[**Kinect for Windows SDK开发入门(十一)：手势识别 下：基本手势识别**](http://www.cnblogs.com/yangecnu/archive/2012/04/28/KinectSDK_GesturesDetection_part2_BasicGesturesDetection.html)

[上文](http://www.cnblogs.com/yangecnu/archive/2012/04/21/KinectSDK_GesturesDetection_part1_BasicConception.html)简要介绍了手势识别的基本概念和手势识别的基本方法，并以八种手势中的挥手(wave)为例讲解了如何使用算法对手势进行识别，本文接[上文](http://www.cnblogs.com/yangecnu/archive/2012/04/21/KinectSDK_GesturesDetection_part1_BasicConception.html)，继续介绍如何建立一个手部追踪类库，并以此为基础，对剩余7中常用的手势进行识别做一些介绍。

**1. 基本的手势追踪**

手部追踪在技术上和手势识别不同，但是它和手势识别中用到的一些基本方法是一样的。在开发一个具体的手势控件之前，我们先建立一个可重用的追踪手部运动的类库以方便我们后续开发。这个手部追踪类库包含一个以动态光标显示的可视化反馈机制。手部追踪和手势控件之间的交互高度松耦合。

首先在Visual Studio中创建一个WPF控件类库项目。然后添加四个类： KinectCursorEventArgs.cs，KinectInput.cs，CusrorAdorner.cs和KinectCursorManager.cs这四个类之间通过相互调用来基于用户手所在的位置来完成光标位置的管理。KinectInput类包含了一些事件，这些事件可以在KinectCursorManager和一些控件之间共享。KinectCursorEventArgs提供了一个属性集合，能够用来在事件触发者和监听者之间传递数据。KinectCursorManager用来管理从Kinect传感器中获取的骨骼数据流，然后将其转换到WPF坐标系统，提供关于转换到屏幕位置的可视化反馈，并寻找屏幕上的控件，将事件传递到这些控件上。最后CursorAdorner.cs类包含了代表手的图标的可视化元素。

KinectCursorEventArgs继承自RoutedEventArgs类，它包含四个属性：X、Y、Z和Cursor。X、Y、Z是一个小数，代表待转换的用户手所在位置的宽度，高度和深度值。Cursor用来存储CursorAdorner类的实例，后面将会讨论，下面的代码展示了KinectCursorEventArgs类的基本结构，其中包含了一些重载的构造器。

public class KinectCursorEventArgs:RoutedEventArgs

{

public double X { get; set; }

public double Y { get; set; }

public double Z { get; set; }

public CursorAdorner Cursor { get; set; }

public KinectCursorEventArgs(double x, double y)

{

X = x;

Y = y;

}

public KinectCursorEventArgs(Point point)

{

X = point.X;

Y = point.Y;

}

}

RoutedEventArgs基类有一个构造函数能够接收RoutedEvent作为参数。这是一个有点特别的签名，WPF中的UIElement使用这种特殊的语法触发事件。下面的代码是KinectCursorEventArgs类对这一签名的实现，以及其他一些重载方法。

public KinectCursorEventArgs(RoutedEventroutedEvent) : base(routedEvent) { }   
  
publicKinectCursorEventArgs(RoutedEventroutedEvent, doublex, doubley, doublez)   
 : base(routedEvent) { X = x; Y = y; Z = z; }   
  
publicKinectCursorEventArgs(RoutedEventroutedEvent, Pointpoint)   
 : base(routedEvent) { X = point.X; Y = point.Y; }   
  
publicKinectCursorEventArgs(RoutedEventroutedEvent, Pointpoint,doublez)   
 : base(routedEvent) { X = point.X; Y = point.Y; Z = z; }   
  
publicKinectCursorEventArgs(RoutedEventroutedEvent, objectsource)   
 : base(routedEvent, source) {}   
  
publicKinectCursorEventArgs(RoutedEventroutedEvent,objectsource,doublex,doubley,doublez)   
 : base(routedEvent, source) { X = x; Y = y; Z = z; }   
  
publicKinectCursorEventArgs(RoutedEventroutedEvent, objectsource, Pointpoint)   
 : base(routedEvent, source) { X = point.X; Y = point.Y; } publicKinectCursorEventArgs(RoutedEventroutedEvent, objectsource, Pointpoint,doublez)   
 : base(routedEvent, source) { X = point.X; Y = point.Y; Z = z; }

接下来，要在KinectInput类中创建事件来将消息从KinectCursorManager中传递到可视化控件中去。这些事件传递的数据类型为KinectCursorEventArgs类型。

在KinectInput类中添加一个KinectCursorEventHandler的代理类型：(1) 添加一个静态的routed event声明。(2) 添加KinectCursorEnter，KinectCursorLeave，KinectCursorMove，KinectCursorActive和KinectCursorDeactivated事件的add和remove方法。下面的代码展示了三个和cursor相关的事件，其他的如KinectCursorActivated和KinectCursorDeactivated事件和这个结构相同：

public delegate void KinectCursorEventHandler(object sender,KinectCursorEventArgs e);

public static class KinectInput

{

public static readonly RoutedEvent KinectCursorEnterEvent=EventManager.RegisterRoutedEvent("KinectCursorEnter",RoutingStrategy.Bubble,

typeof(KinectCursorEventHandler),typeof(KinectInput));

public static void AddKinectCursorEnterHandler(DependencyObject o, KinectCursorEventHandler handler)

{

((UIElement)o).AddHandler(KinectCursorEnterEvent, handler);

}

public static void RemoveKinectCursorEnterHandler(DependencyObject o, KinectCursorEventHandler handler)

{

((UIElement)o).RemoveHandler(KinectCursorEnterEvent, handler);

}

public static readonly RoutedEvent KinectCursorLeaveEvent=EventManager.RegisterRoutedEvent("KinectCursorLeave",RoutingStrategy.Bubble,

typeof(KinectCursorEventHandler),typeof(KinectInput));

public static void AddKinectCursorLeaveHandler(DependencyObject o, KinectCursorEventHandler handler)

{

((UIElement)o).AddHandler(KinectCursorEnterEvent,handler);

}

public static void RemoveKinectCursorLeaveHandler(DependencyObject o, KinectCursorEventHandler handler)

{

((UIElement)o).RemoveHandler(KinectCursorEnterEvent, handler);

}

}

注意到以上代码中没有声明任何GUI编程中的Click事件。这是因为在设计控件类库时，Kinect中并没有点击事件，相反Kinect中两个重要的行为是enter和leave。手势图标可能会移入和移出某一个可视化控件的有效区域。如果要实现普通GUI控件的点击效果的话，必须在Kinect中对这一事件进行模拟，因为Kinect原生并不支持点击这一行为。

CursorAdorner类用来保存用户手势图标可视化元素，它继承自WPF的Adorner类型。之所以使用这个类型是因为它有一个特点就是总是在其他元素之上绘制，这在我们的项目中非常有用，因为我们不希望我们的光标会被其他元素遮挡住。代码如下所示，我们默认的adorner对象将绘制一个默认的可视化元素来代表光标，当然也可以传递一个自定义的可视化元素。

public class CursorAdorner:Adorner

{

private readonly UIElement \_adorningElement;

private VisualCollection \_visualChildren;

private Canvas \_cursorCanvas;

protected FrameworkElement \_cursor;

StroyBoard \_gradientStopAnimationStoryboard;

readonly static Color \_backColor = Colors.White;

readonly static Color \_foreColor = Colors.Gray;

public CursorAdorner(FrameworkElement adorningElement)

: base(adorningElement)

{

this.\_adorningElement = adorningElement;

CreateCursorAdorner();

this.IsHitTestVisible = false;

}

public CursorAdorner(FrameworkElement adorningElement, FrameworkElement innerCursor)

: base(adorningElement)

{

this.\_adorningElement = adorningElement;

CreateCursorAdorner(innerCursor);

this.IsHitTestVisible = false;

}

public FrameworkElement CursorVisual

{

get { return \_cursor; }

}

public void CreateCursorAdorner()

{

var innerCursor = CreateCursor();

CreateCursorAdorner(innerCursor);

}

protected FrameworkElement CreateCursor()

{

var brush = new LinearGradientBrush();

brush.EndPoint = new Point(0, 1);

brush.StartPoint = new Point(0, 0);

brush.GradientStops.Add(new GradientStop(\_backColor, 1));

brush.GradientStops.Add(new GradientStop(\_foreColor, 1));

var cursor = new Ellipse()

{

Width=50,

Height=50,

Fill=brush

};

return cursor;

}

public void CreateCursorAdorner(FrameworkElement innerCursor)

{

\_visualChildren = new VisualCollection(this);

\_cursorCanvas = new Canvas();

\_cursor = innerCursor;

\_cursorCanvas.Children.Add(this.\_cursorCanvas);

\_visualChildren.Add(this.\_cursorCanvas);

AdornerLayer layer = AdornerLayer.GetAdornerLayer(\_adorningElement);

layer.Add(this);

}

}

因为继承自Adorner基类，我们需要重写某些基类的方法，下面的代码展示了基类中的方法如何和CreateCursorAdorner方法中实例化的\_visualChildren和\_cursorCanvas字段进行绑定。

protected override int VisualChildrenCount

{

get

{

return \_visualChildren.Count;

