**[Kinect for Windows SDK开发入门(五)：景深数据处理 下](http://www.cnblogs.com/yangecnu/archive/2012/04/05/KinectSDK_Depth_Image_Processing_Part2.html)**

**1. 简单的景深影像处理**

在上篇文章中，我们讨论了如何获取像素点的深度值以及如何根据深度值产生影像。在之前的例子中，我们过滤掉了阈值之外的点。这就是一种简单的图像处理，叫阈值处理。使用的阈值方法虽然有点粗糙，但是有用。更好的方法是利用机器学习来从每一帧影像数据中计算出阈值。Kinect深度值最大为4096mm，0值通常表示深度值不能确定，一般应该将0值过滤掉。微软建议在开发中使用1220mm（4’）~3810mm(12.5’)范围内的值。在进行其他深度图像处理之前，应该使用阈值方法过滤深度数据至1220mm-3810mm这一范围内。

使用统计方法来处理深度影像数据是一个很常用的方法。阈值可以基于深度数据的平均值或者中值来确定。统计方法可以帮助确定某一点是否是噪声、阴影或者是其他比较有意义的物体，比如说用户的手的一部分。有时候如果不考虑像素的视觉意义，可以对原始深度进行数据挖掘。对景深数据处理的目的是进行形状或者物体的识别。通过这些信息，程序可以确定人体相对于Kinect的位置及动作。

**1.1深度影像数据直方图**

直方图是统计数据分布的一个很有效的工具。在这里我们关心的是一个景深影像图中深度值的分布。直方图能够直观地反映给定数据集中数据的分布状况。从直方图中，我们能够看出深度值出现的频率以及聚集分组。通过这些信息，我们能够确定阈值以及其他能够用来对图像进行过滤的指标，使得能够最大化的揭示深度影像图中的深度信息。为了展示这一点，接下来我们将会展示一副景深影像数据的直方图，并通过直方图，使用一些简单的技术来过滤掉我们不想要的像素点。

首先创建一个新的项目。然后根据之前文章中讲的步骤发现和初始化KinectSensor对象来进行深度影像数据处理，包括注册DepthFrameReady事件。在添加实现深度直方图之前，将UI界面更改为如下：

<Window x:Class="KinectDepthHistogram.MainWindow"

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

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

Title="MainWindow" Height="800" Width="1200" WindowStartupLocation="CenterScreen">

<Grid>

<StackPanel>

<StackPanel Orientation="Horizontal">

<Image x:Name="DepthImage" Width="640" Height="480" />

<Image x:Name="FilteredDepthImage" Width="640" Height="480" />

</StackPanel>

<ScrollViewer Margin="0,15" HorizontalScrollBarVisibility="Auto" VerticalScrollBarVisibility="Auto">

<StackPanel x:Name="DepthHistogram" Orientation="Horizontal" Height="300" />

</ScrollViewer>

</StackPanel>

</Grid>

</Window>

创建直方图的方法很简单，就是创建一系列的矩形元素，然后将它添加到名为DepthHistogram的StackPanel元素中，由于DepthHistogram对象的Orientation属性设置为Horizontal,所以这些矩形会水平排列。大多数应用程序计算直方图只是用来进行中间过程处理用，如果想要将直方图展现出来，则需要在绘图上面做些工作。下面的代码展现了如何绘制直方图：

private void KinectDevice\_DepthFrameReady(object sender, DepthImageFrameReadyEventArgs e)

{

using (DepthImageFrame frame = e.OpenDepthImageFrame())

{

if (frame != null)

{

frame.CopyPixelDataTo(this.\_DepthPixelData);

CreateBetterShadesOfGray(frame, this.\_DepthPixelData);

CreateDepthHistogram(frame, this.\_DepthPixelData);

}

}

}

private void CreateDepthHistogram(DepthImageFrame depthFrame, short[] pixelData)

{

int depth;

int[] depths = new int[4096];

double chartBarWidth = Math.Max(3, DepthHistogram.ActualWidth / depths.Length);

int maxValue = 0;

DepthHistogram.Children.Clear();

//计算并获取深度值.并统计每一个深度值出现的次数

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

{

depth = pixelData[i] >> DepthImageFrame.PlayerIndexBitmaskWidth;

if (depth >= LoDepthThreshold && depth <= HiDepthThreshold)

{

depths[depth]++;

