[**Kinect for Windows SDK开发入门(八)：骨骼追踪进阶 上**](http://www.cnblogs.com/yangecnu/archive/2012/04/13/KinectSDK_AdvanceSkeletonTracking_part1.html)

前7篇文件我们介绍了Kinect SDK中各种传感器的各种基本知识，我们用实验的方式演示了这些基本对象和方法的如何使用，这些都是Kinect开发最基本的知识。了解了这些基本知识后，就可以开发出一个基于Kinect的简单程序了。但是这些离开发出一个好的基于Kinect的应用程序还有一段距离。**后面的文章中，将会结合Kinect SDK介绍WPF以及其它第三方工具，类库来建立一个以Kinect为驱动的有较好用户体验的程序。我们将利用之前讲到的知识来进行下面一些比较复杂的话题。**

**Kinect传感器核心只是发射红外线，并探测红外光反射，从而可以计算出视场范围内每一个像素的深度值。*从深度数据中最先提取出来的是物体主体和形状，以及每一个像素点的游戏者索引信息。然后用这些形状信息来匹配人体的各个部分，最后计算匹配出来的各个关节在人体中的位置。这就是我们之前介绍过的骨骼追踪。***

红外影像和深度数据对于Kinect系统来说很重要，它是Kinect的核心，在Kinect系统中其重要性仅次于骨骼追踪。事实上，这些数据相当于一个输入终端。随着Kinect或者其他深度摄像机的流行和普及。开发者可以不用关注原始的深度影像数据，他们变得不重要或者只是作为获取其他数据的一个基础数据而已。我们现在就处在这个阶段，Kinect SDK并没有提供给开发者访问原始红外影像数据流的接口，但是其它第三方的SDK可以这么做。**可能大多数开发者不会使用原始的深度数据，用到的只是Kinect处理好了的骨骼数据。但是，一旦姿势和手势识别整合到Kinect SDK并成为其一部分时，可能开发者甚至不用接触到骨骼数据了。**

希望能够早日实现这种集成，因为它代表这Kinect作为一种技术的走向成熟。本篇文章和下篇文章仍将讨论骨骼追踪，但是采用不同的方法来处理骨骼数据。**我们将Kinect作为一个如同鼠标，键盘或者触摸屏那样的一个最基本的输入设备**。**微软当初推出Kinect for Xbox的口号是“你就是控制器”，从技术方面讲，就是“你就是输入设备”。**通过骨骼数据，应用程序可以做鼠标或者触摸屏可以做的事情，所不同的是深度影像数据使得用户和应用程序可以实现以前从没有过的交互方法。下面来看看Kinect控制并与用户界面进行交互的机制吧。

**1. 用户交互**

运行在电脑上的应用程序需要输入信息。传统的信息来自于鼠标或者键盘等这些输入设备。用户直接与这些硬件设备进行交互，然后硬件设备响应用户的操作，将这些操作转换成数据传输到计算机中。计算机接收这些输入设备的信息然后将结果以可视化的形式展现出来。大多数计算机的图像用户界面上会有一个光标(Cursor)，他通常代表鼠标所在的位置，因为鼠标是最开始有个滚轮设备。但是现在，如果将这个光标指代鼠标光标的话，可能不太准确，因为现在一些触摸板或手写设备也能像鼠标那样控制光标。当用户移动鼠标或者在触摸板上移动手指时，光标也能响应这种变化。当用户将光标移动到一个按钮上时，通常按钮的外观会发生变化，提示用户光标正位于按钮上。当用户点击按钮时，按钮则为显示另一种外观。当用户松开鼠标上的按键，按钮就会出现另外一种外观。显然，简单的点击事件会涉及到按钮的不同状态。

开发者可能对这些交互界面和操作习以为常，因为诸如WPF之类的用户交互平台使得程序与用户进行交互变得非常简单。当开发网页程序时，浏览器响应用户的交互，开发者只需要根据用户鼠标的悬停状态来设置样式即可进行交互。但是Kinect不同，他作为一个输入设备，并没有整合到WPF中去，因此，作为一个开发者。对操作系统和WPF所不能直接响应的那部分工作需要我们来完成。

在底层，鼠标，触摸板或者手写设备都是提供一些X,Y坐标，操作系统将这些X，Y坐标从其在的空间坐标系统转换到计算机屏幕上，这一点和上篇文章讨论的空间变换类似。操作系统的职责是响应这些标准输入设备输入的数据，然后将其转换到图形用户界面或者应用程序中去。操作系统的图形用户界面显示光标位置，并响应用户的输入。在有些时候，这个过程没有那么简单，需要我们了解GUI平台。以WPF应用程序为例，它并没有对Kinect提供像鼠标和键盘那样的原生的支持。这个工作就落到开发者身上了，我们需要从Kinect中获取数据，然后利用这些数据与按钮，下拉框或者其他控件进行交互。根据应用程序或者用户界面的复杂度的不同，这种工作可能需要我们了解很多有关WPF的知识。

**1.1 WPF应用程序中输入系统介绍**

当开发一个WPF应用程序时，开发者并不需要特别关注用户输入机制。WPF会为我们处理这些机制使得我们可以关注于如何响应用户的输入。毕竟作为一个开发者，我们更应该关心如何对用户输入的信息进行分析处理，而不是重新造轮子来考虑如何去收集用户的输入。如果应用程序需要一个按钮，只需要从工具箱中拖一个按钮出来放在界面上，然后在按钮的点击事件中编写处理逻辑即可。在大多数情况下，开发者可能需要对按钮设置不同的外观以响应用户鼠标的不同状态。WPF会在底层上为我们实现这些事件，诸如鼠标何时悬停在按钮上，或者被点击。

WPF有一个健全的输入系统来从输入设备中获取用户的输入信息，并响应这些输入信息所带来的控件变化。这些API位于System.Windows.Input命名空间中(Presentation.Core.dll)，这些API直接从操作系统获取输入设备输入的数据，例如，名为Keyboard，Mouse，Stylus，Touch和Cursor的这些类。InputManager这个类负责管理所有输入设备获取的信息，并将这些信息传递到表现框架中。

WPF的另一类组件是位于System.Windows命名空间(PresentationCore.dll)下面的四个类，他们是UIElement，ContentElement，FrameworkElement以及FrameworkContentElement 。FrameworkElement继承自UIElement，FrameworkContentElement继承自ContentElement。这几个类是WPF中所有可视化元素的基类，如Button，TextBlock及ListBox。更多WPF输入系统相关信息可以参考[MSDN文档](http://msdn.microsoft.com/en-us/library/ms754010.aspx)。