}

}

protected override Visual GetVisualChild(int index)

{

return \_visualChildren[index];

}

protected override Size MeasureOverride(Size constraint)

{

this.\_cursorCanvas.Measure(constraint);

return this.\_cursorCanvas.DesiredSize;

}

protected override Size ArrangeOverride(Size finalSize)

{

this.\_cursorCanvas.Arrange(new Rect(finalSize));

return finalSize;

}

CursorAdorner对象也负责找到手所在的正确的位置，该对象的UpdateCursor方法如下，方法接受X，Y坐标位置作为参数。然后方法在X，Y上加一个偏移量以使得图像的中心在X，Y之上，而不是在图像的边上。另外，我们提供了该方法的一个重载，该重载告诉光标对象一个特殊的坐标会传进去，所有的普通方法调用UpdateCursor将会被忽略。当我们在磁性按钮中想忽略基本的手部追踪给用户更好的手势体验时很有用。

public void UpdateCursor(Pointposition, boolisOverride)   
{   
 \_isOverriden = isOverride;   
 \_cursor.SetValue(Canvas.LeftProperty,position.X-(\_cursor.ActualWidth/2));   
 \_cursor.SetValue(Canvas.LeftProperty, position.Y - (\_cursor.ActualHeight / 2));   
}   
  
public void UpdateCursor(Pointposition)   
{   
 if(\_isOverriden) return;   
 \_cursor.SetValue(Canvas.LeftProperty, position.X - (\_cursor.ActualWidth / 2));   
 \_cursor.SetValue(Canvas.LeftProperty, position.Y - (\_cursor.ActualHeight / 2));   
}

最后，添加光标对象动画效果。当Kinect控件需要悬浮于一个元素之上，在用户等待的时候，给用户反馈一些信息告知正在发生的事情，这一点很有好处。下面了的代码展示了如何使用代码实现动画效果:

public virtual void AnimateCursor(doublemilliSeconds) {   
 CreateGradientStopAnimation(milliSeconds);   
 if(\_gradientStopAnimationStoryboard != null)   
 \_gradientStopAnimationStoryboard.Begin(this, true);   
}   
  
public virtual void StopCursorAnimation(doublemilliSeconds)   
{   
 if(\_gradientStopAnimationStoryboard != null)   
 \_gradientStopAnimationStoryboard.Stop(this);   
}   
  
public virtual void CreateGradientStopAnimation(doublemilliSeconds) {   
  
 NameScope.SetNameScope(this, newNameScope());   
  
 varcursor = \_cursor asShape;   
 if(cursor == null)   
 return;   
 varbrush = cursor.Fill asLinearGradientBrush;   
 varstop1 = brush.GradientStops[0];   
 varstop2 = brush.GradientStops[1];   
 this.RegisterName("GradientStop1", stop1);   
 this.RegisterName("GradientStop2", stop2);   
  
 DoubleAnimationoffsetAnimation = newDoubleAnimation();   
 offsetAnimation.From = 1.0;   
 offsetAnimation.To = 0.0;   
 offsetAnimation.Duration = TimeSpan.FromMilliseconds(milliSeconds);   
  
  
 Storyboard.SetTargetName(offsetAnimation, "GradientStop1");   
 Storyboard.SetTargetProperty(offsetAnimation,   
 newPropertyPath(GradientStop.OffsetProperty));   
  
  
 DoubleAnimationoffsetAnimation2 = newDoubleAnimation();   
 offsetAnimation2.From = 1.0;   
 offsetAnimation2.To = 0.0;   
  
 offsetAnimation2.Duration = TimeSpan.FromMilliseconds(milliSeconds);   
  
 Storyboard.SetTargetName(offsetAnimation2, "GradientStop2");   
 Storyboard.SetTargetProperty(offsetAnimation2,   
 newPropertyPath(GradientStop.OffsetProperty));   
  
 \_gradientStopAnimationStoryboard = newStoryboard();   
 \_gradientStopAnimationStoryboard.Children.Add(offsetAnimation);   
 \_gradientStopAnimationStoryboard.Children.Add(offsetAnimation2);   
 \_gradientStopAnimationStoryboard.Completed += delegate{ \_gradientStopAnimationStoryboard.Stop(this); };   
}

为了实现KinectCursorManager类,我们需要几个帮助方法,代码如下，GetElementAtScreenPoint方法告诉我们哪个WPF对象位于X,Y坐标下面，在这个高度松散的结构中，GetElementAtScreenPoint方法是主要的引擎，用来从KinectCurosrManager传递消息到自定义控件，并接受这些事件。另外，我们使用两个方法来确定我们想要追踪的骨骼数据以及我们想要追踪的手。

private static UIElement GetElementAtScreenPoint(Point point, Window window)

{

if (!window.IsVisible)

return null;

Point windowPoint = window.PointFromScreen(point);

IInputElement element = window.InputHitTest(windowPoint);

if (element is UIElement)

return (UIElement)element;

else

return null;

}

private static Skeleton GetPrimarySkeleton(IEnumerable<Skeleton> skeletons)

{

Skeleton primarySkeleton = null;

foreach (Skeleton skeleton in skeletons)

{

if (skeleton.TrackingState != SkeletonTrackingState.Tracked)

{

continue;

}

if (primarySkeleton == null)

primarySkeleton = skeleton;

else if (primarySkeleton.Position.Z > skeleton.Position.Z)

primarySkeleton = skeleton;

}

return primarySkeleton;

}

private static Joint? GetPrimaryHand(Skeleton skeleton)

{

Joint leftHand=skeleton.Joints[JointType.HandLeft];

Joint rightHand=skeleton.Joints[JointType.HandRight];

if (rightHand.TrackingState == JointTrackingState.Tracked)

{

if (leftHand.TrackingState != JointTrackingState.Tracked)

return rightHand;

else if (leftHand.Position.Z > rightHand.Position.Z)

return rightHand;

else

return leftHand;

}

if (leftHand.TrackingState == JointTrackingState.Tracked)

{

return leftHand;

}

else

return null;

}

KinectCursorManager应该是一个单例类。这样设计是能够使得代码实例化起来简单。任何和KinectCursorManager工作的控件在KinectCursorManager没有实例化的情况下可以独立的进行KinectCursorManager的实例化。这意味着任何开发者使用这些控件不需要了解KinectCursorManager对象本身。相反，开发者能够简单的将控件拖动到应用程序中，控件负责实例化KinectCursorManager对象。为了使得这种自服务功能能和KinectCursorMange类一起使用，我们需要创建一个重载的Create方法来将应用程序的主窗体类传进来。下面的代码展示了重载的构造函数以及特殊的单例模式的实现方法。

public class KinectCursorManager

{

private KinectSensor kinectSensor;

private CursorAdorner cursorAdorner;

private readonly Window window;

private UIElement lastElementOver;

private bool isSkeletonTrackingActivated;

private static bool isInitialized;

private static KinectCursorManager instance;

public static void Create(Window window)

{

if (!isInitialized)

{

instance = new KinectCursorManager(window);

isInitialized = true;

}

}

public static void Create(Window window,FrameworkElement cursor)

{

if (!isInitialized)

{

instance = new KinectCursorManager(window,cursor);

isInitialized = true;

}

}

public static void Create(Window window, KinectSensor sensor)

{

if (!isInitialized)

{

instance = new KinectCursorManager(window, sensor);

isInitialized = true;

}

}

public static void Create(Window window, KinectSensor sensor, FrameworkElement cursor)

{

if (!isInitialized)

{

instance = new KinectCursorManager(window, sensor, cursor);

isInitialized = true;

}

}

public static KinectCursorManager Instance

{

get { return instance; }

}

private KinectCursorManager(Window window) : this(window, KinectSensor.KinectSensors[0]) { }

private KinectCursorManager(Window window, FrameworkElement cursor) : this(window, KinectSensor.KinectSensors[0], cursor) { }

private KinectCursorManager(Window window, KinectSensor sensor) : this(window, sensor, null) { }

private KinectCursorManager(Window window, KinectSensor sensor, FrameworkElement cursor)

{

this.window = window;

if (KinectSensor.KinectSensors.Count > 0)

{

window.Unloaded += delegate

{

if (this.kinectSensor.SkeletonStream.IsEnabled)

this.kinectSensor.SkeletonStream.Disable();

};

window.Loaded += delegate

{

if (cursor == null)

cursorAdorner = new CursorAdorner((FrameworkElement)window.Content);

else

cursorAdorner = new CursorAdorner((FrameworkElement)window.Content, cursor);

this.kinectSensor = sensor;

this.kinectSensor.SkeletonFrameReady += SkeletonFrameReady;

this.kinectSensor.SkeletonStream.Enable(new TransformSmoothParameters());

this.kinectSensor.Start();

};

}

}

……

下面的代码展示了KinectCursorManager如何和窗体上的可视化元素进行交互。当用户的手位于应用程序可视化元素之上时，KinectCursorManager对象始终保持对当前手所在的可视化元素以及之前手所在的可视化元素的追踪。当这一点发生改变时，KinectCursorManager会触发之前控件的leave事件和当前控件的enter事件。我们也保持对KinectSensor对象的追踪，并触发activated和deactivated事件。

private void SetSkeletonTrackingActivated()

