[**Kinect for Windows SDK开发入门(七)：骨骼追踪基础 下**](http://www.cnblogs.com/yangecnu/archive/2012/04/09/KinectSDK_Skeleton_Tracking_Part2.html)

[上一篇](http://www.cnblogs.com/yangecnu/archive/2012/04/06/Kinect-SDK_Skeleton_Tracking_Part1.html)文章用在UI界面上绘制骨骼数据的例子展示了骨骼追踪系统涉及的主要对象，然后详细讨论了骨骼追踪中所涉及的对象模型。但是了解了基本概念是一回事，能够建立一个完整的可用的应用程序又是另外一回事，本文通过介绍一个简单的Kinect游戏来详细讨论如何应用这些对象来建立一个完整的Kinect应用，以加深对Kinect骨骼追踪所涉及的各个对象的了解。

**1. Kinect连线游戏**

相信大家在小时候都做过一个数学题目，就是在纸上将一些列数字(用一个圆点表示)从小到大用线连起来。游戏逻辑很简单，只不过我们在这里要实现的是动动手将这些点连起来，而不是用笔或者鼠标。

这个小游戏显然没有第一人称射击游戏那样复杂，但如果能够做成那样更好。我们要使用骨骼追踪引擎来收集游戏者的关节数据，执行操作并渲染UI界面。这个小游戏展示了自然用户界面（Natural User Interface，NUI）的理念，这正是基于Kinect开发的常见交互界面，就是手部跟踪。这个连线小游戏没有仅仅用到了WPF的绘图功能，没有好看的图片和动画效果，这些以后可以逐步添加。

在开始写代码之前，需要明确定义我们的游戏目标。连线游戏是一个智力游戏，游戏者需要将数字从小到大连起来。程序可以自定义游戏上面的数字和位置（合称一个关卡）。每一个关卡包括一些列的数字(以点表示)及其位置。我们要创建一个DotPuzzle类来管理这些点对象的集合。可能一开始不需要这个类，仅仅需要一个集合就可以，但是为了以后方便添加其他功能，使用类更好一点。这些点在程序中有两个地方需要用到，一个是最开始的时候在界面上绘制关卡，其次是判断用户是否碰到了这些点。

当用户碰到点时，程序开始绘制，直线以碰到的点为起始点，直线的终点位用户碰到的下一个点。然后下一个点又作为另一条直线的起点，依次类推。直到最后一个点和第一个点连起来，这样关卡算是通过了，游戏结束。

游戏规则定义好了之后，我们就可以开始编码了，随着这个小游戏的开发进度，可能会添加一些其他的新功能。一开始，建一个WPF工程，然后引用Microsoft.Kinect.dll，和之前的项目一样，添加发现和初始化Kinect传感器的代码。然后注册KinectSensor对象的SkeletonFrameReady事件。

**1.1 游戏的用户界面**

游戏界面代码如下，有几个地方需要说明一下。Polyline对象用来表示点与点之间的连线。当用户在点和点之间移动手时，程序将点添加到Polyline对象中。PuzzleBoardElement Canvas对象用来作为UI界面上所有点的容器。Grid对象下面的Canvas的顺序是有意这样排列的，我们使用另外一个GameBoardElement Canvas对象来存储手势，以Image来表示，并且能够保证这一层总是在点图层之上。 将每一类对象放在各自层中的另外一个好处是重新开始一个新的游戏变得很容易，只需要将PuzzleBoardElement节点下的所有子节点清除，CrayonElement元素和其他的UI对象不会受到影响。

Viewbox和Grid对象对于UI界面很重要。如上一篇文章中讨论的，骨骼节点数据是基于骨骼空间的。这意味着我们要将骨骼向量转化到UI坐标系中来才能进行绘制。我们将UI控件硬编码，不允许它随着UI窗体的变化而浮动。Grid节点将UI空间大小定义为1920\*1200。通常这个是显示器的全屏尺寸，而且他和深度影像数据的长宽比是一致的。这能够使得坐标转换更加清楚而且能够有更加流畅的手势移动体验。

<Window x:Class="KinectDrawDotsGame.MainWindow"

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

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

Title="MainWindow" Height="600" Width="800" Background="White">

<Viewbox>

<Grid x:Name="LayoutRoot" Width="1920" Height="1200">

<Polyline x:Name="CrayonElement" Stroke="Black" StrokeThickness="3" />

<Canvas x:Name="PuzzleBoardElement" />

<Canvas x:Name="GameBoardElement">

<Image x:Name="HandCursorElement" Source="Images/hand.png" Width="75"