InputManager监听所有的输入设备，并通过一系列方法和事件来通知UIElement和ContentElement对象，告知这些对象输入设备进行了一些有关可视化元素相关的操作。例如，在WPF中，当鼠标光标进入到可视化控件的有效区域时就会触发MouseEnterEvent事件。UIElement和ContentElement对象也有OnMouseEnter事件。这使得任何继承自UIElement或者ContentElement类的对象也能够接受来自输入设备的所触发的事件。WPF会在触发任何其它输入事件之前调用这些方法。在UIElement和ContentElement类中也有一些类似的事件包括MouseEnter，MouseLeave，MouseLeftButtonDown，MouseLeftButtonUp，TouchEnter，TouchLeave，TouchUp和TouchDown。

有时候开发者需要直接访问鼠标或者其他输出设备，InputManager对象有一个称之为PrimaryMouseDevice的属性。他返回一个MouseDevice对象。使用MouseDevice对象，能够在任何时候通过调用GetScreenPositon来获取鼠标的位置。另外，MouseDevice有一个名为GetPositon的方法，可以传入一个UI界面元素，将会返回在该UI元素所在的坐标空间中的鼠标位置。当需要判断鼠标悬停等操作时，这些信息尤其重要。当Kinect SDK每一次产生一幅新的SkeletonFrame帧数据时，我们需要进行坐标空间转换，将关节点位置信息转换到UI空间中去，使得可视化元素能够直接使用这些数据。当开发者需要将鼠标作为输入设备时， MouseDevice对象中的GetScreenPositon和GetPosition方法能提供当前鼠标所在点的位置信息。

在有些情况下，Kinect虽然和鼠标相似，但是某些方面差别很大。骨骼节点进入或者离开UI上的可视化元素这一点和鼠标移入移出行为类似。换句话说，关节点的悬停行为和鼠标光标一样。但是，类似鼠标点击和鼠标按钮的按下和弹起这些交互，关节点与UI的交互是没有。在后面的文章中，可以看到使用手可以模拟点击操作。在Kinect中相对于实现鼠标移入和移出操作来说，对鼠标点击这种支持相对来说较弱。

Kinect和触摸板也没有太多相同的地方。触摸输入可以通过名为Touch或者TouchDevice的类来访问。单点的触摸输入和鼠标输入类似，然而，多点触控是和Kinect类似的。鼠标和UI之间只有一个交互点（光标）但是触摸设备可以有多个触控点。就像Kinect可以有多个游戏者一样。从每一个游戏者身上可以捕捉到20个关节点输入信息。Kinect能够提供的信息更多，因为我们知道每一个输入点是属于游戏者身体的那个部位。而触控输入设备，应用程序不知道有多少个用户正在触摸屏幕。如果一个程序接收到了10个输入点，无法判断这10个点是一个人的10个手指还是10个人的一个手指触发的。 虽然触控设备支持多点触控，但这仍然是一种类似鼠标或者手写板的二维的输入。然而，触控输入设备除了有X,Y点坐标外，还有触控接触面积这个字段。毕竟，用户用手指按在屏幕上没有鼠标光标那样精确，触控接触面积通常大于1个像素。

当然，他们之间也有相似点。Kinect输入显然严格地符合WPF 所支持的任何输入设备的要求。除了有其它输入设备类似的输入方式外，他有独特的和用户进行交互的方式和图形用户界面。核心上，鼠标，触控板和手写板只传递一个像素点位置嘻嘻你。输入系统确定该点在可见元素上下文中的像素点位置，然后这些相关元素响应这个位置信息，然后进行响应操作。

期望是在未来Kinect能够完整的整合进WPF。在WPF4.0中，触控设备作为一个单独的模块。最开始触控设备被作为微软的Surface引入。Surface SDK包括一系列的WPF控件，诸如SurfaceButton，SurfaceCheckBox，和SurfaceListBox。如果你想按钮能够响应触摸事件，最好使用SurfaceButton控件。

能够想象到，如果Kinect被完整的整合进WPF，可能会有一个称之为SkeletonDevice的类。他和Kinect SDK中的SkeletonFrame对象类似。每一个Skeleton对象会有一个称之为GetJointPoint的方法，他和MouseDevice的GetPositon和TouchDevice的GetTouchPoint类似。另外，核心的可视化元素(UElement, ContentElement, FrameworkElement, FrameworkContentElement) 有能够相应的事件或者方法能够通知并处理骨骼关节点交互。例如，可能有一个JointEnter，JointLeave，和JointHover事件。更进一步，就像触控类有一个ManipulationStarted和ManipulationEnded事件一样，在Kinect输入的时候可能伴随GetstureStarted和GestureEnded事件。

目前，Kinect SDK和WPF是完全分开的，因此他和输入系统没有在底层进行整合。所以作为开发者的我们需要追踪骨骼关节点位置，并判断节点位置是否和UI界面上的元素有交互。当关节点在对应的UI坐标系可视化界面的有效范围内时，我们必须手动的改变这些可视化元素的外观以响应这种交互。

**1.2 探测用户的交互**

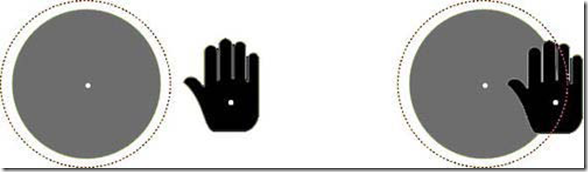
在确定用户是否和屏幕上的某一可视化元素进行交互之前，我们必须定义什么叫用户和可视化元素的交互。在以鼠标或者光标驱动的应用程序中有两种用户交互方式。鼠标悬停和点击交互。这些将事件划分为更精细的交互。就拿光标悬停来说，它必须进行可视化组件的坐标空间区域，当光标离开这一区域，悬停交互也就结束了。在WPF中，当用户进行这些操作时，会触发MouseEnter和MouseLeave操作。

除了点击和悬停外，鼠标还有另外一种常用的交互，那就是拖放。当光标移动到可视化组件上方，按下鼠标左键，然后在屏幕上拖动，我们称之为拖动(drag)，当用户松开鼠标左键时，我们之位释放操作(drop)。鼠标拖动和释放是一个比较复杂的交互，这点和Kinect中的手势类似。

