**[Kinect for Windows SDK开发入门(六)：骨骼追踪基础 上](http://www.cnblogs.com/yangecnu/archive/2012/04/06/KinectSDK_Skeleton_Tracking_Part1.html)**

Kinect产生的景深数据作用有限，要利用Kinect创建真正意义上交互，有趣和难忘的应用，还需要除了深度数据之外的其他数据。这就是骨骼追踪技术的初衷，骨骼追踪技术通过处理景深数据来建立人体各个关节的坐标,骨骼追踪能够确定人体的各个部分，如那部分是手，头部，以及身体。骨骼追踪产生X,Y,Z数据来确定这些骨骼点。在[上文中](http://www.cnblogs.com/yangecnu/archive/2012/04/05/Kinect-SDK_Depth_Image_Processing_Part2.html" \t "_blank)，我们讨论了景深图像处理的一些技术。**骨骼追踪系统采用的景深图像处理技术使用更复杂的算法如矩阵变换，机器学习及其他方式来确定骨骼点的坐标。**

本文首先用一个例子展示骨骼追踪系统涉及的主要对象，然后在此基础上详细讨论骨骼追踪中所涉及的**对象模型**。

**1. 获取骨骼数据**

本节将会创建一个应用来将获取到的骨骼数据绘制到UI界面上来。在开始编码前，首先来看看一些基本的对象以及如何从这些对象中如何获取骨骼数据。在进行数据处理之前了解数据的格式也很有必要。这个例子很简单明了，只需要骨骼数据对象然后将获取到的数据绘制出来。

彩色影像数据，景深数据分别来自ColorImageSteam和DepthImageStream，同样地，骨骼数据来自SkeletonStream。访问骨骼数据和访问彩色影像数据、景深数据一样，也有事件模式和 “拉”模式两种方式。在本例中我们采用基于事件的方式，因为这种方式简单，代码量少，并且是一种很普通基本的方法。KinectSensor对象有一个名为SkeletonFrameReady事件。当SkeletonStream中有新的骨骼数据产生时就会触发该事件。通过AllFramesReady事件也可以获取骨骼数据。在下一节中，我们将会详细讨论**骨骼追踪对象模型**，现在我们只展示如何从SkeletonStream流中获取骨骼数据。SkeletonStream产生的每一帧数据都是一个骨骼对象集合。每一个骨骼对象包含有描述骨骼位置以及骨骼关节的数据。每一个关节有一个唯一标示符如头(head)、肩(shoulder)、肘(dlbow)等信息和3D向量数据。

现在来写代码。首先创建一个新的wpf工程文件，添加Microsoft.Kinect.dll。添加基本查找和初始化传感器的代码，这些代码参考之前的文章。在开始启动传感器之前，初始化SkeletonStream数据流，并注册KinectSensor对象的SkeletonFrameReady事件，这个例子没有使用彩色摄像机和红外摄像机产生的数据，所以不需要初始化这些数据流。UI界面采用默认的，将Grid的名称改为LayoutRoot，之后就再Grid里面绘制。代码如下：

<Window x:Class="KinectSkeletonTracking.MainWindow"

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

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

Title="MainWindow" Height="350" Width="525">

<Grid x:Name="LayoutRoot" Background="White">

</Grid>

</Window>

后台逻辑代码如下：

private KinectSensor kinectDevice;

private readonly Brush[] skeletonBrushes;//绘图笔刷

private Skeleton[] frameSkeletons;

public MainWindow()

{

InitializeComponent();

skeletonBrushes = new Brush[] { Brushes.Black, Brushes.Crimson, Brushes.Indigo, Brushes.DodgerBlue, Brushes.Purple, Brushes.Pink };

KinectSensor.KinectSensors.StatusChanged += KinectSensors\_StatusChanged;

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

}

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();**

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];**

**this.kinectDevice.SkeletonFrameReady += KinectDevice\_SkeletonFrameReady;**

this.kinectDevice.Start();