Height="75" RenderTransformOrigin="0.5,0.5">

<Image.RenderTransform>

<TransformGroup>

<ScaleTransform x:Name="HandCursorScale" ScaleX="1" />

</TransformGroup>

</Image.RenderTransform>

</Image>

</Canvas>

</Grid>

</Viewbox>

</Window>

硬编码UI界面也能够简化开发过程，能够使得从骨骼坐标向UI坐标的转化更加简单和快速，只需要几行代码就能完成操作。况且，如果不应编码，相应主UI窗体大小的改变将会增加额外的工作量。通过将Grid嵌入Viewbox节点来让WPF来帮我们做缩放操作。最后一个UI元素是Image对象，他表示手的位置。在这个小游戏中，我们使用这么一个简单的图标代表手。你可以选择其他的图片或者直接用一个Ellipse对象来代替。本游戏中图片使用的是右手。在游戏中，用户可以选择使用左手或者右手，如果用户使用左手，我们将该图片使用ScaleTransform变换，使得变得看起来像右手。

**1.2 手部追踪**

游戏者使用手进行交互，因此准确判断是那只手以及手的位置对于基于Kinect开发的应用程序显得至关重要。手的位置及动作是手势识别的基础。追踪手的运动是从Kinect获取数据的最重要用途。在这个应用中，我们将忽视其他关节点信息。

小时候，我们做这中连线时一般会用铅笔或者颜料笔，然后用手控制铅笔或则颜料笔进行连线。我们的这个小游戏颠覆了这种方式，我们的交互非常自然，就是手。这样有比较好的沉浸感，使得游戏更加有趣。当然，开发基于Kinect的应用程序这种交互显得自然显得至关重要。幸运的是，我们只需要一点代码就能实现这一点。

在应用程序中可能有多个游戏者，我们设定，不论那只手离Kinect最近，我们使用距离Kinect最近的那个游戏者的那只手作为控制程序绘图的手。当然，在游戏中，任何时候用户可以选择使用左手还是右手，这会使得用户操作起来比较舒服，SkeletonFrameReady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);

Skeleton[] dataSet2 = new Skeleton[this.frameSkeletons.Length];

frame.CopySkeletonDataTo(dataSet2);

if (skeleton == null)

{

HandCursorElement.Visibility = Visibility.Collapsed;

}

else

{

Joint primaryHand = GetPrimaryHand(skeleton);

TrackHand(primaryHand);

TrackPuzzle(primaryHand.Position);

}

}

}

}

private static Skeleton GetPrimarySkeleton(Skeleton[] skeletons)

{

Skeleton skeleton = null;

if (skeletons != null)

{

//查找最近的游戏者

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;

}

每一次事件执行时，我们查找第一个合适的游戏者。程序不会锁定某一个游戏者。如果有两个游戏者，那么靠Kinect最近的那个会是活动的游戏者。这就是GetPrimarySkeleton的功能。如果没有活动的游戏者，手势图标就隐藏。否则，我们使用活动游戏者离Kinect最近的那只手作为控制。查找控制游戏手的代码如下：

private static Joint GetPrimaryHand(Skeleton skeleton)

{

Joint primaryHand = new Joint();

if (skeleton != null)

{

primaryHand = skeleton.Joints[JointType.HandLeft];

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

if (righHand.TrackingState != JointTrackingState.NotTracked)

{

if (primaryHand.TrackingState == JointTrackingState.NotTracked)

{

primaryHand = righHand;

}

else

{

if (primaryHand.Position.Z > righHand.Position.Z)

{

primaryHand = righHand;

}

}

}

}

return primaryHand;

}

优先选择的是距离Kinect最近的那只手。但是，代码不单单是比较左右手的Z值来判断选择Z值小的那只手，如[前篇文章](http://www.cnblogs.com/yangecnu/archive/2012/04/06/Kinect-SDK_Skeleton_Tracking_Part1.html)讨论的，Z值为0表示该点的深度信息不能确定。所以，我们在进行比较之前需要进行验证，检查每一个节点的TrackingState状态。左手是默认的活动手，除非游戏者是左撇子。右手必须显示的追踪，或者被计算认为离Kinect更近。在操作关节点数据时，一定要检查TrackingState的状态，否则会得到一些异常的位置信息，这样会导致UI绘制错误或者是程序异常。

知道了哪只手是活动手后，下一步就是在界面上更新手势图标的位置了。如果手没有被追踪，隐藏图标。在一些比较专业的应用中，隐藏手势图标可以做成一个动画效果，比如淡入或者放大然后消失。在这个小游戏中只是简单的将其状态设置为不可见。在追踪手部操作时，确保手势图标可见，并且设定在UI上的X,Y位置，然后根据是左手还是右手确定UI界面上要显示的手势图标，然后更新。计算并确定手在UI界面上的位置可能需要进一步检验，这部分代码和[上一篇](http://www.cnblogs.com/yangecnu/archive/2012/04/06/Kinect-SDK_Skeleton_Tracking_Part1.html)文章中绘制骨骼信息类似。后面将会介绍空间坐标转换，现在只需要了解的是，获取的手势值是在骨骼控件坐标系中，我们需要将手在骨骼控件坐标系统中的位置转换到对于的UI坐标系统中去。

private void TrackHand(Joint hand)