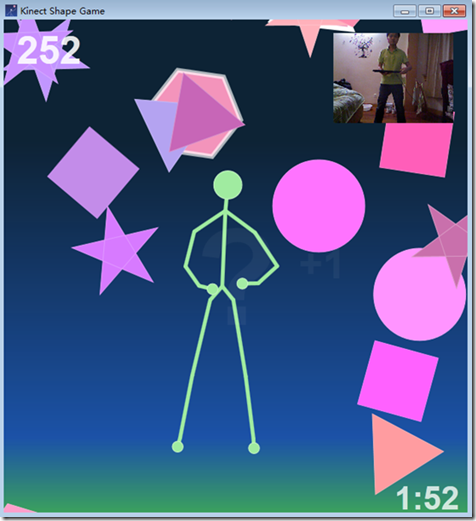
本节我们来看一下一些简单的诸如光标悬停，进入，离开可视化控件的交互。在[前篇文章](http://www.cnblogs.com/yangecnu/archive/2012/04/09/Kinect-SDK_Skeleton_Tracking_Part2.html)中的Kinect连线小游戏中，我们在绘制直线时需要判断手是否在点的合适范围内。在那个小游戏中，应用程序并没有像用户界面和人那样直接响应用户界的操作。这种差别很重要。应用程序在屏幕坐标空间中产生一些点的位置(数字)，但是这些点并没有直接从屏幕空间派生。这些点只是存储在变量中的数据而已。我们改变屏幕大小使得很容易展现出来。在接收到新的骨骼数据帧之前。骨骼数据中手的位置被转换到屏幕中点所在的空间坐标系，然后我们判断手所在的位置的点是否在点序列中。技术上来讲，这个应用程序即使没有用户界面也能够正常运行。用户界面是动态的由这些数据产生的。用户直接和这些数据而不是和界面进行交互。

**1.2.1命中测试(Hit testing)**

判断用户的手是否在点的附近远没有判断手是否在点的位置上那么简单。每一个点只是一个象元。为了使得应用程序能够工作。我们并不要求手的位置敲好在这个点所在的象元上，而是要求在以这个点为中心的某一个区域范围内。我们在点的周围创建了一个圆圈代表点的区域范围，用户的手的中心必须进入到这个点的区域范围才被认为是悬停在该点上。如图所示在圆形中的白色的点是实际的点，虚线绘制的圆形是该点的最大可触及范围。手形图标的中心用白色的点表示。所以，有可能手的图标和点的最大范围接触了，但是手的中心却不在该点的最大范围内。判断手的中心是否在点的最大范围之内称之为命中测试。

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204131225245394.png)

在Kinect连线游戏中，用户界面响应数据，依据产生的坐标将点绘制在图像界面上，系统使用点而不是用可视化控件的有效区间来进行命中测试。大多数的应用程序和游戏都不是这样做的。用户界面通常很复杂，而且是动态的。例如在Kinect for Windows SDK中自带的ShapeGame应用就是这样一个例子，它动态的从上至下产生一些形状。当用户触碰这些形状时形状会消失或者弹开。

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204131225257096.png)

ShapeGame这个应用比之前的Kinect连线游戏需要更为复杂的命中测试算法。WPF提供了一些工具来帮助我们实现命中测试。在System.Windows.Media命名空间下的VisualTreeHelper帮助类中有一个HitTest方法。这个方法有很多个重载，但是最基本的方法接受两个参数，一个是可视化控件对象，另一个是待测试的点。他返回可视化对象树中该点所命中的最顶层的那个可视化对象。听起来可能有点复杂，一个最简单的解释是，在WPF中有一个分层的可视化输出，有多个对象可能占据同一个相对空间，但是在不同的层。如果该点所在位置有多个对象，那么HitTest返回处在可视化树中处在最顶层的可视化对象。由于WPF的样式和模板系统使得一个控件能够由一个或者多个元素或者其它控件组成，所在通常在一个点可能有多个可视化元素。

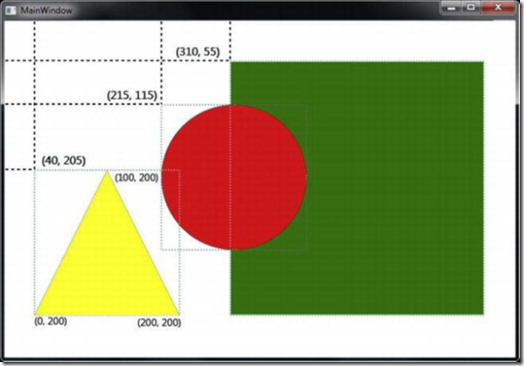
[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204131225273150.png)

上图可能帮助我们理解可视元素的分层。图中有三个元素：圆形，矩形和按钮。所有三个元素都在Canvas容器中。圆形和按钮在矩形之上，左边第一幅图中，鼠标位于圆形之上，在这点上的命中测试结果将返回这个圆形。第二幅图，即使矩形最底层，由于鼠标位于矩形上，所以命中测试会返回矩形。这是因为矩形在最底层，他是唯一个占据了鼠标光标象元所在位置的可视化元素。在第三幅图中，光标在按钮的文字上，命中测试将返回TextBlock对象，如果鼠标没有位于按钮的文字上，命中测试将会返回ButtonChrome元素。按钮的可视化表现通常由一个或者多个可视化控件组成，并能够定制。实际上，按钮没有继承可视化样式，它是一个没有可视化表现的控件。上图中的按钮使用的是默认样式，它由TextBlock和ButtonChrome这两个控件构成的。在这个例子中，我们通常会获得到有按钮样式组成的元素，但是永远获取不到实际的按钮控件。

为了使得命中测试更为方便，WPF提供了其他的方法来协助进行命中测试。UIElement类定义了一个InputHitTest方法，它接受一个Point对象，并返回该Point对象指定的一个IIputElement元素。UIElement和ContentElement两个类都实现了IInputElement接口。这意味着所有的WPF用户界面元素都实现了这个接口。VisualTreeHelper类中的HitTest方法可以用在一般的场合。

Note: MSDN中关于UIElement.InputHitTest方法的建议“应用程序一般不需要调用该方法，只有应用程序需要自己重新实现一系列已经实现了的底层输入特征，例如要重新实现鼠标设备的输入逻辑时才会去调用该方法。”由于Kinect并没有原生的集成到WPF中，所以必须重新实现类似鼠标设备的输入逻辑。

WPF中，命中测试依赖于两个变量，一个是可视化元素，另一个是点。测试首先该点转换到可视化元素所在坐标空间，然后确定是否处于该可视化元素的有效范围内。下图可以更好的理解可视化元素的坐标空间。WPF中的每一个可视化元素，不论其形状和大小，都有一个外轮廓：这个轮廓是一个矩形，它包含可视化元素并定义了可视化元素的宽度和高度。布局系统使用这个外轮廓来确定可视化元素的整体尺寸以及如何将其排列在屏幕上。当开发者使用Canvas，Grid，StackPanel等容器来布局其子元素时，元素的外轮廓是这些容器控件如进行布局计算的基础。用户看不到元素的外轮廓，下图中，可视化元素周围的虚线矩形显示了这些元素的外轮廓。此外，每一个元素有一个X,Y坐标用来指定该元素在其父容器中的位置。可以通过System.Windows.Controls.Primitives命名空间中的LayoutInformation静态类中的GetLayoutSlot方法来获取元素的外轮廓和其位置。举例来说，图中三角形的外轮廓的左上角坐标点为（0,0），三角形的宽和高都是200像素。所以在三角形外轮廓中，三角形的三个点的坐标分别为(100,0),(200,200),(0,200)。并不是在三角形外轮廓中的所有点在命中测试中都会成功，只有在三角形内部的点才会成功。点(0,0)不会命中，而三角形的中心(100,100)则能命中。

