本文不做介绍。

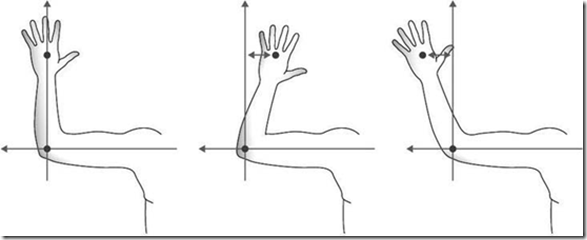
不论选择哪种手势识别的方法，都必须考虑手势的变化范围。系统必须具有灵活性，并允许某一个手势有某个范围内的变动。很少有人能够每次都做一模一样的手势。例如，考虑周伯通当前左右手画圆圈这个手势，重复这一手势10次，圆形的中心每次都在一个点吗，圆形的起点和重点每次都在相同的地方吗？每次画圆的时长都一样吗？然后使用右手做这个动作，最后比较结果。或者拉几个朋友或者家人来做，然后观察。也可以站在镜子前面看自己做，或者使用录像设备录下来再看。技巧就是对于某一手势，让尽可能多的人来做，然后试图标准化这一手势。手势识别一个比较好的方式就是关注手势最核心的部分而不是哪些外在的细枝末节。

**5.1 挥动(wave)手势**

只要玩过Xbox上的体感游戏，可能都使用过挥手这个手势。挥手这一手势不论年龄大小都能够做的一个简单动作。这是一个友好的，快乐的手势，人们通常挥手或者招手来打招呼或者道别。在应用开发的上下文中，挥手手势通常告诉应用程序已经准备好了，可以开始体验应用了。

挥手是最简单最基本的手势。使用算法方法能够很容易识别这一手势，但是之前讲到的任何方法也能够使用。虽然挥手是一个很简单的手势，但是如何使用代码来识别这一手势呢？读者可以在镜子前做向自己挥手，然后仔细观察手的运动，尤其注意观察手和胳膊之间的关系。继续观察手和胳膊之间的关系，然后观察在做这个手势事身体的整个姿势。有些人保持身体和胳膊的不动，使用手腕左右移动来挥手。有些人保持身体和胳膊不动使用手腕前后移动来挥手。可以通过观察这些姿势来了解其他各种不同挥手的方式。

XBOX中的挥手动作定义为：从胳膊开始到肘部弯曲。用户以胳膊肘为焦点来回移动前臂，移动平面和肩部在一个平面上，并且胳膊和地面保持平行,在手势的中部（下图1），前臂垂直于后臂和地面。下图展示了这一姿势。

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204212305292486.png)

从图中可以观察得出一些规律，第一个规律就是，手和手腕都是在肘部和肩部之上的，这也是大多是挥手动作的特征。这也是我们识别挥手这一手势的第一个标准。

第一幅图展示了挥手这一姿势的中间位置，前臂和后臂垂直。如果用户手臂改变了这种关系，前臂在垂直线左边或者右边，我们则认为这是该手势的一个片段。对于挥手这一姿势，每一个姿势片段必须来回重复多次，否则就不是一个完整的手势。这一运动规律就是我们的第二个准则：当某一手势是挥手时，手或者手腕，必须在中间姿势的左右来回重复特定的次数。使用这两点通过观察得到的规律，我们可以通过算法建立算法准则，来识别挥动手势了。

算法通过计算手离开中间姿势区域的次数。中间区域是一个以胳膊肘为原点并给予一定阈值的区域。算法也需要用户在一定的时间段内完成这个手势，否则识别就会失败。这里定义的挥动手势识别算法只是一个单独的算法，不包含在一个多层的手势识别系统内。算法维护自身的状态，并在识别完成时以事件形式告知用户识别结果。挥动识别监视多个用户以及两双手的挥动手势。识别算法计算新产生的每一帧骨骼数据，因此必须记录这些识别的状态。