{

if (hand.TrackingState == JointTrackingState.NotTracked)

{

HandCursorElement.Visibility = Visibility.Collapsed;

}

else

{

HandCursorElement.Visibility = Visibility.Visible;

DepthImagePoint point = this.kinectDevice.MapSkeletonPointToDepth(hand.Position, this.kinectDevice.DepthStream.Format);

point.X = (int)((point.X \* LayoutRoot.ActualWidth / kinectDevice.DepthStream.FrameWidth) - (HandCursorElement.ActualWidth / 2.0));

point.Y = (int)((point.Y \* LayoutRoot.ActualHeight / kinectDevice.DepthStream.FrameHeight) - (HandCursorElement.ActualHeight / 2.0));

Canvas.SetLeft(HandCursorElement, point.X);

Canvas.SetTop(HandCursorElement, point.Y);

if (hand.JointType == JointType.HandRight)

{

HandCursorScale.ScaleX = 1;

}

else

{

HandCursorScale.ScaleX = -1;

}

}

}

编译运行程序，当移动手时，手势图标会跟着移动。

**1.3 绘制游戏界面逻辑**

为了显示绘制游戏的逻辑，我们创建一个新的类DotPuzzle。这个类的最主要功能是保存一些数字，数字在集合中的位置决定了在数据系列中的前后位置。这个类允许序列化，我们能够从xml文件中读取关卡信息来建立新的关卡。

public class DotPuzzle

{

public List<Point> Dots { get; set; }

public DotPuzzle()

{

this.Dots = new List<Point>();

}

}

定义好结构之后，就可以开始将这些点绘制在UI上了。首先创建一个DotPuzzle类的实例，然后定义一些点，puzzleDotIndex用来追踪用户解题的进度，我们将puzzleDotIndex设置为-1表示用户还没有开始整个游戏，代码如下:

public MainWindow()

{

InitializeComponent();

puzzle = new DotPuzzle();

this.puzzle.Dots.Add(new Point(200, 300));

this.puzzle.Dots.Add(new Point(1600, 300));

this.puzzle.Dots.Add(new Point(1650, 400));

this.puzzle.Dots.Add(new Point(1600, 500));

this.puzzle.Dots.Add(new Point(1000, 500));

this.puzzle.Dots.Add(new Point(1000, 600));

this.puzzle.Dots.Add(new Point(1200, 700));

this.puzzle.Dots.Add(new Point(1150, 800));

this.puzzle.Dots.Add(new Point(750, 800));

this.puzzle.Dots.Add(new Point(700, 700));

this.puzzle.Dots.Add(new Point(900, 600));

this.puzzle.Dots.Add(new Point(900, 500));

this.puzzle.Dots.Add(new Point(200, 500));

this.puzzle.Dots.Add(new Point(150, 400));

this.puzzleDotIndex = -1;

this.Loaded += (s, e) =>

{

KinectSensor.KinectSensors.StatusChanged += KinectSensors\_StatusChanged;

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

DrawPuzzle(this.puzzle);

};

}

最后一步是在UI界面上绘制点信息。我们创建了一个名为DrawPuzzle的方法，在主窗体加载完成的时候触发改事件。DrawPuzzle遍历集合中的每一个点，然后创建UI元素表示这个点，然后将这个点添加到PuzzleBoardElement节点下面。另一种方法是使用XAML 创建UI界面，将DotPuzzle对象作为ItemControl的ItemSource属性，ItemsControl对象的ItemTemplate对象能够定义每一个点的外观和位置。这种方式更加优雅，他允许定义界面的风格及主体。在这个例子中，我们把精力集中在Kinect代码方面而不是WPF方面，尽量减少代码量来实现功能。如果有兴趣的话，可以尝试改为ItemControl这种形式。DrawPuzzle代码如下：

private void DrawPuzzle(DotPuzzle puzzle)

{

PuzzleBoardElement.Children.Clear();

if (puzzle != null)

{

for (int i = 0; i < puzzle.Dots.Count; i++)

{

Grid dotContainer = new Grid();

dotContainer.Width = 50;

dotContainer.Height = 50;

dotContainer.Children.Add(new Ellipse { Fill = Brushes.Gray });