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204131225308148.png)

命中测试的结果依赖于可视化元素的布局。在目前所有的项目中，我们使用Canvas容器来包含所有可视化元素。Canvas是一个可视化的容器，能够使得开发者对可视化元素的位置进行完全控制，这一点在使用Kinect的时候尤其明显。像手部跟踪这类基本的方法也可以使用WPF中的其他容器，但是需要做很多其他工作，并且性能没有使用Canvas好。使用Cnavas容器，用户可以通过CanvasLeft和CanvasTop显式设定其所有子元素的起始X,Y的位置。前面讨论的坐标空间转换使用Cnavas作为容器，因为不需要太多的处理操作，转换也非常明了，只需要少量的代码就可以实现较好的性能。

使用Canvas作为容器的缺点也是其的优点。由于开发者可以完全控制在Canvas中子元素的位置，所以当窗体大小发生改变或者有比较复杂的布局时，也需要开发者去更新这些可视化元素的位置。而另外一些容器控件，如Grid，StackPanel则会帮助我们实现这些更新操作。但是，这些容器控件增加了可视化树的结构和坐标空间，从而增加了命中测试的复杂度。坐标空间越多，需要的点的转换就越多。这些容器还有alignment属性（水平和垂直）和相对于FrameworkElement的margin属性，进一步增加了命中测试的计算复杂度。如果可是化元素有RenderTransforms方法的话，我们可以直接使用这些方法而不用去自己写命中测试的算法了。

一个折中的方法是，将那些基于骨骼节点位置的需要频繁变化的可视化元素，如手形图标放在Canvas容器内，而将其他UI元素放在其他容器控件内。这种布局模式需要多个坐标空间转换，会影响程序性能，并且在进行坐标空间转换计算时可能会引入一些bug。这种混合的布局方案在大多数情况下是最好的选择，它充分利用了WPF布局系统的优点。要详细了解各种容器及其命中测试的相关概念，可以参阅MSDN中WPF的布局系统。

**1.2.2响应输入**

命中测试只能告诉当前用户输入点是否在可视化元素的有效区间内。用户界面最重要的一个功能是要给予用户一些对输入操作的反馈。当鼠标移到一个按钮上时，我们期望按钮能够改变其外观，告诉我们这个按钮是可以点击的。如果没有这种反馈，用户不仅用户体验不好，而且还会使用户感到迷惑。有时候即使功能完备，用户体验失败意味着应用的失败。

WPF有一套功能强大的系统来通知和响应用户的输入。只要用户的输入设备是鼠标，手写笔，触摸板这些标准设备，WPF的样式和模版系统使得开发出能够响应用户输入的高度的定制化的用户界面非常容易。而Kinect的开发者有两种选择：不使用WPF系统提供的功能，手动实现所有功能，或者创建一个通用的控件来响应Kinect输入。第二种方法虽然不是特别难，但是初学者也不太容易能够实现。

了解了这一点，在下面的章节中，我们将会开发一个游戏来使用命中测试并手动响应用户的输入。在开始之前，思考一个问题，到目前位置，还有那些问题没有很好解决？使用Kinect骨骼数据和用户界面进行交互是什么意思？核心的鼠标交互有：进入，离开和点击。触摸输入交互有进入，离开，按下，弹起。鼠标只有一个触控点，触摸版可以有多个触控点，但是只有一个是主触控点。Kinect骨骼节点数据有20个可能的数据点，哪一个点是主触控点？应该有一个主控点吗？一个可视化元素，比如说按钮，会在任何一个关节点数据到达按钮的有效范围内触发，还是只是特定的关节点数，比如手，进入范围后才能触发？

没有一个回答能够完全回答好上面的问题。这取决于应用程序界面的设计及要实现的功能。这些问题其实是自然交互界面设计中的一部分典型问题。在后面我们会介绍。对于大多数Kinect应用程序，包括本文中的例子，只允许手部关节点数据才能和用户界面进行交互。最开始的交互是进入和离开。除此之外的交互可能会很复杂。在后面我们将介绍这些复杂的交互，现在让我们来看看最基本的交互。

**2. “我说你做”游戏**

为了演示如何将Kinect作为一个输入设备，我们开始开发我们的项目：该项目使用手部关节点数据模仿鼠标或者触控板和用户界面进行交互。这个项目的目标是展示如何进行命中测试和使用WPF可视化元素来创建用户界面。项目是一个称之为“我说你做”(Simon Say)的小游戏。

“我说你做”（[Simon says](http://baike.baidu.com/view/2171156.htm)）是一个英国传统的儿童游戏。一般由3个或更多的人参加。其中一个人充当"Simon"。其他人必须根据情况对充当"Simon"的人宣布的命令做出不同反应。如果充当"Simon"的人以"Simon says"开头来宣布命令，则其他人必须按照命令做出相应动作。如：充当"Simon"的人说："Simon says jump（跳）"。其他人就必须马上跳起；而如果充当"Simon"的人没有说"Simon says"而直接宣布命令，如：充当"Simon"的人说"jump"。则其他人不准有动作，如果有动作则做动作的人被淘汰出游戏。

在70年代末80年代初有一个叫Milton Bradley的游戏公司开发了一个电子版的Simon say游戏。该游戏界面由4个不同颜色 (红色，蓝色，绿色，黄色) 的按钮组成，这个游戏在电脑上运行，让游戏者按演示的顺序按下这些按钮。当开始游戏时，程序首先按照一定的顺序亮起每一个按钮，游戏者必须按照这个亮灯的顺序依次按下这些按钮。如果游戏者操作正确，那么下一个亮灯序列又开始，到后面变化会越来越快，直到游戏者不能够按照给定的顺序按下这些按钮位置。

我们要做的是，使用Kinect设备来实现这么一个Simon Say游戏。这是个很好的使用Kinect展示如何和用户界面进行交互的例子。这个游戏也有一些规则。下图展示了我们将要做的用户界面，他包含四个矩形，他用来模拟游戏中的按钮。界面上方是游戏标题，中间是游戏的操作指南。

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204131225329775.png)