}

}

}

}

}

private void KinectSensors\_StatusChanged(object sender, StatusChangedEventArgs e)

{

switch (e.Status)

{

case KinectStatus.Initializing:

case KinectStatus.Connected:

case KinectStatus.NotPowered:

case KinectStatus.NotReady:

case KinectStatus.DeviceNotGenuine:

this.KinectDevice = e.Sensor;

break;

case KinectStatus.Disconnected:

//TODO: Give the user feedback to plug-in a Kinect device.

this.KinectDevice = null;

break;

default:

//TODO: Show an error state

break;

}

}

以上代码中，值得注意的是frameSkeletons数组以及该数组如何在流初始化时进行内存分配的。Kinect能够追踪到的骨骼数量是一个常量。这使得我们在整个应用程序中能够一次性的为数组分配内存。为了方便，Kinect SDK在SkeletonStream对象中定义了一个能够追踪到的骨骼个数常量FrameSkeletonArrayLength，使用这个常量可以方便的对数组进行初始化。代码中也定义了一个笔刷数组，这些笔刷在绘制骨骼时对多个游戏者可以使用不同的颜色进行绘制。也可以将笔刷数组中的颜色设置为自己喜欢的颜色。

下面的代码展示了SkeletonFrameReady事件的响应方法，每一次事件被激发时，通过调用事件参数的OpenSkeletonFrame方法就能够获取当前的骨骼数据帧。剩余的代码遍历骨骼数据帧的Skeleton数组frameSkeletons，在UI界面通过关节点将骨骼连接起来，用一条直线代表一根骨骼。UI界面简单，将Grid元素作为根结点，并将其背景设置为白色。

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

{

using (SkeletonFrame frame = e.OpenSkeletonFrame())

{

if (frame != null)

{

Polyline figure;

Brush userBrush;

Skeleton skeleton;

LayoutRoot.Children.Clear();

frame.CopySkeletonDataTo(this.frameSkeletons);

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

{

skeleton = this.frameSkeletons[i];

if (skeleton.TrackingState == SkeletonTrackingState.Tracked)

{

userBrush = this.skeletonBrushes[i % this.skeletonBrushes.Length];

//绘制头和躯干

figure = CreateFigure(skeleton, userBrush, new[] { JointType.Head, JointType.ShoulderCenter, JointType.ShoulderLeft, JointType.Spine,

JointType.ShoulderRight, JointType.ShoulderCenter, JointType.HipCenter

});

LayoutRoot.Children.Add(figure);

figure = CreateFigure(skeleton, userBrush, new[] { JointType.HipLeft, JointType.HipRight });

LayoutRoot.Children.Add(figure);

//绘制作腿

figure = CreateFigure(skeleton, userBrush, new[] { JointType.HipCenter, JointType.HipLeft, JointType.KneeLeft, JointType.AnkleLeft, JointType.FootLeft });

LayoutRoot.Children.Add(figure);

//绘制右腿

figure = CreateFigure(skeleton, userBrush, new[] { JointType.HipCenter, JointType.HipRight, JointType.KneeRight, JointType.AnkleRight, JointType.FootRight });

LayoutRoot.Children.Add(figure);

//绘制左臂

figure = CreateFigure(skeleton, userBrush, new[] { JointType.ShoulderLeft, JointType.ElbowLeft, JointType.WristLeft, JointType.HandLeft });

LayoutRoot.Children.Add(figure);

//绘制右臂

figure = CreateFigure(skeleton, userBrush, new[] { JointType.ShoulderRight, JointType.ElbowRight, JointType.WristRight, JointType.HandRight });

LayoutRoot.Children.Add(figure);

}

}

}

}

}

循环遍历frameSkeletons对象，每一次处理一个骨骼，在处理之前需要判断是否是一个追踪好的骨骼，可以使用Skeleton对象的TrackingState属性来判断，只有骨骼追踪引擎追踪到的骨骼我们才进行绘制，忽略哪些不是游戏者的骨骼信息即过滤掉那些TrackingState不等于SkeletonTrackingState.Tracked的骨骼数据。Kinect能够探测到6个游戏者，但是同时只能够追踪到2个游戏者的骨骼关节位置信息。在后面我们将会详细讨论TrackingState这一属性。