下面的代码展示了记录手势识别状态的两个枚举和一个结构。第一个名为WavePosition的枚举用来定义手在挥手这一动作中的不同位置。手势识别类使用WaveGestureState枚举来追踪每一个用户的手的状态。WaveGestureTracker结构用来保存手势识别中所需要的数据。他有一个Reset方法，当用户的手达不到挥手这一手势的基本动作条件时，比如当手在胳膊肘以下时，可调用Reset方法来重置手势识别中所用到的数据。

private enum WavePosition

{

None = 0,

Left = 1,

Right = 2,

Neutral = 3

}

private enum WaveGestureState

{

None = 0,

Success = 1,

Failure = 2,

InProgress = 3

}

private struct WaveGestureTracker

{

public int IterationCount;

public WaveGestureState State;

public long Timestamp;

public WavePosition StartPosition;

public WavePosition CurrentPosition;

public void Reset()

{

IterationCount = 0;

State = WaveGestureState.None;

Timestamp = 0;

StartPosition = WavePosition.None;

CurrentPosition = WavePosition.None;

}

}

下面代码显示了手势识别类的最基本结构：它定义了五个常量：中间区域阈值，手势动作持续时间，手势离开中间区域左右移动次数，以及左手和右手标识常量。这些常量应该作为配置文件的配置项存储，在这里为了简便，所以以常量声明。WaveGestureTracker数组保存每一个可能的游戏者的双手的手势的识别结果。当挥手这一手势探测到了之后，触发GestureDetected事件。

当主程序接收到一个新的数据帧时，就调用WaveGesture的Update方法。该方法循环遍历每一个用户的骨骼数据帧，然后调用TrackWave方法对左右手进行挥手姿势识别。当骨骼数据不在追踪状态时，重置手势识别状态。

public class WaveGesture

{

private const float WAVE\_THRESHOLD = 0.1f;

private const int WAVE\_MOVEMENT\_TIMEOUT = 5000;

private const int LEFT\_HAND = 0;

private const int RIGHT\_HAND = 1;

private const int REQUIRED\_ITERATIONS = 4;

private WaveGestureTracker[,] \_PlayerWaveTracker = new WaveGestureTracker[6, 2];

public event EventHandler GestureDetected;

public void Update(Skeleton[] skeletons, long frameTimestamp)

{

if (skeletons != null)

{

Skeleton skeleton;

for (int i = 0; i < skeletons.Length; i++)

{

skeleton = skeletons[i];

if (skeleton.TrackingState != SkeletonTrackingState.NotTracked)

{

TrackWave(skeleton, true, ref this.\_PlayerWaveTracker[i, LEFT\_HAND], frameTimestamp);

TrackWave(skeleton, false, ref this.\_PlayerWaveTracker[i, RIGHT\_HAND], frameTimestamp);

}

else

{

this.\_PlayerWaveTracker[i, LEFT\_HAND].Reset();

this.\_PlayerWaveTracker[i, RIGHT\_HAND].Reset();

}

}

}

}

}

下面的代码是挥手姿势识别的主要逻辑方法TrackWave的主体部分。它验证我们先前定义的构成挥手姿势的条件，并更新手势识别的状态。方法识别左手或者右手的手势，第一个条件是验证，手和肘关节点是否处于追踪状态。如果这两个关节点信息不可用，则重置追踪状态，否则进行下一步的验证。

如果姿势持续时间超过阈值且还没有进入到下一步骤，在姿势追踪超时，重置追踪数据。下一个验证手部关节点是否在肘关节点之上。如果不是，则根据当前的追踪状态，挥手姿势识别失败或者重置识别条件。如果手部关节点在Y轴上且高于肘部关节点，方法继续判断手在Y轴上相对于肘关节的位置。调用UpdatePosition方法并传入合适的手关节点所处的位置。更新手关节点位置之后，最后判断定义的重复次数是否满足，如果满足这些条件，挥手这一手势识别成功，触发GetstureDetected事件。

private void TrackWave(Skeleton skeleton, bool isLeft, ref WaveGestureTracker tracker, long timestamp)