这个Kinect版的Simon says游戏追踪游戏者的手部关节。当用户的手碰到了这四个填充了颜色的方框中的任何一个时，程序认为游戏者按下了一个按钮。在Kinect应用程序中，使用悬停或者点击来和按钮进行交互很常见。现在，我们的游戏操作指南还很简单。游戏一开始，我们提示用户将手放在界面上红色矩形中手势图标所在的位置。在用户将双手放到指定位置后，界面开始发出指令。如果游戏者不能够重复这个过程，游戏将会结束，并返回到这个状态。现在，我们对这个游戏的概念，规则和样子有了一些了解，现在开始编码。

**2.1 Simon say “设计一个用户界面”**

首先来设计一个用户界面，下面的代码展示的主界面中的XAML和之前的连线游戏一样，我们将所有的主界面的UI元素包含在Viewbox容器中，让他来帮助我们进行不同显示器分辨率下面的缩放操作。主UI界面分辨率设置为1920\*1080。UI界面共分为4个部分：标题及游戏指导，游戏界面，游戏开始界面以及用来追踪手部的手形图标。第一个TextBlock用来显示标题，游戏引导放在接下来的StackPanel元素中。这些元素是用来给游戏者提供当前游戏状态。他们没有功能性的作用，和Kinect或者骨骼追踪没有关系。

GameCanvas，ControlCanvas和HandCanvas包含了所有的和Kienct相关的UI元素，这些元素是基于当前用户手的位置和用户界面进行交互的。手的位置来自骨骼追踪。HandCanvas应该比较熟悉，程序中有两个手形图标，用来追踪游戏者两只手的运动。ControlCanvas存储的UI元素用来触发开始游戏。GameCanvas用来存储这4个矩形，在游戏中，用户需要点击这些矩形。不同的交互元素存储在不同的容器中，使得用户界面能够比较容易使用代码进行控制。比如，当用户开始游戏后，我们需要隐藏所有的ControlCanvas容器内的子元素，显然隐藏这个容器比隐藏其每个子控件容易的多。整个UI代码如下：

<Window x:Class="KinectSimonSay.MainWindow"

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

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

xmlns:c="clr-namespace:KinectSimonSay"

Title="MainWindow" WindowState="Maximized">

<Viewbox>

<Grid x:Name="LayoutRoot" Height="1080" Width="1920" Background="White" TextElement.Foreground="Black">

<c:SkeletonViewer x:Name="SkeletonViewerElement"/>

<TextBlock Text="Simon Say" FontSize="72" Margin="0,25,0,0" HorizontalAlignment="Center" VerticalAlignment="Top"></TextBlock>

<StackPanel HorizontalAlignment="Center" VerticalAlignment="Center" Width="600">

<TextBlock x:Name="GameStateElement" FontSize="55" Text=" GAME OVER!" HorizontalAlignment="Center" />

<TextBlock x:Name="GameInstructionsElement" Text="将手放在对象上开始游戏。" FontSize="45" HorizontalAlignment="Center"

TextAlignment="Center" TextWrapping="Wrap" Margin="0,20,0,0" />

</StackPanel>

<Canvas x:Name="GameCanvas">

<Rectangle x:Name="RedBlock" Height="400" Width="400" Fill="Red" Canvas.Left="170" Canvas.Top="90" Opacity="0.2" />

<Rectangle x:Name="BlueBlock" Height="400" Width="400" Fill="Blue" Canvas.Left="170" Canvas.Top="550" Opacity="0.2" />

<Rectangle x:Name="GreenBlock" Height="400" Width="400" Fill="Green" Canvas.Left="1350" Canvas.Top="550" Opacity="0.2" />

<Rectangle x:Name="YellowBlock" Height="400" Width="400" Fill="Yellow" Canvas.Left="1350" Canvas.Top="90" Opacity="0.2" />

</Canvas>

<Canvas x:Name="ControlCanvas">

<Border x:Name="RightHandStartElement" Background="Red" Height="200" Padding="20" Canvas.Left="1420" Canvas.Top="440" >

<Image Source="Images/hand.png" />

</Border>

<Border x:Name="LeftHandStartElement" Background="Red" Height="200" Padding="20" Canvas.Left="300" Canvas.Top="440" >

<Image Source="Images/hand.png" >

<Image.RenderTransform>

<TransformGroup>

<TranslateTransform X="-130" />

<ScaleTransform ScaleX="-1" />

</TransformGroup>

</Image.RenderTransform>

</Image>

</Border>

</Canvas>

<Canvas x:Name="HandCanvas">

<Image x:Name="RightHandElement" Source="Images/hand.png" Visibility="Collapsed" Height="100" Width="100" />

<Image x:Name="LeftHandElement" Source="Images/hand.png" Visibility="Collapsed" Height="100" Width="100" >

<Image.RenderTransform>

<TransformGroup>

<ScaleTransform ScaleX="-1" />

<TranslateTransform X="90" />

</TransformGroup>

</Image.RenderTransform>

</Image>

</Canvas>

</Grid>

</Viewbox>

</Window>

**2.2 Simon say “构建程序的基础结构”**

UI界面设计好了之后，我们现在来看游戏的基础结构。需要在代码中添加响应SkeletonFrameReady事件的逻辑。在SkeletonFrameReady事件中，添加代码来跟踪游戏者手部关节的运动。基本代码如下：

private void KinectDevice\_SkeletonFrameReady(object sender, SkeletonFrameReadyEventArgs e)

{

using (SkeletonFrame frame = e.OpenSkeletonFrame())

{

if (frame != null)

{

frame.CopySkeletonDataTo(this.frameSkeletons);

Skeleton skeleton = GetPrimarySkeleton(this.frameSkeletons);

if (skeleton == null)

{

ChangePhase(GamePhase.GameOver);

}

else

{

LeftHandElement.Visibility = Visibility.Collapsed;

RightHandElement.Visibility = Visibility.Collapsed;

}

}

}

}

private static Skeleton GetPrimarySkeleton(Skeleton[] skeletons)

{

Skeleton skeleton = null;

if (skeletons != null)

{

//Find the closest skeleton

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

{

if (skeletons[i].TrackingState == SkeletonTrackingState.Tracked)

{

if (skeleton == null)

{

skeleton = skeletons[i];

}

else

{

if (skeleton.Position.Z > skeletons[i].Position.Z)

{

skeleton = skeletons[i];

}

}

}

}

}

return skeleton;

}

上面代码中TrackHand和GetJointPoint代码和Kinect连线游戏中相同。对于大多数游戏来说，使用“拉模型”来获取数据比使用事件模型获取数据性能要好。游戏通常是一个循环，可以手动的从骨骼数据流中获取下一帧骨骼数据。但是在我们的例子中，仍然使用的是事件模型，为的是能够减少代码量和复杂度。