{

if (lastElementOver != null && isSkeletonTrackingActivated == false)

{

lastElementOver.RaiseEvent(new RoutedEventArgs(KinectInput.KinectCursorActivatedEvent));

}

isSkeletonTrackingActivated = true;

}

private void SetSkeletonTrackingDeactivated()

{

if (lastElementOver != null && isSkeletonTrackingActivated == false)

{

lastElementOver.RaiseEvent(new RoutedEventArgs(KinectInput.KinectCursorDeactivatedEvent));

}

isSkeletonTrackingActivated = false ;

}

private void HandleCursorEvents(Point point, double z)

{

UIElement element = GetElementAtScreenPoint(point, window);

if (element != null)

{

element.RaiseEvent(new KinectCursorEventArgs(KinectInput.KinectCursorMoveEvent, point, z) {Cursor=cursorAdorner });

if (element != lastElementOver)

{

if (lastElementOver != null)

{

lastElementOver.RaiseEvent(new KinectCursorEventArgs(KinectInput.KinectCursorLeaveEvent, point, z) { Cursor = cursorAdorner });

}

element.RaiseEvent(new KinectCursorEventArgs(KinectInput.KinectCursorEnterEvent, point, z) { Cursor = cursorAdorner });

}

}

lastElementOver = element;

}

最后需要两个核心的方法来管理KinectCursorManger类。SkeletonFrameReady方法与之前一样，用来从Kinect获取骨骼数据帧时触发的事件。在这个项目中，SkeletonFrameReady方法负责获取合适的骨骼数据，然后获取合适的手部关节点数据。然后将手部关节点数据传到UpdateCusror方法中，UpdateCursor方法执行一系列方法将Kinect骨骼空间坐标系转化到WPF的坐标系统中，Kinect SDK中MapSkeletonPointToDepth方法提供了这一功能。SkeletonToDepthImage方法返回的X,Y值，然后转换到应用程序中实际的宽和高。和X,Y不一样，Z值进行了不同的缩放操作。简单的从Kinect深度摄像机中获取的毫米数据。代码如下，一旦这些坐标系定义好了之后，将他们传递到HandleCursorEvents方法然后CursorAdorner对象将会给用户以反馈。相关代码如下：

private void SkeletonFrameReady(objectsender, SkeletonFrameReadyEventArgse)   
{   
 using(SkeletonFrameframe = e.OpenSkeletonFrame())   
 {   
 if(frame == null|| frame.SkeletonArrayLength == 0) return;   
  
 Skeleton[] skeletons = newSkeleton[frame.SkeletonArrayLength];   
 frame.CopySkeletonDataTo(skeletons);   
 Skeletonskeleton = GetPrimarySkeleton(skeletons);   
  
 if(skeleton == null)   
 {   
 SetHandTrackingDeactivated();   
 }   
 else   
 {   
 Joint? primaryHand = GetPrimaryHand(skeleton);   
 if(primaryHand.HasValue)   
 {   
 UpdateCursor(primaryHand.Value);   
 }   
 else   
 {   
 SetHandTrackingDeactivated();   
 }   
 }   
 }   
}   
  
private voidSetHandTrackingDeactivated()   
{   
 cursorAdorner.SetVisibility(false);   
 if(lastElementOver != null&& isHandTrackingActivated == true)   
 {lastElementOver.RaiseEvent(newRoutedEventArgs(KinectInput.KinectCursorDeactivatedEvent)); };   
 isHandTrackingActivated = false;   
}   
  
private voidUpdateCursor(Jointhand)   
{   
 varpoint = kinectSensor.MapSkeletonPointToDepth(hand.Position, kinectSensor.DepthStream.Format);   
 floatx = point.X;   
 floaty = point.Y;   
 floatz = point.Depth;   
 x = (float)(x \* window.ActualWidth / kinectSensor.DepthStream.FrameWidth);   
 y = (float)(y \* window.ActualHeight / kinectSensor.DepthStream.FrameHeight);   
  
 PointcursorPoint = newPoint(x, y);   
 HandleCursorEvents(cursorPoint, z);   
 cursorAdorner.UpdateCursor(cursorPoint);   
}

至此，我们已经简单实现了一些基础结构，这些仅仅是实现了将用户手部的运动显示在屏幕上。现在我们要创建一个基类来监听光标对象的事件，首先创建一个KinectButton对象，该对象继承自WPF Button类型。定义三个之前在KinectInput中定义好的事件，同时创建这些事件的添加删除方法，代码如下：

public class KinectButton:Button   
{   
 public static readonlyRoutedEventKinectCursorEnterEvent = KinectInput.KinectCursorEnterEvent.AddOwner(typeof(KinectButton));   
 public static readonlyRoutedEventKinectCursorLeaveEvent = KinectInput.KinectCursorLeaveEvent.AddOwner(typeof(KinectButton));   
 public static readonlyRoutedEventKinectCursorMoveEvent = KinectInput.KinectCursorMoveEvent.AddOwner(typeof(KinectButton));   
 public static readonlyRoutedEventKinectCursorActivatedEvent = KinectInput.KinectCursorActivatedEvent.AddOwner(typeof(KinectButton));   
 public static readonlyRoutedEventKinectCursorDeactivatedEvent = KinectInput.KinectCursorDeactivatedEvent.AddOwner(typeof(KinectButton));   
  
  
 public eventKinectCursorEventHandlerKinectCursorEnter   
 {   
 add{ base.AddHandler(KinectCursorEnterEvent, value); }   
 remove{ base.RemoveHandler(KinectCursorEnterEvent, value); }   
 }   
  
 public eventKinectCursorEventHandlerKinectCursorLeave   
 {   
 add{ base.AddHandler(KinectCursorLeaveEvent, value); }   
 remove{ base.RemoveHandler(KinectCursorLeaveEvent, value); }   
 }   
  
 public eventKinectCursorEventHandlerKinectCursorMove   
 {   
 add{ base.AddHandler(KinectCursorMoveEvent, value); }   
 remove{ base.RemoveHandler(KinectCursorMoveEvent, value); }   
 }   
  
 public eventRoutedEventHandlerKinectCursorActivated   
 {   
 add{ base.AddHandler(KinectCursorActivatedEvent, value); }   
 remove{ base.RemoveHandler(KinectCursorActivatedEvent, value); }   
 }   
  
 public eventRoutedEventHandlerKinectCursorDeactivated   
 {   
 add{ base.AddHandler(KinectCursorDeactivatedEvent, value); }   
 remove{ base.RemoveHandler(KinectCursorDeactivatedEvent, value); }   
 }   
}

在KinectButton的构造函数中，首先检查当前控件是否运行在IDE或者一个实际的应用程序中。如果没有在设计器中，如果KinectCursorManager对象不存在，我们实例化KinectCursorManager对象。通过这种方式，我们可以在同一个窗体上添加多个Kinect 按钮。这些按钮自动创建KinectCursorManager的实例而不用开发者去创建。下面的代码展示了如何实现这一功能。KinectCursorManager类中的HandleCursorEvents方法负责处理这些事件。

public KinectButton()   
{   
 if(!System.ComponentModel.DesignerProperties.GetIsInDesignMode(this))   
 KinectCursorManager.Create(Application.Current.MainWindow);   
 this.KinectCursorEnter+=newKinectCursorEventHandler(OnKinectCursorEnter); this.KinectCursorLeave+=newKinectCursorEventHandler(OnKinectCursorLeave);   
 this.KinectCursorMove+=newKinectCursorEventHandler(OnKinectCursorMove);   
}   
  
protected virtual voidOnKinectCursorLeave(Objectsender, KinectCursorEventArgse)   
{ }   
  
protected virtual voidOnKinectCursorMove(Objectsender, KinectCursorEventArgse)   
{ }

下面的代码中，KinectCursorEnter事件中触发ClickEvent，将其改造成了一个标准的点击事件。使得KinectButton能够在鼠标移入时触发Click事件。Kinect中应用程序的交互术语还是使用之前GUI交互界面中的术语，这使得读者能够更容易理解。更重要的是，也能够使得开发者更容易理解，因为我们之前有很多使用按钮来构造用户界面的经验。当然终极的目标是舍弃这些各种各样的控件，改而使用纯粹的手势交互界面，但是按钮在现阶段的交互界面中还是很重要的。另外，这样也能够使用按钮来布局图形用户界面，只需要将普通的按钮换成Kinect按钮就可以了。

protected virtual void OnKinectCursorEnter(object sender, KinectCursorEventArgs e)

{

RaiseEvent(new RoutedEventArgs(ClickEvent));

}

这种控件有一个最大的问题，在大多数基于Kinect的应用程序中你看不到这个问题，那就是，你不能区分开是有意的还是无意的点击。在传统的基于鼠标的GUI应用中也有类似的倾向，每一次将鼠标移动到按钮上不用点击就会激活按钮。这种用户界面很容易不能使用，这也提醒了一个潜在的值得注意的问题，那就是将按钮从图形用户界面中移植到其他界面中可能存在的问题。悬浮按钮是微软试图解决这一特殊问题的一个尝试。