处理骨骼数据相对简单，首先，我们根Kinect追踪到的游戏者的编号，选择一种颜色笔刷。然后利用这只笔刷绘制曲线。CreateFigure方法为每一根骨骼绘制一条直线。GetJointPoint方法在绘制骨骼曲线中很关键。**该方法以关节点的三维坐标作为参数，然后调用KinectSensor对象的MapSkeletonPointToDepth方法将骨骼坐标转换到深度影像坐标上去。**后面我们将会讨论为什么需要这样转换以及如何定义坐标系统。现在我们只需要知道的是，**骨骼坐标系和深度坐标及彩色影像坐标系不一样**，**甚至和UI界面上的坐标系不一样**。**在开发Kinect应用程序中，从一个坐标系转换到另外一个坐标系这样的操作非常常见**，**GetJointPoint方法的目的就是将骨骼关节点的三维坐标转换到UI绘图坐标系统，返回该骨骼关节点在UI上的位置**。下面的代码展示了CreateFigure和GetJointPoint这两个方法。

private Polyline CreateFigure(Skeleton skeleton, Brush brush, JointType[] joints)

{

Polyline figure = new Polyline();

figure.StrokeThickness = 8;

figure.Stroke = brush;

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

{

figure.Points.Add(GetJointPoint(skeleton.Joints[joints[i]]));

}

return figure;

}

private Point GetJointPoint(Joint joint)

{

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

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

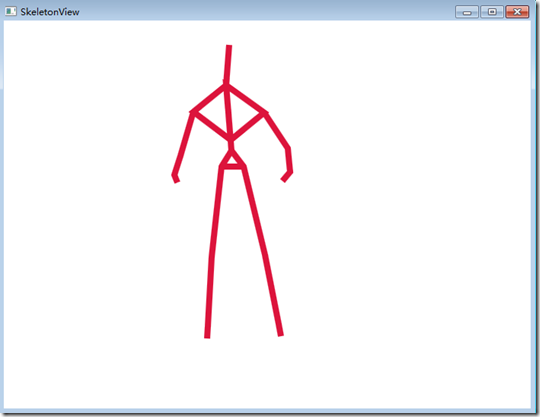
point.Y \*= (int)this.LayoutRoot.ActualHeight / KinectDevice.DepthStream.FrameHeight;

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

}

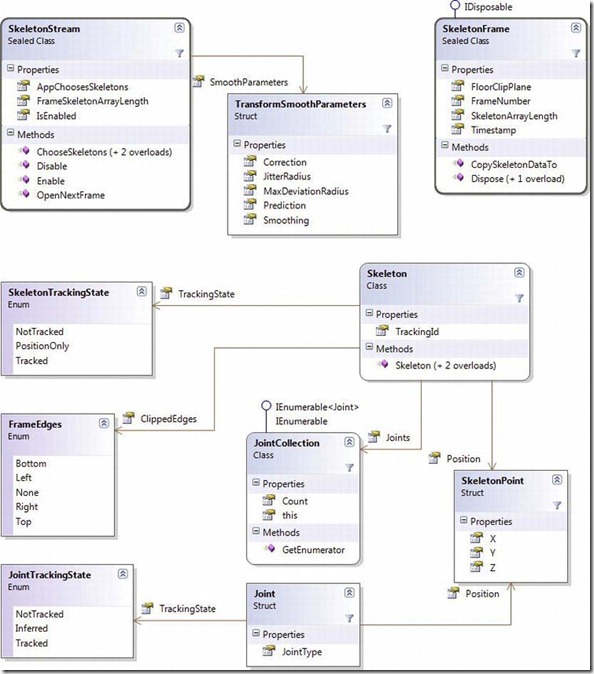
**值得注意的是，**骨骼关节点的三维坐标中我们舍弃了Z值，只用了X,Y值。**Kinect好不容易为我们提供了每一个节点的深度数据(Z值)**而我们却没有使用，这看起来显得很浪费。其实不是这样的，我们使用了节点的Z值，只是没有直接使用，没有在UI界面上展现出来而已。在坐标空间转换中是需要深度数据的。可以试试在GetJointPoint方法中，将joint的Position中的Z值改为0，然后再调用MapSkeletonPointToDepth方法，你会发现返回的对象中x和y值均为0，可以试试，将图像以Z值进行等比缩放，可以发现图像的大小是和Z值(深度)成反的。也就是说，深度值越小，图像越大，即人物离Kinect越近，骨骼数据越大。