**2.3 Simon say “添加游戏基本元素”**

Simon say游戏分成三步。起始步骤，我们之为GameOver，意味着当前没有可以玩的游戏。这是游戏的默认状态。这也是当Kinect探测不到游戏者时所切换到的状态。然后游戏开始循环，Simon给出一些指令，然后游戏者重复执行这些指令，重复这一过程，直到用户没能够正确的执行Simon给出的指令为止。应用程序定义了一个枚举变量来描述游戏所有可能的状态，以及定义了一个变量来跟踪游戏这当前所执行了的指令位置。另外我们需要一个变量来描述游戏者成功的次数或者游戏等级。当游戏者成功的执行了Simon给出的指令后，这个变量加1。下面的代码展示了这个枚举以及变量，变量的初始化在类的够着函数中执行。

public enum GamePhase

{

GameOver = 0,

SimonInstructing = 1,

PlayerPerforming = 2

}

public MainWindow()

{

InitializeComponent();

KinectSensor.KinectSensors.StatusChanged += KinectSensors\_StatusChanged;

this.KinectDevice = KinectSensor.KinectSensors.FirstOrDefault(x => x.Status == KinectStatus.Connected);

ChangePhase(GamePhase.GameOver);

this.currentLevel = 0;

}

SkeletonFrameReady事件需要根据当前游戏所处的状态来执行不同的操作。下面的代码中根据当前游戏的状态执行ChangePhase，ProcessGameOver和ProcessPlayerPerforming子方法。这些方法的详细执行过程将在后面介绍。ChangePhase方法接受一个GamePhase枚举值，后两个方法接受一个Skeleton类型的参数。

当应用程序探测不到骨骼数据时，游戏会终止，并切换到Game Over阶段。当游戏者离开Kinect视野时会发生这种情况。当游戏处在Simon给出操作步骤阶段时，隐藏界面上的手势图标。否则，更新这两个图标的位置。当游戏处在其它状态时，程序基于当前特定的游戏阶段调用特定的处理方法。

private void KinectDevice\_SkeletonFrameReady(object sender, SkeletonFrameReadyEventArgs e)

{

using (SkeletonFrame frame = e.OpenSkeletonFrame())

{

if (frame != null)

{

frame.CopySkeletonDataTo(this.frameSkeletons);

Skeleton skeleton = GetPrimarySkeleton(this.frameSkeletons);

if (skeleton == null)

{

ChangePhase(GamePhase.GameOver);

}

else

{

if (this.currentPhase == GamePhase.SimonInstructing)

{

LeftHandElement.Visibility = Visibility.Collapsed;

RightHandElement.Visibility = Visibility.Collapsed;

}

else

{

TrackHand(skeleton.Joints[JointType.HandLeft], LeftHandElement, LayoutRoot);

TrackHand(skeleton.Joints[JointType.HandRight], RightHandElement, LayoutRoot);

switch (this.currentPhase)

{

case GamePhase.GameOver:

ProcessGameOver(skeleton);

break;

case GamePhase.PlayerPerforming:

ProcessPlayerPerforming(skeleton);

break;

}

}

}

}

}

}

**2.4 开始新游戏**

当游戏处在GameOver阶段时，应用程序只调用了一个方法：该方法判断用户是否想玩游戏。当用户将相应的手放在UI界面上手势所处的位置时，游戏开始。左右手需要分别放在LeftHandStartElement和RightHandStartElement所处的位置内。在这个例子中，我们使用WPF自带的命中测试功能。我们的UI界面很小也很简单。InputHitTest操作所需要处理的UI元素很少，因此性能上没有太大问题。下面的代码展示了ProcessGameOver方法和GetHitTarget方法。

private void ProcessGameOver(Skeleton skeleton)

{

//判断用户是否想开始新的游戏

if (HitTest(skeleton.Joints[JointType.HandLeft], LeftHandStartElement) && HitTest(skeleton.Joints[JointType.HandRight], RightHandStartElement))

{

ChangePhase(GamePhase.SimonInstructing);

}

}

private bool HitTest(Joint joint, UIElement target)

{

return (GetHitTarget(joint, target) != null);

}

private IInputElement GetHitTarget(Joint joint, UIElement target)

{

Point targetPoint = LayoutRoot.TranslatePoint(GetJointPoint(this.KinectDevice, joint, LayoutRoot.RenderSize, new Point()), target);

return target.InputHitTest(targetPoint);

}

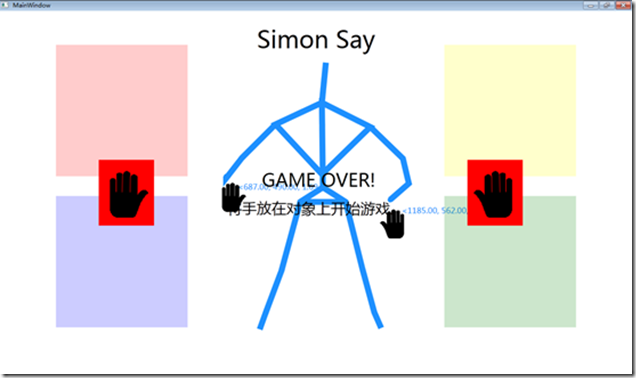
ProcessGameOver方法的逻辑简单明了：如果游戏者的任何一只手在UI界面上的对应位置，就切换当前游戏所处的状态。GetHitTarget方法用来测试给定的关节点是否在可视化控件有效范围内。他接受关节点数据和可视化控件，返回该点所在的特定的IInputElement对象。虽然代码只有两行，但是了解背后的逻辑很重要。

命中测试算法包含三个步骤，首先需要将关节点所在的骨骼空间坐标系中坐标转换到对应的LayoutRoot元素所在的空间坐标中来。GetJointPoint实现了这个功能。其次，使用UIElement类中的TranslatePoint方法将关节点从LayoutRoot元素所在的空间坐标转换到目标元素所在的空间坐标中。最后，点和目标元素在一个坐标空间之后，调用目标元素的InputHitTest方法，方法返回目标对象树中，点所在的确切的UI元素，任何非空值都表示命中测试成功。

注意到逻辑之所以这么简单是因为我们采用的UI布局方式，应用程序假定全屏运行并且不能调整大小。将UI界面设置为静态的，确定大小能够极大的简化计算量。另外，将所有的可交互的UI元素放在Canvas容器内使得我们只有一个坐标空间。使用其他容器空间来包含元素或者使用诸如HorizonAlignment，VerticalAlignment或者Margin这些自动布局属性会增加命中测试的复杂性。简言之，越是复杂的UI布局，命中测试的逻辑越复杂，也越会影响程序的性能。

**2.4.1 更改游戏状态**

编译并运行程序，如果没问题的话，结果应该如下图。应用程序能够追踪手部的运动，并且当用户将手放到对应的位置后，应用程序的状态会从GameOver转到SimonInstructing状态。下一步是要实现ChangePhase方法，代码如下：

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204131225337889.png)

private void ChangePhase(GamePhase newPhase)