**2. 剩余七种常见手势的识别**

前面的文章中已经讲述了挥手手势的识别，本文接下来讲解余下7中常见手势的识别。

**2.1悬浮按钮(Hover Button)**

悬浮按钮是微软在2010年为Kinect对Xbox的操纵盘进行改进而引入的。 悬浮按钮通过将鼠标点击换成悬浮然后等待(hover-and-wait)动作，解决了不小心点击的问题。当光标位于按钮之上时，意味着用户通过将光标悬浮在按钮上一段时间来表示想选中按钮。另一个重要特点是悬浮按钮在用户悬浮并等待时，多少提供了视觉反馈。

在Kinect中实现悬浮按钮和在Windows Phone开发中实现轻点然后维持(tap-and-hold)这一手势在技术上比较类似。必须使用一个计时器来记录当前用户光标停留在按钮上的时间。一旦用户的手的光标和按钮的边界交叉就开始计时。如果某一个时间阈值内用户光标还没有移除，那么就触发点击事件。

创建一个名为HoverButton的类，他继承自之前创建的KinectButton类，在类中添加一个名为hoverTimer的DispatcherTime实例，代码如下。另外创建一个布尔型的timerEnable字段，将其设置为true。虽然目前不会用到这个字段，但是在后面部分将会用到，当我们想使用HoverButton的某些功能，但是不需要DispatcherTimer时就会非常有用。最后创建一个HoverInterval的依赖属性，使得运行我们将悬浮时间用代码或者xaml进行定义。默认设置为2秒，这是在大多是Xbox游戏中的时间。

public class HoverButton:KinectButton   
{   
 readonlyDispatcherTimerhoverTimer = newDispatcherTimer();   
 protected booltimerEnabled = true;   
  
 public doubleHoverInterval   
 {   
 get{ return(double)GetValue(HoverIntervalProperty); }   
 set   
 {   
 SetValue(HoverIntervalProperty, value);   
 }   
 }   
  
 public static readonlyDependencyPropertyHoverIntervalProperty =   
 DependencyProperty.Register("HoverInterval", typeof(double), typeof(HoverButton), newUIPropertyMetadata(2000d));   
……

}

要实现悬浮按钮的核心功能，我们必须覆写基类中的OnKinectCursorLeave和OnKinectCursorEnter方法，所有和KinectCursorManger进行交互的部分在KinectButton中已经实现了，因此我们在这里不用操心。在类的构造方法中，只需要实例化DispathcerTimer对象，HoverInterval依赖属性和注册hoverTimer\_Tick方法到计时器的Tick事件上即可。计时器在一定的间隔时间会触发Tick事件，该事件简单的处理一个Click事件，在OnKinectCursorEnter方法中启动计数器，在OnKinectCursorLeave事件中停止计数器。另外，重要的是，在enter和leave方法中启动和停止鼠标光标动画效果。

public HoverButton()   
{   
 hoverTimer.Interval = TimeSpan.FromMilliseconds(HoverInterval);   
 hoverTimer.Tick += newEventHandler(hoverTimer\_Tick);   
 hoverTimer.Stop();   
}   
  
voidhoverTimer\_Tick(objectsender, EventArgse)   
{   
 hoverTimer.Stop();   
 RaiseEvent(newRoutedEventArgs(ClickEvent));   
}   
  
protected override voidOnKinectCursorLeave(objectsender, KinectCursorEventArgse)   
{   
 if(timerEnabled)   
 {   
 e.Cursor.StopCursorAnimation();   
 hoverTimer.Stop();   
 }   
}   
  
protected override voidOnKinectCursorEnter(objectsender, KinectCursorEventArgse)   
{   
 if(timerEnabled)   
 {   
 hoverTimer.Interval = TimeSpan.FromMilliseconds(HoverInterval);   
 e.Cursor.AnimateCursor(HoverInterval);   
 hoverTimer.Start();   
 }   
}

悬浮按钮在基于Kinect的Xbox游戏中几乎无处不在。悬浮按钮唯一存在的问题是，光标手势悬停在按钮上时会抖动，这可能是Kinect中骨骼识别本身的问题。当在运动状态时，Kinect能够很好的对这些抖动进行平滑，因为即使在快速移动状态下，Kinect中的软件使用了一系列预测和平滑技术来对抖动进行处理。姿势，和上面的悬停一样，因为是静止的，所以可能存在抖动的问题。另外，用户一般不会保持手势静止，即使他们想哪样做。Kinect将这些小的运动返回给用户。当用户什么都没做时，抖动的手可能会破坏手势的动画效果。对悬浮按钮的一个改进就是磁性按钮(Magnet Button)，随着体感游戏的升级，这种按钮逐渐取代了之前的悬浮按钮,后面我们将看到如何实现磁性按钮。

**2.2 下压按钮(Push Button)**

就像悬浮按钮在Xbox中那样普遍一样，一些Kinect开发者也想创建一些类似PC上的那种交互方式的按钮，这种按钮称之为下压按钮(push button)。下压按钮试图将传统的GUI界面上的按钮移植到Kinect上去。为了代替鼠标点击，下压按钮使用一种将手向前推的手势来表示按下这一动作。

这种手势，手掌张开向前，在形式上有点像动态鼠标。下压按钮的核心算法就是探测手势在Z轴上有一个向负方向的运动。另外，相符方向必须有一个距离阈值，使得超过这一阈值就认为用户想要执行下压指令。代码如下所示：下压按钮有一个称之为Threshold的依赖属性，单位为毫米，这个值可以由开发者来根据动作的灵敏度来进行设置。当用户的手移动到下压按钮的上方时，我们记录一下当前位置手的Z值，以此为基准，然后比较手的深度值和阈值，如果超过阈值，就触发点击事件。

public class PushButton:KinectButton

{

protected double handDepth;

public double PushThreshold

{

get { return (double)GetValue(PushThresholdProperty); }

set { SetValue(PushThresholdProperty, value); }

}

public static readonly DependencyProperty PushThresholdProperty =

DependencyProperty.Register("PushThreshold", typeof(double), typeof(PushButton), new UIPropertyMetadata(100d));

protected override void OnKinectCursorMove(object sender, KinectCursorEventArgs e)

{

if (e.Z < handDepth - PushThreshold)

{

RaiseEvent(new RoutedEventArgs(ClickEvent));

}

}

protected override void OnKinectCursorEnter(object sender, KinectCursorEventArgs e)

{

handDepth = e.Z;

}

}

**2.3 磁性按钮(Magnet Button)**

如前面所讨论的，磁性按钮是对悬浮按钮的一种改进。他对用户悬浮在按钮上的这一体验进行了一些改进。他试图追踪用户手的位置，然后自动将光标对齐到磁性按钮的中间。当用户的手离开磁性按钮的区域是，手势追踪又恢复正常。在其他方面磁性按钮和悬浮按钮的行为一样。考虑到磁性按钮和悬浮按钮在功能方面差异很小，而我们将他单独作为一个完全不同的控件来对待可能有点奇怪。但是，在用户体验设计领域(UX)，这一点差异就是一个完全不同的概念。从编码角度看，这一点功能性的差异也使得代码更加复杂。

首先，创建一个继承自HoverButton的名为MagnetButton的类。磁性按钮需要一些额外的事件和属性来管理手进入到磁性按钮区域和手自动对齐到磁性按钮中间区域的时间。我们需要在KinectInput类中添加新的lock和unlock事件，代码如下：

public static readonly RoutedEvent KinectCursorLockEvent = EventManager.RegisterRoutedEvent("KinectCursorLock", RoutingStrategy.Bubble,

typeof(KinectCursorEventHandler), typeof(KinectInput));

public static void AddKinectCursorLockHandler(DependencyObject o, KinectCursorEventHandler handler)

{

((UIElement)o).AddHandler(KinectCursorLockEvent, handler);

}

public static readonly RoutedEvent KinectCursorUnlockEvent = EventManager.RegisterRoutedEvent("KinectCursorUnlock", RoutingStrategy.Bubble,

typeof(KinectCursorEventHandler), typeof(KinectInput));

public static void RemoveKinectCursorUnlockHandler(DependencyObject o, KinectCursorEventHandler handler)

{

((UIElement)o).RemoveHandler(KinectCursorUnlockEvent, handler);

}

public class MagnetButton : HoverButton

{

protected bool isLockOn = true;

public static readonly RoutedEvent KinectCursorLockEvent = KinectInput.KinectCursorUnlockEvent.AddOwner(typeof(MagnetButton));

public static readonly RoutedEvent KinectCursorUnlockEvent = KinectInput.KinectCursorLockEvent.AddOwner(typeof(MagnetButton));

private Storyboard move;

public event KinectCursorEventHandler KinectCursorLock

{

add { base.AddHandler(KinectCursorLockEvent, value); }

remove { base.RemoveHandler(KinectCursorLockEvent, value); }

}

public event KinectCursorEventHandler KinectCursorUnLock

{

add { base.AddHandler(KinectCursorUnlockEvent, value); }

remove { base.RemoveHandler(KinectCursorUnlockEvent, value); }

}

public double LockInterval

{

get { return (double)GetValue(LockIntervalProperty); }

set { SetValue(LockIntervalProperty, value); }

}

public static readonly DependencyProperty LockIntervalProperty =

DependencyProperty.Register("LockInterval", typeof(double), typeof(MagnetButton), new UIPropertyMetadata(200d));

public double UnlockInterval

{

get { return (double)GetValue(UnlockIntervalProperty); }

set { SetValue(UnlockIntervalProperty, value); }

}

public static readonly DependencyProperty UnlockIntervalProperty =

DependencyProperty.Register("UnlockInterval", typeof(double), typeof(MagnetButton), new UIPropertyMetadata(80d));