运行程序，会得到如下骨骼图像，这个是手握键盘准备截图的姿势。一开始可能需要调整一些Form窗体的大小。程序会为每一个游戏者以一种颜色绘制骨骼图像，可以试着在Kinect前面移动，可以看到骨骼图像的变化，也可以走进然后走出图像以观察颜色的变化。仔细观察有时候可以看到绘图出现了一些奇怪的图案，在讨论完骨骼追踪相关的API之后，就会明白这些现象出现的原因了。

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204061859358404.png)

**2. 骨骼对象模型**

Kinect SDK中骨骼追踪有一些和其他对象不一样的对象结构和枚举。在SDK中骨骼追踪相关的内容几乎占据了三分之一的内容，可见Kinect中骨骼追踪技术的重要性。下图展示了骨骼追踪系统中涉及到的一些主要的**对象模型。有四个最主要的对象，他们是**SkeletonStream，SkeletonFrame，Skeleton和Joint。下面将详细介绍这四个对象。

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/20120406185937780.jpg)

**2.1 SkeletonStream对象**

SkeletonStream对象产生SkeletonFrame。从SkeletonStream获取骨骼帧数据和从ColorStream及DepthStream中获取数据类似。可以注册SkeletonFrameReady事件或者AllFramesReady事件通过事件模型来获取数据，或者是使用OpenNextFrame方法通过“拉”模型来获取数据。不能对同一个SkeletonStream同时使用这两种模式。如果注册了SkeletonFrameReady事件然后又调用OpenNextFrame方法将会返回一个InvalidOperationException异常。

**SkeletonStream的启动和关闭**

除非启动了SkeletonStream对象，否则，不会产生任何数据，默认情况下，SkeletonStream对象是关闭的。要使SkeletonStream产生数据，必须调用对象的Enabled方法。相反，调用Disable方法能够使SkeletonStream对象暂停产生数据。SkeletonStream有一个IsEnabled方法来描述当前SkeletonStream对象的状态。只有SkeletonStream对象启动了，KinectSensor对象的SkeletonFrameReady事件才能被激活。如果要使用“拉”模式来获取数据SkeletonStream也必须启动后才能调用OpenNextFrame方法。否则也会抛出InvalidOperationException异常。

一般地在应用程序的声明周期中，一旦启动了SkeletonStream对象，一般会保持启动状态。**但是在有些情况下，我们希望关闭SkeletonStream对象。比如在应用程序中使用多个Kinect传感器时。只有一个Kinect传感器能够产生骨骼数据，这也意味着，即使使用多个Kinect传感器，同时也只能追踪到两个游戏者的骨骼数据信息。在应用程序执行的过程中，有可能会关闭某一个Kinect传感器的SkeletonStream对象而开启另一个Kinect传感器的SkeletonStream对象。**

另一个有可能关闭骨骼数据产生的原因是出于性能方面的考虑，**骨骼数据处理是很耗费计算性能的操作**。打开骨骼追踪是可以观察的到CPU的占用率明显增加。当不需要骨骼数据时，关闭骨骼追踪很有必要。例如，在有些游戏场景中可能在展现一些动画效果或者播放视频，在这个动画效果或者视频播放时，停止骨骼追踪可能可以使得游戏更加流畅。