{

if (newPhase != this.currentPhase)

{

this.currentPhase = newPhase;

switch (this.currentPhase)

{

case GamePhase.GameOver:

this.currentLevel = 0;

RedBlock.Opacity = 0.2;

BlueBlock.Opacity = 0.2;

GreenBlock.Opacity = 0.2;

YellowBlock.Opacity = 0.2;

GameStateElement.Text = "GAME OVER!";

ControlCanvas.Visibility = Visibility.Visible;

GameInstructionsElement.Text = "将手放在对象上开始新的游戏。";

break;

case GamePhase.SimonInstructing:

this.currentLevel++;

GameStateElement.Text = string.Format("Level {0}", this.currentLevel);

ControlCanvas.Visibility = Visibility.Collapsed;

GameInstructionsElement.Text = "注意观察Simon的指示。";

GenerateInstructions();

DisplayInstructions();

break;

case GamePhase.PlayerPerforming:

this.instructionPosition = 0;

GameInstructionsElement.Text = "请重复 Simon的指示";

break;

}

}

}

上面的代码和Kinect无关，事实上可以使用鼠标或者触控板来实现这一步，但是这段代码是必须的。ChangePhase方法用来控制UI界面来显示当前游戏状态的变化，维护一些游戏进行所需要的数据。在GameOver状态时，矩形框会渐变消失，然后改变操作指示，显示按钮来开始一个新的游戏。SimonInStructing状态不在更新UI界面讨论范围内，他调用了两个方法，用来产生指令集合 (GenerateInstructions)，并将这些指令显示到UI界面上(DisplayInstructions)，代码中也定义了instructionPosition变量，来维护当前所完成的指令步骤。

**2.4.2 显示Simon的指令**

下面的代码显示了一些局部变量和GenerateInstructions方法。instructionSequence变量用来存储一系列的UIElements对象，这些对象组成了Simon的指令集合。游戏者必须用手依次移动到这些指令上。这些指令的顺序是随机设定的。每一关指令的个数和当前等级是一样的。比如，到了第五关，就有5个指令。代码也显示了DisplayInstruction方法，他创建并触发了一个故事板动画效果来根据指令的顺序来改变每一个矩形的透明度。

private UIElement[] instructionSequence;

private int instructionPosition;

private int currentLevel;

private Random rnd = new Random();

private void GenerateInstructions()

{

this.instructionSequence = new UIElement[this.currentLevel];

for (int i = 0; i < this.currentLevel; i++)

{

switch (rnd.Next(1, 4))

{

case 1:

this.instructionSequence[i] = RedBlock;

break;

case 2:

this.instructionSequence[i] = BlueBlock;

break;

case 3:

this.instructionSequence[i] = GreenBlock;

break;

case 4:

this.instructionSequence[i] = YellowBlock;

break;

}

}

}

private void DisplayInstructions()

{

Storyboard instructionsSequence = new Storyboard();

DoubleAnimationUsingKeyFrames animation;

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

{

this.instructionSequence[i].ApplyAnimationClock(FrameworkElement.OpacityProperty, null);

animation = new DoubleAnimationUsingKeyFrames();

animation.FillBehavior = FillBehavior.Stop;

animation.BeginTime = TimeSpan.FromMilliseconds(i \* 1500);

Storyboard.SetTarget(animation, this.instructionSequence[i]);

Storyboard.SetTargetProperty(animation, new PropertyPath("Opacity"));

instructionsSequence.Children.Add(animation);

animation.KeyFrames.Add(new EasingDoubleKeyFrame(0.3, KeyTime.FromTimeSpan(TimeSpan.Zero)));

animation.KeyFrames.Add(new EasingDoubleKeyFrame(1, KeyTime.FromTimeSpan(TimeSpan.FromMilliseconds(500))));

animation.KeyFrames.Add(new EasingDoubleKeyFrame(1, KeyTime.FromTimeSpan(TimeSpan.FromMilliseconds(1000))));

animation.KeyFrames.Add(new EasingDoubleKeyFrame(0.3, KeyTime.FromTimeSpan(TimeSpan.FromMilliseconds(1300))));

}

instructionsSequence.Completed += (s, e) => { ChangePhase(GamePhase.PlayerPerforming); };

instructionsSequence.Begin(LayoutRoot);

}

运行程序，当双手放到指定位置是，Simon游戏开始。

**2.4.3 执行Simon的指令**

游戏的最后一步就是根据指令来捕捉用户的动作。注意到当故事版动画完成了显示Simon的指令后，程序调用ChangePhase方法使游戏进入PlayerPerforming阶段。当在PlayerPerforming阶段时，应用程序执行ProcessPlayerPerforming方法。表面上，实现该方法很简单。逻辑是游戏者重复Simon给出的操作步骤，将手放在对应矩形上方。这和之前做的命中测试逻辑是一样的。但是，和测试两个静态的UI对象不同，我们测试指令集合中的下一个指令对应的元素。下面的代码展示的ProcessPlayerPerforming方法，编译并运行就可以看到效果了，虽然能够运行，但是它对用户非常不友好。实际上，这个游戏不能玩。我们的用户界面不完整。

private void ProcessPlayerPerforming(Skeleton skeleton)

{

//Determine if user is hitting a target and if that target is in the correct sequence.

UIElement correctTarget = this.\_InstructionSequence[this.\_InstructionPosition];

IInputElement leftTarget = GetHitTarget(skeleton.Joints[JointType.HandLeft], GameCanvas);

IInputElement rightTarget = GetHitTarget(skeleton.Joints[JointType.HandRight], GameCanvas);

if(leftTarget != null && rightTarget != null)

{

ChangePhase(GamePhase.GameOver);

}

else if(leftTarget == null && rightTarget == null)

{

//Do nothing - target found

}

else if((leftTarget == correctTarget && rightTarget == null) ||

(rightTarget == correctTarget && leftTarget == null))

{

this.\_InstructionPosition++;

if(this.\_InstructionPosition >= this.\_InstructionSequence.Length)

{

ChangePhase(GamePhase.SimonInstructing);

}

}

else

{

ChangePhase(GamePhase.GameOver);

}

}

上面的代码中，第一行获取目标对象元素，即指令序列中的当前指令。然后执行命中测试，获取左右手对应的命中元素。下面的代码对这三个变量进行操作。如果两只手都在UI元素上，游戏结束。我们的游戏很简单，只能允许一次点击一个矩形。当两只手都不在UI元素上时，什么都不做。如果一只手命中了期望的对象，我们就把当前指令步骤加1。当指令集合中还有其他指令时游戏继续运行，直到完成了指令集合中的最后一个指令。当完成了最后一个指令后，游戏状态又变为了SimonInstruction状态，然后将游戏者带入下一轮游戏。直到游戏者不能重复Simon指令而进入GameOver状态。