……}

磁性按钮的代码中，核心地方在于光标从当前位置移动到磁性按钮的中心位置。看起来很简单，实际上实现起来有点麻烦。需要重写基类中的OnKinectCursorEnter和OnKinectCursorLeave方法。确定磁性按钮的锁定位置第一步需要找到磁性按钮本身所处的位置。代码如下，我们使用WPF中最常见名为FindAncestor帮助方法来遍历可视化对象树来进行查找，需要找到承载该磁性按钮的Windows对象，匹配磁性按钮的当前实例到Windows上，然后将其赋给名为Point的变量。但是point对象只保存了当前磁性按钮的左上角的位置。所以，我们需要给在这个点上加一个磁性按钮一半长宽的偏移值，才能获取到磁性按钮的中心位置x,y。

private T FindAncestor<T>(DependencyObjectdependencyObject) whereT:class   
{   
 DependencyObjecttarget=dependencyObject;   
 do   
 {   
 target=VisualTreeHelper.GetParent(target);   
 }   
 while(target!=null&&!(target isT));   
 returntarget asT;   
}   
  
protected override void OnKinectCursorEnter(objectsender, KinectCursorEventArgse)   
{   
 //获取按钮位置   
 varrootVisual=FindAncestor<Window>(this);   
 varpoint=this.TransformToAncestor(rootVisual).Transform(newPoint(0,0));   
  
 varx=point.X+this.ActualWidth/2;   
 vary=point.Y+this.ActualHeight/2;   
  
 varcursor=e.Cursor;   
 cursor.UpdateCursor(newPoint(e.X,e.Y),true);   
  
 //找到目的位置   
 PointlockPoint=newPoint(x-cursor.CursorVisual.ActualWidth/2,y-cursor.CursorVisual.ActualHeight/2);   
 //当前位置   
 PointcursorPoint=newPoint(e.X-cursor.CursorVisual.ActualWidth/2,e.Y-cursor.CursorVisual.ActualHeight/2);   
 //将光标从当前位置传送到目的位置   
 AnimateCursorToLockPosition(e,x,y,cursor,reflockPoint,refcursorPoint);   
 base.OnKinectCursorEnter(sender,e);   
}   
  
protected override void OnKinectCursorLeave(objectsender, KinectCursorEventArgse)   
{   
 base.OnKinectCursorLeave(sender, e);   
 e.Cursor.UpdateCursor(newPoint(e.X,e.Y),false);   
  
 varrootVisual=FindAncestor<Window>(this);   
 varpoint=this.TransformToAncestor(rootVisual).Transform(newPoint(0,0));   
  
 varx=point.X+this.ActualWidth/2;   
 vary=point.Y+this.ActualHeight/2;   
  
 varcursor=e.Cursor;   
   
 //找到目的位置   
 PointlockPoint=newPoint(x-cursor.CursorVisual.ActualWidth/2,y-cursor.CursorVisual.ActualHeight/2);   
 //当前位置   
 PointcursorPoint=newPoint(e.X-cursor.CursorVisual.ActualWidth/2,e.Y-cursor.CursorVisual.ActualHeight/2);   
   
 AnimateCursorAwayFromLockPosition(e,cursor,reflockPoint,refcursorPoint);   
}

接下来，我们用手所在的X,Y位置替换手势图标的位置。然而，我们也传入了第二个参数，告诉手势图标自动停止追踪手的位置一段时间。当用户看到光标不听手的使唤自动对齐到磁性按钮的中心，这可能有点不太友好。

虽然我们现在有了磁性按钮的中心位置，但是我们仍不能很好的将手势光标定位到中心。我们必须额外的给手势光标本身给一个一半长宽的偏移值，以使得手在光标的中心位置而不是在左上角。在完成这些操作之后，我们将最终的值赋给lockPoint变量。我们也执行了同样的操作来查找光标目前的左上角位置以及偏移量，并将其赋值给cursorPoint变量。有了这两个值，我们就可以从当前的位置使用动画移动到目标位置了。动画方法代码如下：

private void AnimateCursorAwayFromLockPosition(KinectCursorEventArgse,CursorAdornercursor,refPointlockPoint,refPointcursorPoint)   
{   
 DoubleAnimationmoveLeft = newDoubleAnimation(lockPoint.X, cursorPoint.X, newDuration(TimeSpan.FromMilliseconds(UnlockInterval)));   
 Storyboard.SetTarget(moveLeft, cursor.CursorVisual);   
 Storyboard.SetTargetProperty(moveLeft, newPropertyPath(Canvas.LeftProperty));   
 DoubleAnimationmoveTop = newDoubleAnimation(lockPoint.Y, cursorPoint.Y, newDuration(TimeSpan.FromMilliseconds(UnlockInterval)));   
 Storyboard.SetTarget(moveTop, cursor.CursorVisual);   
 Storyboard.SetTargetProperty(moveTop, newPropertyPath(Canvas.TopProperty));   
 move = newStoryboard();   
 move.Children.Add(moveTop);   
 move.Children.Add(moveLeft);   
 move.Completed += delegate{   
 move.Stop(cursor);   
 cursor.UpdateCursor(newPoint(e.X, e.Y), false);   
 this.RaiseEvent(newKinectCursorEventArgs(KinectCursorUnlockEvent, newPoint(e.X, e.Y), e.Z) { Cursor = e.Cursor });   
 };   
 move.Begin(cursor, true);   
}   
  
private voidAnimateCursorToLockPosition(KinectCursorEventArgse,doublex,doubley,CursorAdornercursor,refPointlockPoint,refPointcursorPoint)   
{   
 DoubleAnimationmoveLeft=newDoubleAnimation(cursorPoint.X,lockPoint.X,newDuration(TimeSpan.FromMilliseconds(LockInterval)));   
 Storyboard.SetTarget(moveLeft,cursor.CursorVisual);   
 Storyboard.SetTargetProperty(moveLeft,newPropertyPath(Canvas.LeftProperty));   
   
 DoubleAnimationmoveTop=newDoubleAnimation(cursorPoint.Y,lockPoint.Y,newDuration(TimeSpan.FromMilliseconds(LockInterval)));   
 Storyboard.SetTarget(moveTop,cursor.CursorVisual);   
 Storyboard.SetTargetProperty(moveTop,newPropertyPath(Canvas.TopProperty));   
 move=newStoryboard();   
 move.Children.Add(moveTop);   
 move.Children.Add(moveLeft);   
 move.Completed+=delegate   
 {   
 this.RaiseEvent(newKinectCursorEventArgs(KinectCursorLockEvent,newPoint(x,y),e.Z){Cursor=e.Cursor});   
 };   
 if(move!=null)   
 move.Stop(e.Cursor);   
 move.Begin(cursor,false);   
}

在上面的lock和unlock动画中，我们等到动画结束时触发KinectCursorLock和KinectCursorUnlock事件。对于磁性按钮本身，这些事件用处不大。但是在后面可以给磁性幻灯片按钮提供一些帮助。

**2.4 划动(Swipe)**

划动手势和挥手(wave)手势类似。识别划动手势需要不断的跟踪用户手部运动，并保持当前手的位置之前的手的位置。因为手势有一个速度阈值，我们需要追踪手运动的时间以及在三维空间中的坐标。下面的代码展示了存储手势位置点的X,Y,Z坐标以及时间值。如果熟悉图形学中的矢量计算，可以将这个认为是一个四维向量。将下面的结构添加到类库中。

public struct GesturePoint   
{   
 public double X { get; set; }   
 public double Y { get; set; }   
 public double Z { get; set; }   
 public DateTime T { get; set; }   
  
 public override bool Equals(object obj)   
 {   
 var o = (GesturePoint)obj;   
 return (X == o.X) && (Y == o.Y) && (Z == o.Z)&&(T==o.T);   
 }   
 public override int GetHashCode()   
 {   
 return base.GetHashCode();   
 }   
}