当然关闭SkeletonStream**也有一些副作用**。当SkeletonStream的状态发生改变时，所有的数据产生都会停止和从新开始。SkeletonStream的状态改变会使传感器重新初始化，将TimeStamp和FrameNumber重置为0。在传感器重新初始化时也有几毫秒的延迟。

**平滑化**

**在前面的例子中，会注意到，骨骼运动会呈现出跳跃式的变化。**有几个原因会导致出现这一问题，可能是应用程序的性能，游戏者的动作不够连贯，也有可能是Kinect硬件的性能问题。骨骼关节点的相对位置可能在帧与帧之间变动很大，这回对应用程序产生一些负面的影像。除了会影像用户体验和不愉快意外，也可能会导致用户的形象或者手的颤动抽搐而使用户感到迷惑。

SkeletonStream**对象有一种方法能够解决这个问题**。**他通过将骨骼关节点的坐标标准化来减少帧与帧之间的关节点位置差异**。当初始化SkeletonStream对象调用重载的Enable方法时可以传入一个TransformSmoothParameters参数。SkeletonStream对象有两个与平滑有关只读属性：IsSmoothingEnabled和SmoothParameters。当调用Enable方法传入了TransformSmoothParameters是IsSmoothingEnabled返回true而当使用默认的不带参数的Enable方法初始化时，IsSmoothingEnabled对象返回false。SmoothParameters属性用来存储定义平滑参数。TransformSmoothParameters这个结构定义了一些属性：

* 修正值(Correction)属性，接受一个从0-1的浮点型。值越小，修正越多。
* 抖动半径(JitterRadius)属性，设置修正的半径，如果关节点“抖动”超过了设置的这个半径，将会被纠正到这个半径之内。该属性为浮点型，单位为米。
* 最大偏离半径(MaxDeviationRadius)属性，用来和抖动半径一起来设置抖动半径的最大边界。任何超过这一半径的点都不会认为是抖动产生的，而被认定为是一个新的点。该属性为浮点型，单位为米。
* 预测帧大小(Prediction)属性，返回用来进行平滑需要的骨骼帧的数目。
* 平滑值(Smoothing)属性，设置处理骨骼数据帧时的平滑量，接受一个0-1的浮点值，值越大，平滑的越多。0表示不进行平滑。

**对骨骼关节点进行平滑处理会产生性能开销。平滑处理的越多，性能消耗越大。**设置平滑参数没有经验可以遵循。需要不断的测试和调试已达到最好的性能和效果。在程序运行的不同阶段，可能需要设置不同的平滑参数。

Note：**SDK使用[霍尔特指数平滑](http://forecasters.org/pdfs/foresight/free/Issue19_goodwin.pdf)(Holt Double Exponential Smoothing)来对减少关节点的抖动。**[指数平滑](http://en.wikipedia.org/wiki/Exponential_smoothing)数据处理与时间有关。骨骼数据是时间序列数据，因为骨骼引擎会以某一时间间隔不断产生一帧一帧的骨骼数据。平滑处理使用统计方法进行滑动平均，这样能够减少时间序列数据中的噪声和极值。类似的处理方法最开始被用于金融市场和经济数据的预测。

**骨骼追踪对象选择**

默认情况下，骨骼追踪引擎会对视野内的所有活动的游戏者进行追踪。但只会选择两个可能的游戏者产生骨骼数据，大多数情况下，这个选择过程不确定。如果要自己选择追踪对象，**需要使用AppChoosesSkeletons属性和ChooseSkeletons方法**。 默认情况下AppChoosesSkeleton属性为false，骨骼追踪引擎追踪所有可能的最多两个游戏者。要手动选择追踪者，需要将AppChoosesSkeleton设置为true，并调用ChooseSkeletons方法，传入TrackingIDs已表明需要追踪那个对象。ChooseSkeletons方法接受一个，两个或者0个TrackingIDs。当ChooseSkeletons方法传入0个参数时，引擎停止追踪骨骼信息。有一些需要注意的地方：