TextBlock dotLabel = new TextBlock();

dotLabel.Text = (i + 1).ToString();

dotLabel.Foreground = Brushes.White;

dotLabel.FontSize = 24;

dotLabel.HorizontalAlignment = HorizontalAlignment.Center;

dotLabel.VerticalAlignment = VerticalAlignment.Center;

dotContainer.Children.Add(dotLabel);

//在UI界面上绘制点

Canvas.SetTop(dotContainer, puzzle.Dots[i].Y - (dotContainer.Height / 2));

Canvas.SetLeft(dotContainer, puzzle.Dots[i].X - (dotContainer.Width / 2));

PuzzleBoardElement.Children.Add(dotContainer);

}

}

}

**1.4 游戏逻辑实现**

到目前为止，我们的游戏已经有了用户界面和基本的数据。移动手，能够看到手势图标会跟着移动。我们要将线画出来。当游戏者的手移动到点上时，开始绘制直线的起点，然后知道手朋到下一个点时，将这点作为直线的终点，并开始另一条直线，并以该点作为起点。TrackPuzzle代码如下：

private void TrackPuzzle(SkeletonPoint position)

{

if (this.puzzleDotIndex == this.puzzle.Dots.Count)

{

//游戏结束

}

else

{

Point dot;

if (this.puzzleDotIndex + 1 < this.puzzle.Dots.Count)

{

dot = this.puzzle.Dots[this.puzzleDotIndex + 1];

}

else

{

dot = this.puzzle.Dots[0];

}

DepthImagePoint point = this.kinectDevice.MapSkeletonPointToDepth(position, kinectDevice.DepthStream.Format);

point.X = (int)(point.X \* LayoutRoot.ActualWidth / kinectDevice.DepthStream.FrameWidth);

point.Y = (int)(point.Y \* LayoutRoot.ActualHeight / kinectDevice.DepthStream.FrameHeight);

Point handPoint = new Point(point.X, point.Y);

Point dotDiff = new Point(dot.X - handPoint.X, dot.Y - handPoint.Y);

double length = Math.Sqrt(dotDiff.X \* dotDiff.X + dotDiff.Y \* dotDiff.Y);

int lastPoint = this.CrayonElement.Points.Count - 1;

//手势离点足够近

if (length < 25)

{

if (lastPoint > 0)

{

//移去最后一个点

this.CrayonElement.Points.RemoveAt(lastPoint);

}

//设置直线的终点

this.CrayonElement.Points.Add(new Point(dot.X, dot.Y));

//设置新的直线的起点

this.CrayonElement.Points.Add(new Point(dot.X, dot.Y));

//转到下一个点

this.puzzleDotIndex++;

if (this.puzzleDotIndex == this.puzzle.Dots.Count)

{

//通知游戏者游戏结束

}

}

else

{

if (lastPoint > 0)

{

//移除最后一个点，更新界面

Point lineEndpoint = this.CrayonElement.Points[lastPoint];

this.CrayonElement.Points.RemoveAt(lastPoint);

//将手势所在的点作为线的临时终点

lineEndpoint.X = handPoint.X;

lineEndpoint.Y = handPoint.Y;

this.CrayonElement.Points.Add(lineEndpoint);

}

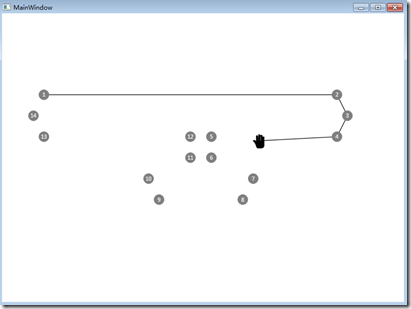
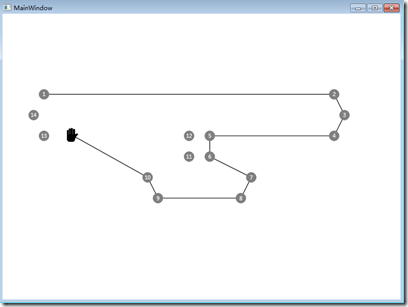
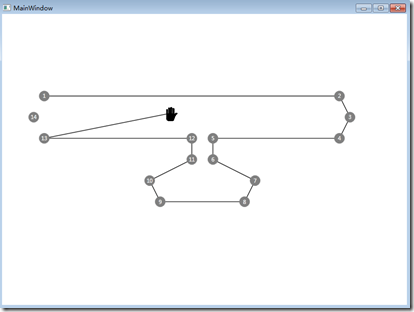
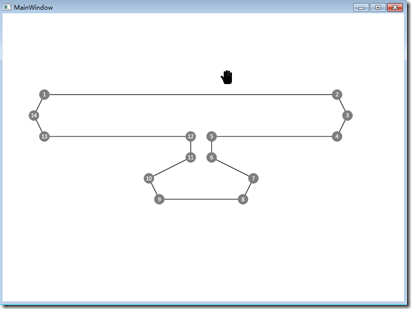
}