我们将在KinectCursorManager对象中实现划动手势识别的逻辑，这样在后面的磁吸幻灯片按钮中就可以复用这部分逻辑。实现代码如下，代码中为了支持划动识别，需要向KinectCurosrManager对象中添加几个字段。GesturePoints集合存储路径上的所有点，虽然我们会一边移除一些点然后添加新的点，但是该集合不可能太大。SwipeTime和swipeDeviation分别提供了划动手势经历的时间和划动手势在y轴上的偏移阈值。划动手势经历时间过长和划动手势路径偏移y值过大都会使得划动手势识别失败。我们会移除之前的路径上的点，然后添加新的划动手势上的点。SwipeLength提供了连续划动手势的阈值。我们提供了两个事件来处理划动手势识别成功和手势不合法两种情况。考虑到这是一个纯粹的手势，与GUI界面无关，所以在实现过程中不会使用click事件。

private List<GesturePoint> gesturePoints;   
private bool gesturePointTrackingEnabled;   
private double swipeLength, swipeDeviation;   
private int swipeTime;   
public event KinectCursorEventHandler swipeDetected;   
public event KinectCursorEventHandler swipeOutofBoundDetected;   
  
private double xOutOfBoundsLength;   
private static double initialSwipeX;

xOutOfBoundsLength和initialSwipeX用来设置划动手势的开始位置。通常，我们并不关心挥划动手势的开始位置，只用在gesturePoints中寻找一定数量连续的点，然后进行模式匹配就可以了。但是有时候，我们只从某一个划动开始点来进行划动识别也很有用。例如如果在屏幕的边缘，我们实现水平滚动，在这种情况下，我们需要一个偏移阈值使得我们可以忽略在屏幕外的点，因为这些点不能产生手势。

下面的代码展示了一些帮助方法以及公共属性来管理手势追踪。GesturePointTrackingInitialize方法用来初始化各种手势追踪的参数。初始化好了划动手势之后，需要调用GesturePointTrackingStart方法。自然需要一个相应的GesturePointTrackingStop方法来结束挥动手势识别。最后我们需要提供两个重载的帮助方法ResetGesturePoint来管理一系列的我们不需要的手势点。

public void GesturePointTrackingInitialize(double swipeLength, double swipeDeviation, int swipeTime, double xOutOfBounds)   
{   
 this.swipeLength = swipeLength; this.swipeDeviation = swipeDeviation;   
 this.swipeTime = swipeTime;   
 this.xOutOfBoundsLength = xOutOfBounds;   
}   
  
public void GesturePointTrackingStart()   
{   
 if (swipeLength + swipeDeviation + swipeTime == 0)   
 throw new InvalidOperationException("挥动手势识别参数没有初始化！");   
 gesturePointTrackingEnabled = true;   
}   
  
public void GesturePointTrackingStop()   
{   
 xOutOfBoundsLength = 0;   
 gesturePointTrackingEnabled = false;   
 gesturePoints.Clear();   
}   
  
public bool GesturePointTrackingEnabled   
{   
 get { return gesturePointTrackingEnabled ; }   
}   
  
private void ResetGesturePoint(GesturePoint point)   
{   
 bool startRemoving = false;   
 for (int i= gesturePoints.Count; i >=0; i--)   
 {   
 if (startRemoving)   
 gesturePoints.RemoveAt(i);   
 else   
 if (gesturePoints[i].Equals(point))   
 startRemoving = true;   
 }   
}   
  
private void ResetGesturePoint(int point)   
{   
 if (point < 1)   
 return;   
 for (int i = point-1; i >=0; i--)   
 {   
 gesturePoints.RemoveAt(i);   
 }   
}

划动(swipe)手势识别的核心算法在HandleGestureTracking方法中，代码如下。将KinectCursorManager中的UpdateCursor方法和Kinect中的骨骼追踪事件绑定。每一次当获取到新的坐标点时，HandGestureTracking方法将最新的GesturePoint数据添加到gesturePoints集合中去。然后执行一些列条件检查，首先判断新加入的点是否以手势开始位置为起点参考，偏离Y轴过远。如果是，抛出一个超出范围的事件，然后将所有之前累积的点清空，然后开始下一次的划动识别。其次，检查手势开始的时间和当前的时间，如果时间差大于阈值，那么移除开始处手势点，然后将紧接着的点作为手势识别的起始点。如果新的手的位置在这个集合中，就很好。紧接着，判断划动起始点的位置和当前位置的X轴上的距离是否超过了连续划动距离的阈值，如果超过了，则触发SwipeDetected事件，如果没有，我们可以有选择性的判断，当前位置的X点是否超过了划动识别的最大区间返回，然后触发对于的事件。然后我们等待新的手部点传到HandleGestureTracking方法中去。

private void HandleGestureTracking(float x, float y, float z)

{

if (!gesturePointTrackingEnabled)

return;

// check to see if xOutOfBounds is being used

if (xOutOfBoundsLength != 0 && initialSwipeX == 0)

{

initialSwipeX = x;

}

GesturePoint newPoint = new GesturePoint() { X = x, Y = y, Z = z, T = DateTime.Now };

gesturePoints.Add(newPoint);

GesturePoint startPoint = gesturePoints[0];

var point = new Point(x, y);

//check for deviation

if (Math.Abs(newPoint.Y - startPoint.Y) > swipeDeviation)

{

//Debug.WriteLine("Y out of bounds");

if (swipeOutofBoundDetected != null)

swipeOutofBoundDetected(this, new KinectCursorEventArgs(point) { Z = z, Cursor = cursorAdorner });

ResetGesturePoint(gesturePoints.Count);

return;

}

if ((newPoint.T - startPoint.T).Milliseconds > swipeTime) //check time

{

gesturePoints.RemoveAt(0);

startPoint = gesturePoints[0];

}

if ((swipeLength < 0 && newPoint.X - startPoint.X < swipeLength) // check to see if distance has been achieved swipe left

|| (swipeLength > 0 && newPoint.X - startPoint.X > swipeLength)) // check to see if distance has been achieved swipe right

{

gesturePoints.Clear();

//throw local event

if (swipeDetected != null)

swipeDetected(this, new KinectCursorEventArgs(point) { Z = z, Cursor = cursorAdorner });

return;

}

if (xOutOfBoundsLength != 0 &&

((xOutOfBoundsLength < 0 && newPoint.X - initialSwipeX < xOutOfBoundsLength) // check to see if distance has been achieved swipe left

|| (xOutOfBoundsLength > 0 && newPoint.X - initialSwipeX > xOutOfBoundsLength))

)

{

if (swipeOutofBoundDetected != null)

swipeOutofBoundDetected(this, new KinectCursorEventArgs(point) { Z = z, Cursor = cursorAdorner });

}

}

**2.5 磁性幻灯片(Magnetic Slide)**

磁性幻灯片是Kinect手势中的精华(holy grail)。他由Harmonix公司的交互设计师们在开发《舞林大会》（Dance Central）这一款游戏时创造的。最初被用在菜单系统中，现在作为一种按钮在很多地方有应用，包括Xbox自身的操作面板。他比磁性按钮好的地方就是，不需要用户等待一段时间。在Xbox游戏中，没有人愿意去等待。而下压按钮又有自身的缺点，最主要的是用户体验不是很好。磁性幻灯片和磁性按钮一样，一旦用户进入到按钮的有效区域，光标就会自定锁定到某一点上。但是在这一点上，可以有不同的表现。除了悬停在按钮上方一段时间触发事件外，用户可以划动收来激活按钮。

从编程角度看，磁性幻灯片基本上是磁性按钮和划动手势(swipe)的组合。要开发一个磁性幻灯片按钮，我们可以简单的在可视化树中的悬浮按钮上声明一个计时器，然后再注册滑动手势识别事件。下面的代码展示了磁性幻灯片按钮的基本结构。其构造函数已经在基类中为我们声明好了计时器。InitializeSwipe和DeinitializeSwipe方法负责注册KinectCursorManager类中的滑动手势识别功能。

public class MagneticSlide:MagnetButton

{

private bool isLookingForSwipes;

public MagneticSlide()

{

base.isLockOn = false;

}

private void InitializeSwipe()

{

if (isLookingForSwipes)

return;

var kinectMgr = KinectCursorManager.Instance;

kinectMgr.GesturePointTrackingInitialize(SwipeLength, MaxDeviation, MaxSwipeTime, xOutOfBoundsLength);

kinectMgr.swipeDetected += new KinectCursorEventHandler(kinectMgr\_swipeDetected);

kinectMgr.swipeOutofBoundDetected += new KinectCursorEventHandler(kinectMgr\_swipeOutofBoundDetected);

kinectMgr.GesturePointTrackingStart();

}

private void DeInitializeSwipe()

{

var KinectMgr = KinectCursorManager.Instance;

KinectMgr.swipeDetected -= new KinectCursorEventHandler(kinectMgr\_swipeDetected);

KinectMgr.swipeOutofBoundDetected -= new KinectCursorEventHandler(kinectMgr\_swipeOutofBoundDetected);

KinectMgr.GesturePointTrackingStop();

isLookingForSwipes = false;

}

另外，我们也需要将控件的滑动手势的初始化参数暴露出来，这样就可以根据特定的需要进行设置了。下面的代码展示了SwipeLength和XOutOfBoundsLength属性，这两个都是默认值的相反数。这是因为磁性幻灯片按钮一般在屏幕的右侧，需要用户向左边划动，因此，相对于按钮位置的识别偏移以及边界偏移是其X坐标轴的相反数。

public static readonly DependencyProperty SwipeLengthProperty =  
 DependencyProperty.Register("SwipeLength", typeof(double), typeof(MagneticSlide), new UIPropertyMetadata(-500d));  
  
public double SwipeLength  
{  
 get { return (double)GetValue(SwipeLengthProperty); }  
 set { SetValue(SwipeLengthProperty, value); }  
}  
  
public static readonly DependencyProperty MaxDeviationProperty =  
 DependencyProperty.Register("MaxDeviation", typeof(double), typeof(MagneticSlide), new UIPropertyMetadata(100d));  
  
public double MaxDeviation  
{  
 get { return (double)GetValue(MaxDeviationProperty); }