* 如果调用ChooseSkeletons方法时AppChoosesSkeletons的属性为false，就会引发InvalidOperationExcepthion的异常。
* 如果在SkeletonStream开启前，经AppChoosesSkeletons设置为true，只有手动调用ChooseSkeleton方法后才会开始骨骼追踪。
* 在AppChoosesSkeletons设置为 true之前，骨骼引擎自动选择追踪的游戏者，并且继续保持这些该游戏者的追踪，直到用户手动指定需要追踪的游戏者。如果自动选择追踪的游戏者离开场景，骨骼引擎不会自动更换追踪者。
* 将AppChoosesSkeletons冲新设置为false后，骨骼引擎会继续对之前手动设置的游戏者进行追踪，直到这些游戏者离开视野。当游戏这离开视野时骨骼引擎才会选择其他的可能的游戏者进行追踪。

**2.2 SkeletonFrame**

SkeletonStream产生SkeletonFrame对象。可以使用事件模型从事件参数中调用OpenSkeletonFrame方法来获取SkeletonFrame对象，或者采用”拉”模型调用SkeletonStream的OpenNextFrame来获取SkeletonFrame对象。SkeletonFrame对象会存储骨骼数据一段时间。同以通过调用SkeletonFrame对象的CopySkeletonDataTo方法将其保存的数据拷贝到骨骼对象数组中。SkeletonFrame对象有一个SkeletonArrayLength的属性，这个属性表示追踪到的骨骼信息的个数。

**时间标记字段**

SkeletonFrame的FrameNumber和Timestamp字段表示当前记录中的帧序列信息。FrameNumber是景深数据帧中的用来产生骨骼数据帧的帧编号。帧编号通常是不连续的，但是之后的帧编号一定比之前的要大。**骨骼追踪引擎**在追踪过程中可能会忽略某一帧深度数据，这跟应用程序的性能和每秒产生的帧数有关。例如，在基于事件获取骨骼帧信息中，如果事件中处理帧数据的时间过长就会导致这一帧数据还没有处理完就产生了新的数据，那么这些新的数据就有可能被忽略了。如果采用“拉”模型获取帧数据，那么取决于应用程序设置的骨骼引擎产生数据的频率，即取决于深度影像数据产生骨骼数据的频率。

Timestap字段记录字Kinect传感器初始化以来经过的累计毫秒时间。不用担心FrameNumber或者Timestamp字段会超出上限。FrameNumber是一个32位的整型，Timestamp是64位整型。如果应用程序以每秒30帧的速度产生数据，应用程序需要运行2.25年才会达到FrameNumber的限，此时Timestamp离上限还很远。另外在Kinect传感器每一次初始化时，这两个字段都会初始化为0。可以认为FrameNumber和Timestamp这两个值是唯一的。

这两个字段在分析处理帧序列数据时很重要，比如进行关节点值的平滑，手势识别操作等。在多数情况下，我们通常会处理帧时间序列数据，这两个字段就显得很有用。目前SDK中并没有包含手势识别引擎。**在未来SDK中加入手势引擎之前，我们需要自己编写算法来对帧时间序列进行处理来识别手势，这样就会大量依赖这两个字段。**

**帧描述信息**

FloorClipPlane字段是一个有四个元素的元组Tuple<int,int,int,int>,每一个都是Ax+By+Cz+**D**=0地面平面(floor plane)表达式里面的系数项。元组中第一个元素表示A，即x前面的系数，一次类推，**最后一个表示常数项，通常为负数，是Kinect距离地面高度。**在可能的情况下SDK会利用图像处理技术来确定这些系数。但是有时候这些系数不肯能能够确定下来，可能需要预估。当地面不能确定时FloorClipPlane中的所有元素均为0.