如果游戏者动作够快，那么上面代码工作正常，因为只要用户手进入到了可视化元素有效区域，那么指令位置就会自增，游戏者在进入到下一个指令之前，没有时间来从UI元素所在的空间上移除手。这么快的速度不可能使得游戏者能够闯过第二关。当游戏者成功的闯过第二关的指令后，游戏就会突然停止。

解决这个问题的办法是在进入到下一个指令前等待，直到游戏者的手势从UI界面上清除。这使得游戏者有机会能够调整手势的位置开始进入下一条指令，我们需要记录用户的手什么时候进入和离开UI对象。

在WPF中，每一个UIElement对象都会在鼠标进入和离开其有效范围内时触发MouseEnter和MouseLeave事件。不幸的是，如前面所讨论的，WPF本身并不支持Kinect产生的关节点数据和UI的直接交互，如果当关节点进入或者离开可视化元素时能够触发诸如JointEnter和JointLeave事件，那么就简单多了。既然不支持，那么我们只有自己手动实现这个逻辑了。要实现一个可重用，优雅，并能像鼠标那样能够在底层追踪关节点运动这样的控件不太容易并且不容易做成通用的。我们只针对我们当前遇到的问题来实现这个功能。

要修正游戏中的这个问题比较容易。我们添加一系列成员变量来保存UI元素上的哪一个鼠标手势最后悬停在上面。当用户的手经过UI元素的上方时，更新这个变量。对于每一个新的骨骼数据帧。我们检查游戏者手的位置，如果它离开了UI元素空间，那么我们处理这个UI元素。下面的代码展示了对上面ProcessPlayerPerforming方法的改进。改进的部分用粗体表示。

**private IInputElement leftHandTarget;**

**private IInputElement rightHandTarget;**

private void ProcessPlayerPerforming(Skeleton skeleton)

{

//判断用户是否手势是否在目标对象上面，且在指定中的正确顺序

UIElement correctTarget = this.instructionSequence[this.instructionPosition];

IInputElement leftTarget = GetHitTarget(skeleton.Joints[JointType.HandLeft], GameCanvas);

IInputElement rightTarget = GetHitTarget(skeleton.Joints[JointType.HandRight], GameCanvas);

**bool hasTargetChange = (leftTarget != this.leftHandTarget) || (rightTarget != this.rightHandTarget);**

**if (hasTargetChange)**

{

if (leftTarget != null && rightTarget != null)

{

ChangePhase(GamePhase.GameOver);

}

**else if ((leftHandTarget == correctTarget && rightHandTarget == null) ||**

**(rightHandTarget == correctTarget && leftHandTarget == null))**

{

this.instructionPosition++;

if (this.instructionPosition >= this.instructionSequence.Length)

{

ChangePhase(GamePhase.SimonInstructing);

}

}

else if (leftTarget != null || rightTarget != null)

{

//Do nothing - target found

}

else

{

ChangePhase(GamePhase.GameOver);

}

**if (leftTarget != this.leftHandTarget)**

{

if (this.leftHandTarget != null)

{

((FrameworkElement)this.leftHandTarget).Opacity = 0.2;

}

if (leftTarget != null)

{

((FrameworkElement)leftTarget).Opacity = 1;

}

this.leftHandTarget = leftTarget;

}

**if (rightTarget != this.rightHandTarget)**

{

if (this.rightHandTarget != null)

{

((FrameworkElement)this.rightHandTarget).Opacity = 0.2;

}

if (rightTarget != null)

{

((FrameworkElement)rightTarget).Opacity = 1;

}

this.rightHandTarget = rightTarget;

}

}

}

现在运行代码，由于游戏需要两只手进行操作，所以没法截图，读者可以自己下载代码运行。

**2.5 需要改进的地方**

这个游戏演示了如何建立一个基于Kinect进行交互的程序，虽然程序可以运行，但是仍然有一些有待改进的地方，有以下三个方面可以进行改进：用户体验，游戏内容和表现形式。

**2.5.1 用户体验**

基于Kinect的应用程序和游戏比较新颖，在这种应用达到成熟前，要想获得良好的用户体验需要进行很多实验和测试。我们的Simon Say游戏的用户界面就有很多值得改进的地方。Simon Say的游戏者可能会意外的触摸到游戏的区间。游戏时在游戏开始的时候，有可能会碰到开始按钮。一旦两只手都在指定的区间，游戏就开始产生指令，如果用户没有及时的放开手，他可能会无意识的碰到一个游戏对象。一个有效的解决方法是在产生指令之前，给予用户一定的时间让其重新设置手的位置。因为人们会自然而然的将手垂在身体两边。一个比较好的变通方法是简单的给一个倒计时。在不同的关卡间，也可以给这样一个时间间隔。在开始新的一关时，用户应该有时间来从可视化元素中移开手。

**2.5.2 游戏内容**

产生游戏指令序列的逻辑比较简单。指令序列中指令的数目和当前的关卡是一致的。每一条指令所选择的可视化元素是随机选择的。在原始的Simon Say游戏中，新一轮的游戏通常会添加一些新的指令。例如，第一关中有红的，第二关中有红的和蓝的，第三关增加了绿的。因此在第三关指令可以是，红绿蓝。另一种改进可以不在每一关增加一个指令。而是将指令的个数设置为当前关卡数的2倍。软件开发一个有趣的地方就是应用程序可以有多种产生指令序列的算法。例如，应用程序可以分为容易，中等，难三种产生指令序列的方法供用户选择。最基本的产生指令序列的逻辑就是每一关要尽可能的比前一关要长，并且指令显示速度要以一个常量的速度显示。要增加游戏的难度，在显示指令序列时可以减少指令展示给用户的时间。

**2.5.3表现形式**

创建一个赋予表现力的程序远不止我们这里所介绍的这些内容。可能做一点改动就可以将我们的UI做的更加好看，比如，可以在显示指令提示，以及用户移入和离开指定区域时可以采用一些比较好看的动画。当用户执行的指令正确时，可以展现一个动画效果给予奖励。同样的，在游戏结束时也可以展现出一个动画。

**3. 结语**

本文围绕Kinect介绍了WPF输入系统的相关知识，并讨论了如何将Kinect作为WPF程序的输入设备与应用程序进行交互，最后展示了一个Simon say的小游戏来讲述如何进行这些实际操作。

限于篇幅，下面一篇文章将会对骨骼追踪进行进一步阐述，并对Simon say这个小游戏增加姿势识别，敬请期待。

本文所有代码点击[此处](http://files.cnblogs.com/yangecnu/KinectSDK_AdvanceSkeletonTracking_part1_KinectSimonSayGame.rar)下载，希望以上文章对你了解Kinect SDK有所帮助，谢谢！