}

}

代码的大部分逻辑是如何将直线绘制到UI上面，另一部分逻辑是实现游戏的规则逻辑，比如点要按照从小到大的顺序连起来。程序计算当前鼠标手势点和下一个点之间的直线距离，如果距离小于25个像素宽度，那么认为手势移动到了这个点上。当然25可能有点绝对，但是对于这个小游戏，这应该是一个合适的值。因为Kinect返回的关节点信息可能有点误差而且用户的手可能会抖动，所以有效点击范围应该要比实际的UI元素大。这一点在Kinect或者其他触控设备上都是应该遵循的设计原则。如果用户移动到了这个点击区域，就可以认为用户点击到了这个目标点。

最后将TrackPuzzle方法添加到SkeletonFrameReady中就可以开始玩这个小游戏了。运行游戏，结果如下：

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204092309127769.png)[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204092309134671.png)[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204092309138226.png)[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204092309138750.png)

**1.5 进一步可改进地方**

在功能上，游戏已经完成了。游戏者可以开始游戏，移动手掌就可以玩游戏了。但是离完美的程序还很远。还需要进一步改进和完善。最主要的是要增加移动的平滑性。游戏过程中可以注意到手势有时候会跳跃。第二个主要问题是需要重新恢复游戏初始化状态。现在的程序，当游戏者完成游戏后只有结束应用程序才能开始新的游戏。

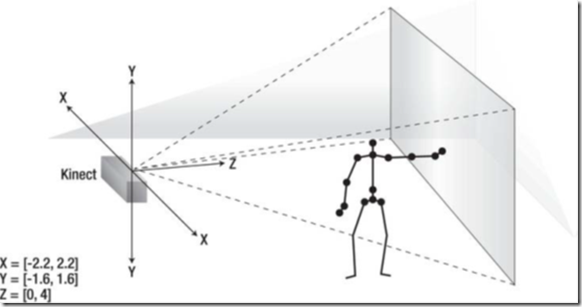
一个解决方式是，在左上角放一个重置按钮，当用户手进入到这个按钮上时，应用程序重置游戏，将puzzleDotIndex设置为0，清除CrayonElement对象中的所有子对象。最好的方式是，创建一个名为ResetPuzzle的新方法。

为了能够使得这个游戏有更好的体验，下面是可以进行改进的地方：

* 创建更多的游戏场景。当游戏加载是，可以从XML文件中读取一系列的数据，然后让随机的产生场景。
* 列出一系列游戏场景，可以让用户选择想玩那一个，可以创建一个菜单，让用户选择。
* 一旦用户完成了当前游戏，自动出现下一个游戏场景。
* 添加一些额外的数据，比如游戏名称，背景图片，提示信息等。例如，如果游戏结束，换一个背景图片，来个动画，或者音乐。
* 添加一些提示，比如可以提示用户下一个点时那一个。可以设置一个计时器，当用户找下一个点超过了某一个时间后，认为用户遇到了困难，可以进行有好的提示，例如可以用文字或者箭头表示下一个点的位置。如果用户找到了，则重置计时器。
* 如果用户离开游戏，应该重置游戏。比如用户可能需要接电话，喝茶或者其他的，可以设置一个定时器，当Kinect探测不到游戏者时可以开始计时，如果用户离开的时间超过了某一个限度，那么就重置游戏。
* 可以试着当用户找到一个下一个点时给一点有效的激励，比如说放一段小的音乐，或者给一个提示音等等。
* 当用户完成游戏后，可以出现一个绘图板，可供用户选择颜色，然后可以在屏幕上绘图。

**2. 各种坐标空间及变换**

在之前的各种例子中，我们处理和操作了关节点数据的位置。**在大多数情况下，原始的坐标数据是不能直接使用的。骨骼点数据和深度数据或者彩色影像数据的测量方法不同。**每一种类的数据(深度数据，影像数据，骨骼数据)都是在特定的集合坐标或空间内定义的。**深度数据或者影像数据用像素来表示，X,Y位置从左上角以0开始。深度数据的Z方位数据以毫米为单位。**与这些不同的是，**骨骼空间是以米为单位来描述的**，以深度传感器为中心，其X,Y值为0。骨骼坐空间坐标系是右手坐标系，X正方向朝右，Y周正方向朝上X轴数据范围为-2.2~2.2，总共范围为4.2米，Y周范围为-1.6~1.6米，Z轴范围为0~4米。下图描述了Skeleton数据流的空间坐标系。

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204092309137081.png)