**2.3 Skeleton**

Skeleton类定义了一系列字段来描述骨骼信息，**包括描述骨骼的位置以及骨骼中关节可能的位置信息**。骨骼数据可以通过调用SkeletonFrame对象的CopySkeletonDataTo方法获得Skeleton数组。CopySkeletonDataTo方法有一些不可预料的行为，可能会影响内存使用和其引用的骨骼数组对象。产生的每一个骨骼数组对象数组都是唯一的。以下面代码为例：

Skeleton[] skeletonA = new Skeleton[frame.SkeletonArrayLength];

Skeleton[] skeletonB = new Skeleton[frame.SkeletonArrayLength];

frame.CopySkeletonDataTo(skeletonA);

frame.CopySkeletonDataTo(skeletonB);

Boolean resultA = skeletonA[0] == skeletonB[0];//false

Boolean resultB = skeletonA[0].TrackingId == skeletonB[0].TrackingId;//true

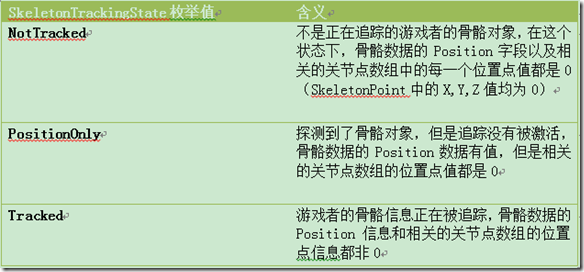
上面的代码可以看出，使用CopySkeletonDataTo是深拷贝对象，会产生两个不同的Skeleton数组对象。

**TrackingID**

**骨骼追踪引擎对于每一个追踪到的游戏者的骨骼信息都有一个唯一编号。**这个值是整型，他会随着新的追踪到的游戏者的产生添加增长。和之前帧序号一样，这个值并不是连续增长的，但是能保证的是后面追踪到的对象的编号要比之前的编号大。另外，这个编号的产生是不确定的。如果骨骼追踪引擎失去了对游戏者的追踪，比如说游戏者离开了Kinect的视野，那么这个对应的唯一编号就会过期。当Kinect追踪到了一个新的游戏者，他会为其分配一个新的唯一编号，编号值为0表示这个骨骼信息不是游戏者的，他在集合中仅仅是一个占位符。应用程序使用TrackingID来指定需要骨骼追踪引擎追踪那个游戏者。调用SkeletonStream对象的ChooseSkeleton能以初始化对指定游戏这的追踪。

**TrackingState**

该字段表示当前的骨骼数据的状态。下表展示了SkeletonTrackingState枚举的可能值机器含义：

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204061859383505.png)

**Position**

Position一个SkeletonPoint类型的字段，代表所有骨骼的中间点。身体的中间点和脊柱关节的位置相当。改字段提供了一个最快且最简单的所有视野范围内的游戏者位置的信息，而不管其是否在追踪状态中。在一些应用中，如果不用关心骨骼中具体的关节点的位置信息，那么该字段对于确定游戏者的位置状态已经足够。该字段对于手动选择要追踪的游戏者(SkeletonStream.ChooseSkeleton)也是一个参考。例如，应用程序可能需要追踪距离Kinect最近的且处于追踪状态的游戏者，那么该字段就可以用来过滤掉其他的游戏者。

**ClippedEdges**

ClippedEdges字段用来描述追踪者的身体哪部分位于Kinect的视野范围外。他大体上提供了一个追踪这的位置信息。使用这一属性可以通过程序调整Kinect摄像头的俯仰角或者提示游戏者让其返回到视野中来。该字段类型为FrameEdges，他是一个枚举并且有一个FlagsAtrribute自定义属性修饰。这意味着ClippedEdges字段可以一个或者多个FrameEdges值。下面列出了FrameEdges的所有可能的值。

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204061859385914.png)

当游戏者身体的某一部分超出Kinect视场范围时，就需要对骨骼追踪产生的数据进行某些改进，因为某些部位的数据可能追踪不到或者不准确。最简单的解决办法就是提示游戏者身体超出了Kinect的某一边界范围让游戏者回到视场中来。例如，有时候应用程序可能不关心游戏者超出Kinect视场下边界的情况，但是如果超出了左边界或者右边界时就会对应用产生影响，这是可以针对性的给游戏者一些提示。另一个解决办法是调整Kinect设备的物理位置。Kinect底座上面有一个小的马达能够调整Kinect的俯仰角度。俯仰角度可以通过更改KinectSensor对象的ElevationAnagle属性来进行调整。如果应用程序对于游戏者脚部动作比较关注，那么通过程序调整Kinect的俯仰角能够决绝脚部超出视场下界的情况。