}

}

//查找最大的深度值

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

{

maxValue = Math.Max(maxValue, depths[i]);

}

//绘制直方图

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

{

if (depths[i] > 0)

{

Rectangle r = new Rectangle();

r.Fill = Brushes.Black;

r.Width = chartBarWidth;

r.Height = DepthHistogram.ActualHeight \* (depths[i] / (double)maxValue);

r.Margin = new Thickness(1, 0, 1, 0);

r.VerticalAlignment = System.Windows.VerticalAlignment.Bottom;

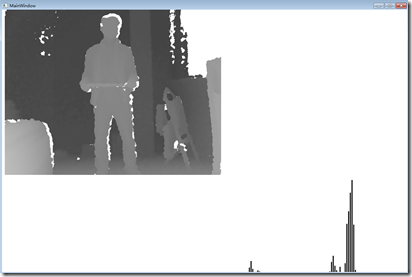
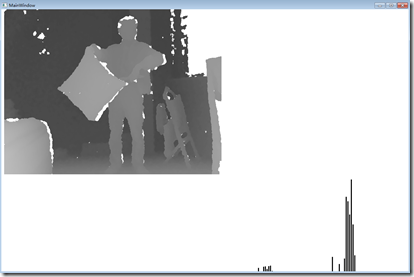
DepthHistogram.Children.Add(r);

}

}

}

绘制直方图时，创建一个数组来存储所有可能的深度值数据，因此数组的大小为4096。第一步遍历深度图像，获取深度值，然后统计深度值出现的次数。因为设置了最高最低的距离阈值，忽略了0值。下图显示了深度值影像的直方图，X轴表示深度值，Y轴表示深度值在图像中出现的次数。

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204050120424547.png)[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204050120508116.png)

当站在Kinect前后晃动时，下面的直方图会不停的变化。图中后面最长的几个线条表示墙壁，大约离摄像头3米左右，前面的几个小的线条是人体，大概离摄像头2米左右，下面那副图中，我手上拿了一个靠垫，可以发现直方图与之前的直方图相比发生了一些变化。

这两幅图中，可以看到直方图都集中在两个地方，前面的一小撮和后面的那一大坨。所以根据直方图可以看出，前面那个表示人体，后面那个代表房间的墙壁，在结合一些图像处理技术，就大致可以把人体和背景区分开来了。

**1.2 一些图像处理相关的知识**

本文不打算详细讲解图像处理的相关知识。只是讨论如何获取原始的深度数据，以及理解数据的用途。很多情况下，基于Kinect的应用程序不会对深度数据进行很多处理。如果要处理数据，也应该使用一些类库诸如OpenCV库来处理这些数据。深度影像处理经常要耗费大量计算资源，不应该使用诸如C#这类的高级语言来进行影像处理。

Note： [OpenCV](http://opencv.willowgarage.com/wiki/)(Open Source Computer Vision)库是是一个经常用来处理和计算影像数据的算法类库。这个类库也包含点云库(Point Cloud Library, PCL) 和机器人操作系统(Robot Operating System, ROS)，这些都涉及到了大量的深度数据处理。有兴趣的可以研究一下OpenCV库。

应用程序处理深度数据目的是用来确定人体在Kinect视场中的位置。虽然Kinect SDK中的骨骼追踪在这方面功能更强大，但是在某些情况下还是需要从深度数据中分析出人物所处的位置。在下节中，我们将会分析人体在深度影像中的范围。在开始之前，有必要了解和研究一下图像处理中常用的一些算法，有时候这些对特征提取非常有帮助。

* 图像处理

[阈值处理](http://en.wikipedia.org/wiki/Thresholding_(image_processing))(Thresholding)

[图像分割](http://en.wikipedia.org/wiki/Segmentation_(image_processing)) (Segmentation)

* [边缘/轮廓探测](http://en.wikipedia.org/wiki/Feature_detection_(computer_vision)) (Edge/Contour Detection)

[高斯滤波](http://en.wikipedia.org/wiki/Gaussian_filter)(Gaussian filter)

[Sobel](http://en.wikipedia.org/wiki/Sobel_operator)、[Prewitt](http://en.wikipedia.org/wiki/Prewitt)、Kirsh算子