**2.1 空间变换**

Kinect的应用程序就是用户和虚拟的空间进行交互。应用程序的交互越频繁。就越能增加应用的参与度和娱乐性。在上面的例子中，用户移动手来进行连线。我们知道用户需要将两个点连接起来，我们也需要知道用户的手是否在某一个点上。**这种判断只有通过将骨骼数据变换到UI空间上去才能确定**。由于SDK中骨骼数据并没有以一种可以直接在UI上绘图的方式提供，所以我们需要做一些变换。

将数据从骨骼数据空间转换到深度数据空间很容易。SDK提供了一系列方法来帮助我们进行这两个空间坐标系的转换。KinectSensor对象有一个称之为MapSkeletonPointToDepth的方法能够将骨骼点数据转换到UI空间中去。SDK中也提供了一个相反的MapDepthToSkeletonPoint方法。MapSkeletonPointToDepth方法接受一个SkeletonPoint点和一个DepthImageFormat作为参数。骨骼点数据来自Skeleton对象或者Joint对象的Position属性。方法的名字中有Depth，并不只是字面上的意思。目标空间并不需要Kinect深度影像。事实上，DepthStream不必初始化，方法通过DepthImageFormat来确定如何变化。一旦骨骼点数据被映射到深度空间中去了之后，他能够进行缩放到任意的纬度。

在之前绘制骨骼数据的例子中，GetJointPoint方法把每一个关节点数据转换LayoutRoot元素所在的到UI空间中，因为只有在UI空间中我们才能进行绘图。在上面的连线小游戏中，我们进行了两次这种转换。一个是在TrackHand方法中，在这个例子中，我们计算并将其转换到UI空间中，调整其位置，时期能够保证在点的中间。另一个地方是在TrackPuzzle方法中，使用用户的手势来绘制直线。这里只是简单的将数据从骨骼数据空间转换到UI空间。

**2.2 骨骼数据镜面对称**

细心地你可能会发现，骨骼数据是镜面对称的。在大多数情况下，应用是可行的，因为人对应于显示屏就应该是镜面对称。在上面的连线小游戏中，人对于与屏幕也应该是镜面对称，这样恰好模拟人的手势。但是在一些游戏中，角色代表实际的游戏者，可能角色是背对着游戏者的，这就是所谓的第三人称视角。在有些游戏中，这种镜像了的骨骼数据可能不好在UI上进行表现。一些应用或者游戏希望能够直面角色，不希望有这种镜像的效果。当游戏者挥动左手时也希望角色能够挥动左手。如果不修改代码直接绘制的话，在镜面效果下，角色会挥动右手，这显然不符合要求。

**不幸的是SDK中并没有一个选项或者属性能够进行设置来使得骨骼追踪引擎能够直接产生非镜像数据。**所以需要我们自己去编码进行这种转换，**幸运的是**，在了解了骨骼数据结构后进行转换比较简单。通过反转骨骼节点数据的X值就可以实现这个效果。要实现X值的反转，只需要将X的值乘以-1即可。我们可以对之前的那个绘制骨骼数据的例子中GetJointPoint方法进行一些调整，代码如下：

private Point GetJointPoint(Joint joint)

{

DepthImagePoint point = this.KinectDevice.MapSkeletonPointToDepth(joint.Position, this.KinectDevice.DepthStream.Format);

point.X \*= -1\*(int) this.LayoutRoot.ActualWidth / KinectDevice.DepthStream.FrameWidth;

point.Y \*= (int) this.LayoutRoot.ActualHeight / KinectDevice.DepthStream.FrameHeight;

return new Point(point.X, point.Y);

}

修改之后运行程序就会看到，当游戏者抬起左臂时，UI界面上的人物将会抬起右脚。

**3. SkeletonViewer自定义控件**

开发Kinect应用程序进行交互时，在开发阶段，将骨骼关节点数据绘制到UI界面上是非常有帮助的。在调试程序时骨骼数据影像能够帮助我们看到和理解原始输入数据，但是在发布程序时，我们不需要这些信息。一种办法是每一处都复制一份将骨骼数据绘制到UI界面上的代码，这显然不符合DIY原则，所以我们应当把这部分代码独立出来，***做成一个自定义控件***。

本节我们的目标是，将骨骼数据查看代码封装起来，并使其在调试时为我们提供更多的实时信息。我们使用自定义控件来实现这一功能点。首先，创建一个名为SkeletonViewer的自定义控件。这个控件可以是任何一个panel对象的一个子节点。创建一个自定义控件，并将其XAML替换成如下代码：

<UserControl x:Class="KinectDrawDotsGame.SkeletonViewer"