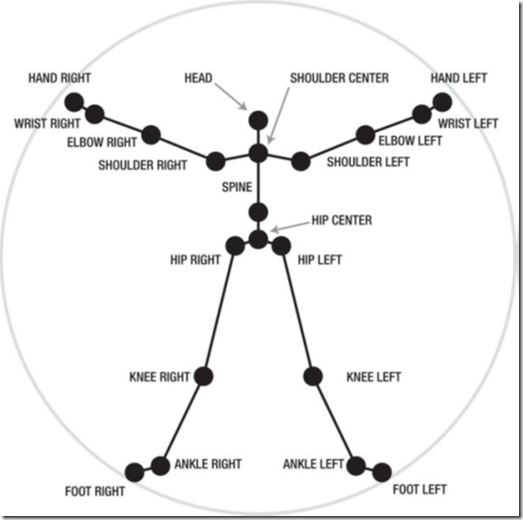
ElevationAnagle以度为单位。KinectSensor的MaxElevationAngle和MinElevationAngle确定了可以调整角度的上下界。任何将ElevationAngle设置超出上下界的操作将会掏出ArgumentOutOfRangeExcepthion异常。微软建议不要过于频繁重复的调整俯仰角以免损坏马达。为了使得开发这少犯错误和保护马达，SDK限制了每秒能调整的俯仰角的值。SDK限制了在连续15次调整之后要暂停20秒。

**Joints**

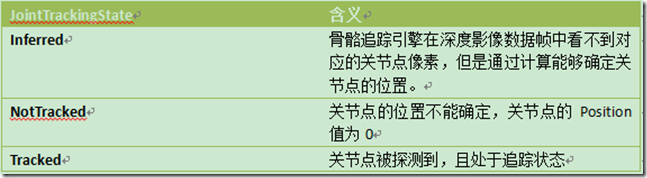
每一个骨骼对象都有一个Joints字段。该字段是一个JointsCollection类型，它存储了一些列的Joint结构来描述骨骼中可追踪的关节点(如head,hands,elbow等等)。应用程序使用JointsCollection索引获取特定的关节点，并通过节点的JointType枚举来过滤指定的关节点。即使Kinect视场中没有游戏者Joints对象也被填充。

**2.4 Joint**

骨骼追踪引擎能够跟踪和获取每个用户的近20个点或者关节点信息。**追踪的数据以关节点数据展现**，它有三个属性。JointType属性是一个枚举类型。下图描述了可追踪的所有关节点。

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204061859406862.png)

每一个关节点都有类型为SkeletonPoint的Position属性，他通过X,Y,Z三个值来描述关节点的控件位置。X,Y值是相对于骨骼平面空间的位置，他和深度影像，彩色影像的空间坐标系不一样。KinectSnesor对象有一些列的坐标转换方法，可以将骨骼坐标点转换到对应的深度数据影像中去。最后每一个Skeleton对象还有一个JointTrackingState属性，他描述了该关节点的跟踪状态及方式，下面列出了所有的可能值。

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204061859415683.png)

**3. 结语**

本文首先通过一个例子展示骨骼追踪系统所涉及的主要对象，并将骨骼数据在UI界面上进行了绘制，在此基础上详细介绍了骨骼追踪对象模型中涉及到的主要对象，方法和属性。SDK中骨骼追踪占了大概三分之一的内容，所以熟悉这些对象对于开发基于Kinect应用程序至关重要。限于篇幅，下一篇文章将会演示一个使用Kinect骨骼追踪系统开发的小游戏，然后讨论控件坐标变换，敬请期待。