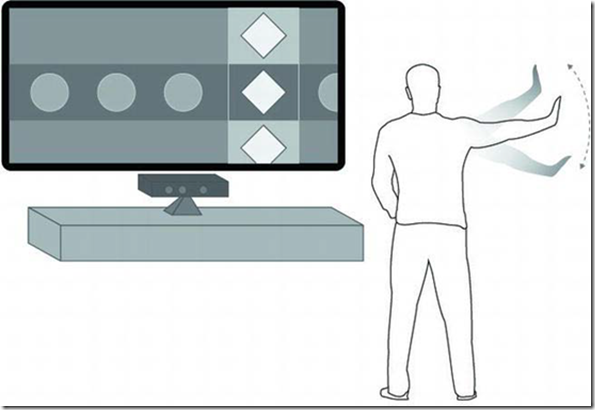
set { SetValue(MaxDeviationProperty, value); }  
}  
  
public static readonly DependencyProperty XOutOfBoundsLengthProperty =  
 DependencyProperty.Register("XOutOfBoundsLength", typeof(double), typeof(MagneticSlide), new UIPropertyMetadata(-700d));  
  
public double XOutOfBoundsLength  
{  
 get { return (double)GetValue(XOutOfBoundsLengthProperty); }  
 set { SetValue(XOutOfBoundsLengthProperty, value); }  
}  
  
public static readonly DependencyProperty MaxSwipeTimeProperty =  
 DependencyProperty.Register("MaxSwipeTime", typeof(int), typeof(MagneticSlide), new UIPropertyMetadata(300));  
public int MaxSwipeTime  
{  
 get { return (int)GetValue(MaxSwipeTimeProperty); }  
 set { SetValue(MaxSwipeTimeProperty, value); }  
}

要实现磁性幻灯片按钮的逻辑，我们只需要处理基类中的enter事件，以及划动手势识别事件即可。我们不会处理基类中的leave事件，因为当用户做划动手势时，极有可能会不小心触发leave事件。我们不想破坏之前初始化好了的deactivate算法逻辑，所以取而代之的是，我们等待要么下一个划动识别成功，要么在关闭划动识别前划动手势超出识别范围。当探测到划动时，触发一个标准的click事件。

public static readonly RoutedEvent SwipeOutOfBoundsEvent = EventManager.RegisterRoutedEvent("SwipeOutOfBounds", RoutingStrategy.Bubble,  
typeof(KinectCursorEventHandler), typeof(KinectInput));  
  
public event RoutedEventHandler SwipeOutOfBounds  
{  
 add { AddHandler(SwipeOutOfBoundsEvent, value); }  
 remove { RemoveHandler(SwipeOutOfBoundsEvent, value); }  
}  
  
void KinectMgr\_swipeOutofBoundDetected(object sender, KinectCursorEventArgs e)  
{  
 DeInitializeSwipe();  
 RaiseEvent(new KinectCursorEventArgs(SwipeOutOfBoundsEvent));  
}  
  
void KinectMgr\_swipeDetected(object sender, KinectCursorEventArgs e)  
{  
 DeInitializeSwipe();  
 RaiseEvent(new RoutedEventArgs(ClickEvent));  
}  
  
protected override void OnKinectCursorEnter(object sender, KinectCursorEventArgs e)  
{  
 InitializeSwipe();  
 base.OnKinectCursorEnter(sender, e);  
}

**2.6 垂直滚动条(Vertical Scroll)**

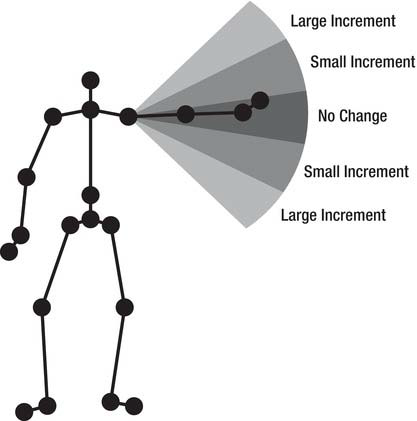
并不是所有的内容都能够在一屏之内显示完。有时候可能有一些内容会大于屏幕的实际尺寸，这就需要用户来滚动屏幕或者列表控件来显示在屏幕之外的内容。传统上，垂直滚动条一直是交互界面设计的一个禁忌。但是垂直滚动条在划动触摸界面中得到了很好的应用。所以Xbox和Sony PlayStation系统中都使用了垂直滚动条来构建菜单。Harmonix’s的《舞林大会》（Dance Central）这一系列游戏使用了垂直滚动条式的菜单系统。Dance Central第一次成功的使用了垂直滚动界面作为手势交互界面。在下面的手势交互图中，当用户抬起或者放下手臂时会使得屏幕的内容垂直滚动。胳膊远离身体，抬起手臂会使得屏幕或者菜单从下往上移动，放下手臂会使得从上往下移动。

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204280001268101.png)

水平的划动在Kinect应用中似乎很常见(尤其是在Metro风格的Xbox游戏交互界面中，水平划动是占主导的手势)，但是垂直滚动用户体验更加友好，也是用户交互界面更好的选择。水平或者垂直划动手势有一些小的用户体验问题。另外，划动手势在识别上也较困难，因为挥动的形式和动作因人而异，且差别很大。就算同一个人，划动手势也不是一直不变的。划动手势在触摸屏设备上能够较好的工作是因为除非不触摸到屏幕，那么动作就不会发生。但是在手势识别界面上，用户的手是和视觉元素进行交互的，这时手就是在某一特定的坐标空间中的视觉元素。

当用户做划动手势时，在整个手的划动过程中会手的位置在水平方向会保持相对一致。这就使得如果想进行多次连续的划动手势时会产生一些问题。有时候会产生一些比较尴尬的场景，那就是会无意中撤销前一次的划动手势。例如，用户使用右手从右向左进行划动手势，使得页面会跳转到下一页，现在用户的右手在身体的左边位置了，然后用户想将手移动回原始的开始位置以准备下一次的从右向左的挥动手势。但是，如果用于依然保持手在水平位置大致一致的话，应用程序会探测到一次从左向右的划动操作然后又将界面切换到了之前的那一页。这就使得用户必须创建一个循环的运动来避免不必要的误读。更进一步，频繁的划动手势也容易使得用户疲劳，而垂直方向的划动也只会加剧这一问题。

但是垂直滚动条则不会有上述的这些用户体验上的缺点。他比较容易使用，对用户来说也更加友好，另外，用户也不需要为了保持手在水平或者垂直方向一致而导致的疲劳。从技术方面来讲，垂直滚动操作识别较划动识别简单。垂直滚动在技术上是一个姿势而不是手势。滚动操作的探测是基于当前手臂的位置而不是手臂的运动。滚动的方向和大小由手臂和水平方向的夹角来确定。下图演示了垂直滚动。



使用之前的[姿势识别](http://www.cnblogs.com/yangecnu/archive/2012/04/16/KinectSDK_AdvanceSkeletonTracking_part2.html)那篇文章中的内容，我们能够计算从用户的身体到肩部和手腕的夹角，定义一个角度区间作为中间姿势，当用户手臂在这一区间内时，不会产生任何动作，如上图中的，当手臂自然处于-5度或者355度时，作为偏移的零点。建议在实际开发中，将零点的偏移上下加上20度左右。当用户的手臂离开这一区域时，离开的夹角及变化的幅度根据需求而定。但是建议至少在0度区间上下有两个区间来表示小幅和大幅的增加。这使得能够更好的实现传统的人机交互界面中的垂直滚动条的逻辑。