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

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

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

mc:Ignorable="d"

d:DesignHeight="300" d:DesignWidth="300">

<Grid x:Name="LayoutRoot">

<Grid x:Name="SkeletonsPanel"/>

<Canvas x:Name="JointInfoPanel"/>

</Grid>

</UserControl>

SkeletonsPanel就是绘制骨骼节点的panel。JointInfoPanel 是在调试时用来显示额外信息的图层。下一步是需要将一个KinectSnesor对象传递到这个自定义控件中来。为此，我们创建了一个DependencyProperty，使得我们可以使用数据绑定。下面的代码展示了这一属性。KinectDeviceChange静态方法对于任何使用该用户控件的方法和功能非常重要。该方法首先取消之前注册到KinectSensor的SkeletonFrameReady事件上的方法。如果不注销这些事件会导致内存泄漏。一个比较好的方法是采用[弱事件处理模式](http://msdn.microsoft.com/en-us/library/aa970850.aspx)(weak event handler pattern),这里不详细讨论。方法另一部分就是当KinectDevice属性部位空值时，注册SkeletonFrameReady事件。

protected const string KinectDevicePropertyName = "KinectDevice";

public static readonly DependencyProperty KinectDeviceProperty = DependencyProperty.Register(KinectDevicePropertyName, typeof(KinectSensor), typeof(SkeletonViewer), new PropertyMetadata(null, KinectDeviceChanged));

private static void KinectDeviceChanged(DependencyObject owner, DependencyPropertyChangedEventArgs e)

{

SkeletonViewer viewer = (SkeletonViewer)owner;

if (e.OldValue != null)

{

((KinectSensor)e.OldValue).SkeletonFrameReady -= viewer.KinectDevice\_SkeletonFrameReady;

viewer.\_FrameSkeletons = null;

}

if (e.NewValue != null)

{

viewer.KinectDevice = (KinectSensor)e.NewValue;

viewer.KinectDevice.SkeletonFrameReady += viewer.KinectDevice\_SkeletonFrameReady;

viewer.\_FrameSkeletons = new Skeleton[viewer.KinectDevice.SkeletonStream.FrameSkeletonArrayLength];

}

}

public KinectSensor KinectDevice

{

get { return (KinectSensor)GetValue(KinectDeviceProperty); }

set { SetValue(KinectDeviceProperty, value); }

}

现在用户控件能够接受来世KinectSensor对象的新的骨骼数据了。我们可以开始绘制这些骨骼数据。下面的代码展示了SkeletonFrameReady事件。大部分的代码和之前例子中的代码是一样的。一开始，判断用户控件的IsEnable控件是否被设置为true。这个属性可以使得应用程序可以方便的控制是否绘制骨骼数据。对于每一个骨骼数据，会调用两个方法，一个是DrawSkeleton方法，DrawSkeleton方法中有两个其他方法（CreateFigure和GetJointPoint）方法。另外一个方法是TrackJoint方法，这个方法显示节点的额外信息。TrackJoint方法在关节点所在的位置绘制圆圈，然后在圆圈上显示X,Y,X坐标信息。X,Y值是想对于用户控件的高度和宽度，以像素为单位。Z值是深度值。

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

{

SkeletonsPanel.Children.Clear();

JointInfoPanel.Children.Clear();

using (SkeletonFrame frame = e.OpenSkeletonFrame())

{

if (frame != null)

{

if (this.IsEnabled)

{

frame.CopySkeletonDataTo(this.\_FrameSkeletons);

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

{

DrawSkeleton(this.\_FrameSkeletons[i], this.\_SkeletonBrushes[i]);

**TrackJoint(this.\_FrameSkeletons[i].Joints[JointType.HandLeft], this.\_SkeletonBrushes[i]);**

**TrackJoint(this.\_FrameSkeletons[i].Joints[JointType.HandRight], this.\_SkeletonBrushes[i]);**

}

}

}

}

}

private void TrackJoint(Joint joint, Brush brush)

{

if (joint.TrackingState != JointTrackingState.NotTracked)

{

Canvas container = new Canvas();

Point jointPoint = GetJointPoint(joint);

double z = joint.Position.Z ;

Ellipse element = new Ellipse();

element.Height = 15;

element.Width = 15;

element.Fill = brush;

Canvas.SetLeft(element, 0 - (element.Width / 2));

Canvas.SetTop(element, 0 - (element.Height / 2));

container.Children.Add(element);

TextBlock positionText = new TextBlock();

positionText.Text = string.Format("<{0:0.00}, {1:0.00}, {2:0.00}>", jointPoint.X, jointPoint.Y, z);

positionText.Foreground = brush;

positionText.FontSize = 24;

positionText.Measure(new Size(double.PositiveInfinity, double.PositiveInfinity));

Canvas.SetLeft(positionText, 35);

Canvas.SetTop(positionText, 15);

container.Children.Add(positionText);