{

JointType handJointId = (isLeft) ? JointType.HandLeft : JointType.HandRight;

JointType elbowJointId = (isLeft) ? JointType.ElbowLeft : JointType.ElbowRight;

Joint hand = skeleton.Joints[handJointId];

Joint elbow = skeleton.Joints[elbowJointId];

if (hand.TrackingState != JointTrackingState.NotTracked && elbow.TrackingState != JointTrackingState.NotTracked)

{

if (tracker.State == WaveGestureState.InProgress && tracker.Timestamp + WAVE\_MOVEMENT\_TIMEOUT < timestamp)

{

tracker.UpdateState(WaveGestureState.Failure, timestamp);

System.Diagnostics.Debug.WriteLine("Fail!");

}

else if (hand.Position.Y > elbow.Position.Y)

{

//使用 (0, 0) 作为屏幕的中心. 从用户的视角看, X轴左负右正.

if (hand.Position.X <= elbow.Position.X - WAVE\_THRESHOLD)

{

tracker.UpdatePosition(WavePosition.Left, timestamp);

}

else if (hand.Position.X >= elbow.Position.X + WAVE\_THRESHOLD)

{

tracker.UpdatePosition(WavePosition.Right, timestamp);

}

else

{

tracker.UpdatePosition(WavePosition.Neutral, timestamp);

}

if (tracker.State != WaveGestureState.Success && tracker.IterationCount == REQUIRED\_ITERATIONS)

{

tracker.UpdateState(WaveGestureState.Success, timestamp);

System.Diagnostics.Debug.WriteLine("Success!");

if (GestureDetected != null)

{

GestureDetected(this, new EventArgs());

}

}

}

else

{

if (tracker.State == WaveGestureState.InProgress)

{

tracker.UpdateState(WaveGestureState.Failure, timestamp);

System.Diagnostics.Debug.WriteLine("Fail!");

}

else

{

tracker.Reset();

}

}

}

else

{

tracker.Reset();

}

}

下面的代码添加到WaveGestureTracker结构中：这些帮助方法维护结构中的字段，使得TrackWave方法易读。唯一需要注意的是UpdatePosition方法。TrackWave调用该方法判断手的位置已经移动。他的最主要目的是更新CurrentPosition和Timestamp属性，该方法也负责更新InterationCount字段合InPorgress状态。

public void UpdateState(WaveGestureState state, long timestamp)

{

State = state;

Timestamp = timestamp;

}

public void Reset()

{

IterationCount = 0;

State = WaveGestureState.None;

Timestamp = 0;

StartPosition = WavePosition.None;

CurrentPosition = WavePosition.None;

}

public void UpdatePosition(WavePosition position, long timestamp)

{

if (CurrentPosition != position)

{

if (position == WavePosition.Left || position == WavePosition.Right)

{

if (State != WaveGestureState.InProgress)

{

State = WaveGestureState.InProgress;

IterationCount = 0;

StartPosition = position;

}

IterationCount++;

}

CurrentPosition = position;

Timestamp = timestamp;

}

}

上述代码片段就可以实现挥动(wave)手势识别的逻辑了。

**6. 结语**

本文主要介绍了手势识别中设计的基本概念以及手势识别的发展过程，在此基础上介绍了手势识别的三种基本方法:基于算法的手势识别、基于神经网络的手势识别和基于样本库的手势识别。最后以8中常用手势的中挥手(wave)介绍了使用算法进行手势识别的基本步骤，并给出了相应的代码，限于篇幅原因，就先介绍到这里，下篇文章，将会介绍手势识别中手部捕捉类库和余下的7中手势的识别方法，由于本文大部分讲述的是概念，所以本部分源代码将和下篇文章中的源代码一同发布，敬请期待！

希望本文对您熟悉Kinect SDK中的手势识别有所帮助！