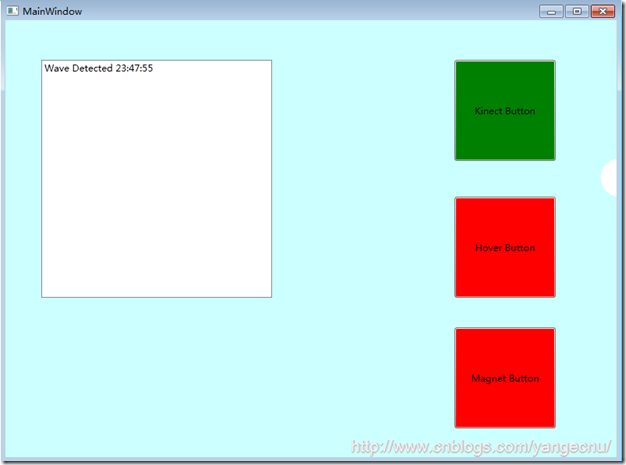
**2.7 通用暂停按钮(Universal Pause)**

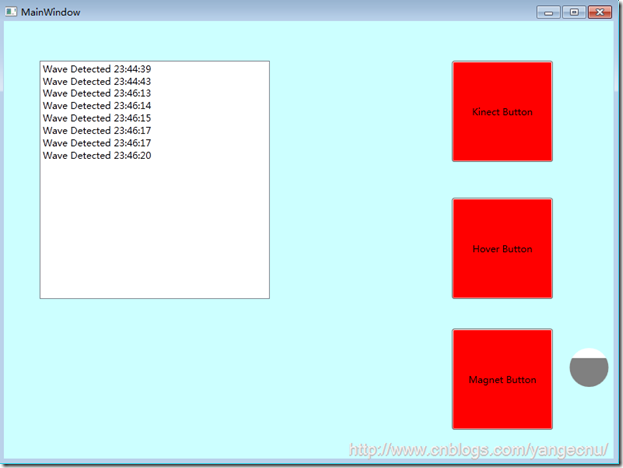
暂停按钮，通常作为引导手势或者退出手势，是微软建议在给用户提供引导时很少的几个手势之一。这个手势是通过将左臂保持和身体45度角来完成的。在很多Kinect的游戏中都使用到了这一手势，用来暂停动作或者唤出Xbox菜单。和本文之前介绍的手势不一样，这个手势并没有什么符号学上的含义，是一个认为设计的动作。通用暂停手势很容易实现，也不一定要限制手臂，并且不容易和其他手势混淆。

通用暂停手势的识别和垂直滚动手势的识别有点类似，就是计算左臂和身体的夹角，然后加上一个阈值即可，相信很简单，在这里就不再赘述了。

**2.8测试Demo**

结合前篇文章中的挥动(wave)手势识别，以及上文将的几种按钮，做了一个小的例子，使用之前开发的手势识别库，以及手势识别按钮。这部分代码很简单，直接引用之前在类库中定义好的控件即可。大家可以下载本文后面的代码自己回去实验一下。 截图如下：

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204280001264546.png)

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/20120428000126675.png)

**3. 手势识别的未来**

我们进入商场可能对各种商品不会有很深的印象，同样，随着时间的流逝，Kinect也会变得不那么令人印象深刻，甚至被大多数人忘记。但是，随着软硬件技术的发展，Kinect或者相似的技术会融入到生活的各个方面，让你看不到Kinect的存在。

当我们进入一个购物商场时，当你靠近入口时，门会自动打开。这在当时很让人印象深刻，但没有人会注意，观察甚至意识到这一特性。在未来的某一天，Kinect也会像自动门一样融入生活的方方面面。使得我们感受不到他的存在。

Kinect以及NUI的世界才刚刚开始，随着时间的推移，这种交互体验会发生巨大变化。在电影《少数派报告》(Minority Report)中，汤姆克鲁斯使用手势浏览和打开大屏幕上的各种文件和影像资料，这一场景现在已经是基于Kinect的应用程序的要实现目标之一。 有时候，科幻片中比现实有更好的想像力，也能提供比现实生活中更好的科技。在星际迷航(Star Trek),星球大战(Star Wars)或者 2001：太空漫游(2001: A Space Odyssey)这些科幻电影中，电脑能够看到和感应人的存在。在这些电影中，用户使用语音和手势无缝的和电脑进行交互。当然这种交互也有负面因素，应该设置一些限制。

虽然科幻电影变成现实的这一前景会引发一些理性的担忧，但是这种变化正在到来。意识到这种变化带来的好处很重要。Kinect及其类似的技术使得我们的生活环境更加智能化。它使得应用程序能够识别用户的手势，进而能够分析出用户的意图，而不需要用户明确的给出这些信息或者指令。现在的Kinect游戏是基于应用程序查找特定的用户的手势而进行操作的，用户必须主动的和应用程序进行交流或者发送指令。但是，还是有很多用户的信息没有被捕捉和处理。如果应用程序能够探测到其他信息，确切的说，如用户的情绪，那么就可以提供更加人性化的定制服务了。现在我们所识别的姿势都很简单，我们只是在学习如何建立交互界面，可以预见在未来，随着基于手势交互的应用程序越来越多，这种用户界面就会消失，就像移动设备中触摸界面逐渐取代光标那样。

想象一下，下班后回到家，走到卧室，说一句“电脑，放点music”。于是电脑就会识别语音指令开始播放音乐。但是，电脑也能够识别到你工作了一天，需要一些音乐来放松心情，于是电脑自动的根据你的心情选择一些歌曲。语音成了首要的发送指令的形式，手势来对一些指令进行增强。在上面的例子中，电脑能够根据你的身体语言，识别你的情绪，这样，手势是一种主动的，有上下文情景的和电脑进行交互的方法。这并不意味这手势变得不那么重要，相反重要性增加了，只不过是以一种间接的方式。

如今有一些声控传感器，例如能够根据声音探测到人进入到房间，然后开灯或者关灯的声控开关。这是一种比较笨拙的系统，他没有上下文提供。如果使用Kinect技术，它能够识别用户的运动，然后根据情形调整灯光的亮度。例如，如果在凌晨2点，你想起床喝点水，电脑可能会将灯光调整的比较暗，以至于不会太刺眼。但是如果某一天你凌晨2点钟从外面回来，Kinect识别到你是清醒的，就会把灯全部打开。

目前，Kinect仍然是一种比较新的技术，我们仍然试图理解然后能够更好的发挥它的潜能。在最开始Kinect出来的时候只是观看或者体验。随着他作为Xbox外设的发布，游戏的主题也有了一些限制。大多数游戏都是运动型的，这些游戏都只能识别用户的一些基本手势，如跑，跳，踢，划动，扔等这些手势或动作。早期的Kinect体感游戏也只有一些简单的菜单系统，使用手来进行操作。

虽然用户体验设计发生了巨大变化，但是目前基于手势的游戏和应用比较简单。我们仍然处在学习如何定义和识别手势的阶段。这使得我们的手势有些粗糙。我们仍需要扩大手势的幅度才能达到好的识别效果。当我们能够识别到手势的细微方面时，应用程序所带来的沉浸感将会大大提升。

现在的足球游戏只能识别到基本的踢球动作，游戏不能够判断用户是使用脚趾，脚背，脚踝还是脚跟跟球进行交互的。这些不同的姿势可能对球产生完全不同的影响，从而使得游戏产生不同的结果。更进一步，游戏应该能够根据用户踢球的动作，脚的位置，应用一些物理特性，给予球一些真实的加速度，旋转，弧度等特性，这样会使得游戏更加真实，玩家也更加有沉浸感。

目前的这些限制主要是由Kinect摄像头的分辨率决定的。下一代的Kinect硬件设备可能会使用更高分辨率的摄像头来提供更好的深度影像数据。微软已经放出了关于第二代Kinect硬件方面的相关信息。这使得更精确的手势识别变为可能，给基于Kinect的应用程序开发带来了两个方面的改进。首先是骨骼关节精度的提升，这不但能够提升手势识别的精度，也能够扩大手势识别的种类。另一个改进是使得能够产生一些额外的关节点，如手指的信息，以及一些非关节点如嘴唇，鼻子，耳朵，眼睛等位置信息。如今这些信息都能够识别的出来，只是需要使用一些第三方的类库，官方的SDK中没有原生的对这些特征进行支持。

对手指进行追踪和识别能够大大的提高符号语言的作用。如果应用程序能够识别手指的运动，用户就能够使用手指进行更高精度和更自然的操作。手指手势交互信息很复杂，在用户进行交流的过程中能够提供丰富的上下文信息。即使能够识别到手指手势，今天的基于Kinect的间交互体验也没有发生革命性的变化，这是因为用户依然必须意识到Kinect的存在，还要知道如何和这些硬件交互，需要做什么样的手势才能达到想要的结果。当你看别人玩Kinect游戏时，你会注意到他是如何在Kinect前面做动作的。用户的姿势有时候比较僵硬和不自然。很多姿势并不能识别，一些需要重复多次，更坏的情况下姿势会被错误识别，这样就会导致一些意想不到的操作。更进一步，用户的手势通常需要过分的夸大才能被kinect识别的到。这些问题可能都是暂时的。在未来，随着软硬件的不断升级和改进，在使用基于手势的人机交互界面时，用户会越来越感到舒服和自然。从这方面讲，Kinect是一个神奇的设备，以至于会像自动门那样被广大的用户所知道和接受。

**4.结语**

在[上一篇](http://www.cnblogs.com/yangecnu/archive/2012/04/21/KinectSDK_GesturesDetection_part1_BasicConception.html)文章介绍手势识别概念和识别方法的基础上，本文进一步解释了如何进行手势识别。首先，构建了一个基本的手势识别的框架，然后在此基础上对常用的8中手势中的剩余7中手势进行逐一讲解与实现。最后展望了Kienct未来在手势识别方面的前景和应用。希望这些知识对您了解和掌握Kinect SDK手势识别有所帮助！

[上一篇](http://www.cnblogs.com/yangecnu/archive/2012/04/21/KinectSDK_GesturesDetection_part1_BasicConception.html)文章和本篇文章的代码点击[此处](http://files.cnblogs.com/yangecnu/KinectSDK_GestureDetection.rar)下载。下一篇文章将会介绍Kinect中的语音识别相关的内容，敬请期待！