Canvas.SetLeft(container, jointPoint.X);

Canvas.SetTop(container, jointPoint.Y);

JointInfoPanel.Children.Add(container);

}

}

将这个自定义控件加到应用中很简单。由于是自定义控件，自需要在应用程序的XAML文件中声明自定义控件，然后在程序中给SkeletonViewer的KinectDevice赋值，主界面和后台逻辑代码更改部分如下加粗所示:

<Window x:Class="KinectDrawDotsGame.MainWindow"

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

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

**xmlns:c="clr-namespace:KinectDrawDotsGame"**

Title="MainWindow" Height="600" Width="800" Background="White">

<Viewbox>

<Grid x:Name="LayoutRoot" Width="1920" Height="1200">

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

<Polyline x:Name="CrayonElement" Stroke="Black" StrokeThickness="3" />

<Canvas x:Name="PuzzleBoardElement" />

<Canvas x:Name="GameBoardElement">

<Image x:Name="HandCursorElement" Source="Images/hand.png" Width="75"

Height="75" RenderTransformOrigin="0.5,0.5">

<Image.RenderTransform>

<TransformGroup>

<ScaleTransform x:Name="HandCursorScale" ScaleX="1" />

</TransformGroup>

</Image.RenderTransform>

</Image>

</Canvas>

</Grid>

</Viewbox>

</Window>

public KinectSensor KinectDevice

{

get { return this.kinectDevice; }

set

{

if (this.kinectDevice != value)

{

//Uninitialize

if (this.kinectDevice != null)

{

this.kinectDevice.Stop();

this.kinectDevice.SkeletonFrameReady -= KinectDevice\_SkeletonFrameReady;

this.kinectDevice.SkeletonStream.Disable();

**SkeletonViewerElement.KinectDevice = null;**

this.frameSkeletons = null;

}

this.kinectDevice = value;

//Initialize

if (this.kinectDevice != null)

{

if (this.kinectDevice.Status == KinectStatus.Connected)

{

this.kinectDevice.SkeletonStream.Enable();

this.frameSkeletons = new Skeleton[this.kinectDevice.SkeletonStream.FrameSkeletonArrayLength];

**SkeletonViewerElement.KinectDevice = this.KinectDevice;**

this.kinectDevice.Start();

this.KinectDevice.SkeletonFrameReady += KinectDevice\_SkeletonFrameReady;

}

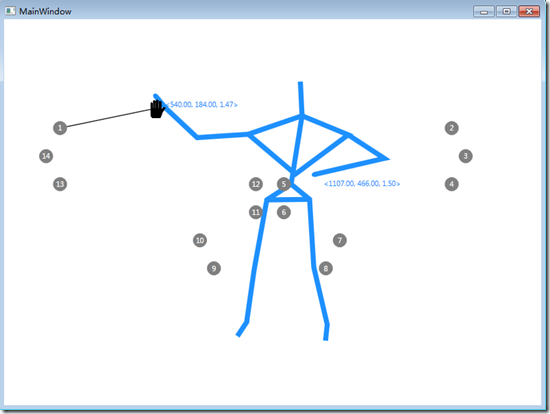
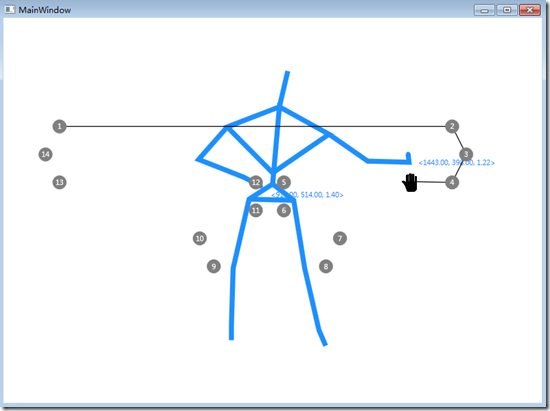
}

}

}

}

添加后，运行之前的程序，就可以看到如下界面：

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204092309141508.png)[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204092309145619.png)

**4. 结语**

本文通过介绍一个简单的Kinect连线游戏的开发来详细讨论如何骨骼追踪引擎来建立一个完整的Kinect应用，然后简要介绍了各个坐标控件以及转换，最后建立了一个显示骨骼信息的自定义控件，并演示了如何将自定义控件引入到应用程序中。下一篇文章将会结合另外一个小游戏来介绍WPF的相关知识以及骨骼追踪方面进一步值得注意和改进的地方。

本文代码[点击此处](http://files.cnblogs.com/yangecnu/KinectSDK_SkeletonTracking_part2_KinectDrawDotsGame.rar)下载，希望本文对您了解Kinect SDK有所帮助！