[Canny](http://en.wikipedia.org/wiki/Canny_edge_detector)算子

[罗伯特](http://en.wikipedia.org/wiki/Roberts_Cross) 算子

* [哈夫变换](http://en.wikipedia.org/wiki/Hough_transform)
* [Blob检测](http://en.wikipedia.org/wiki/Blob_detection)
* [拉普拉斯变换](http://en.wikipedia.org/wiki/Laplacian_of_Gaussian#The_Laplacian_of_Gaussian)
* [Hession](http://en.wikipedia.org/wiki/Hessian_matrix) 算子
* [K均值聚类](http://en.wikipedia.org/wiki/K-means_clustering)

**2. 深度数据和游戏者索引位**

Kinect SDK具有分析景深数据和探测人体或者游戏者轮廓的功能，它一次能够识别多达6个游戏者。SDK为每一个追踪到的游戏者编号作为索引。游戏者索引存储在深度数据的前3个位中。如[前一篇文章](http://www.cnblogs.com/yangecnu/archive/2012/04/04/2431615.html)讨论的，景深数据每一个像素占16位，0-2位存储游戏者索引值，3-15为存储深度值。7 (0000 0111)这个位掩码能够帮助我们从深度数据中获取到游戏者索引值。幸运的是，SDK为游戏者索引位定义了一些列常量。他们是DepthImageFrame.PlayerIndexBitmaskWidth和DepthImageFrame.PlayerIndexBitmask。前一个值是3，后一个是7。开发者应该使用SDK定义的常量而不应该硬编码3或者7。

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204050120522618.png)

游戏者索引位取值范围为0~6，值为0表示该像素不是游戏者。但是初始化了景深数据流并没有开启游戏者追踪。游戏者追踪需要依赖骨骼追踪技术。初始化KinectSensor对象和DepthImageStream对象时，需要同时初始化SkeletonStream对象。只有当SkeletonStream对象初始化了后，景深数据中才会有游戏者索引信息。获取游戏者索引信息并不需要注册SkeletonFrameReady事件。

再创建一个工程来展示如何获取游戏者索引位信息。首先，创建一个新的项目，初始化KinectSensor对象，初始化DepthImageStream和SkeletonStream对象，并注册KinectSensor的DepthFrameReady事件。在UI界面MainWindows.xaml中添加两个Image控件分别取名为RamDepthImage和EnhDepthImage。添加WirteableBitmap对象，代码如下:

<Window x:Class="KinectDepthImagePlayerIndex.MainWindow"

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

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

Title="Kinect Player Index" Height="600" Width="1200">

<Grid>

<StackPanel Orientation="Horizontal">

<Image x:Name="RawDepthImage" Margin="0,0,10,0" Width="640" Height="480"/>

<Image x:Name="EnhDepthImage" Width="640" Height="480"/>

</StackPanel>

</Grid>

</Window>

下面的代码将有游戏者索引位的数据显示为黑色，其他像元显示为白色。

private void KinectDevice\_DepthFrameReady(object sender, DepthImageFrameReadyEventArgs e)

{

using (DepthImageFrame frame = e.OpenDepthImageFrame())

{

if (frame != null)

{

frame.CopyPixelDataTo(this.\_RawDepthPixelData);

this.\_RawDepthImage.WritePixels(this.\_RawDepthImageRect, this.\_RawDepthPixelData, this.\_RawDepthImageStride, 0);

CreatePlayerDepthImage(frame, this.\_RawDepthPixelData);

}

}

}

private void CreatePlayerDepthImage(DepthImageFrame depthFrame, short[] pixelData)

{

int playerIndex;

int depthBytePerPixel = 4;

byte[] enhPixelData = new byte[depthFrame.Width \* depthFrame.Height \* depthBytePerPixel];

for (int i = 0, j = 0; i < pixelData.Length; i++, j += depthBytePerPixel)

{

playerIndex = pixelData[i] & DepthImageFrame.PlayerIndexBitmask;

if (playerIndex == 0)

{

enhPixelData[j] = 0xFF;

enhPixelData[j + 1] = 0xFF;

enhPixelData[j + 2] = 0xFF;

}

else

{

enhPixelData[j] = 0x00;

enhPixelData[j + 1] = 0x00;

enhPixelData[j + 2] = 0x00;

}

}

this.\_EnhDepthImage.WritePixels(this.\_EnhDepthImageRect, enhPixelData, this.\_EnhDepthImageStride, 0);

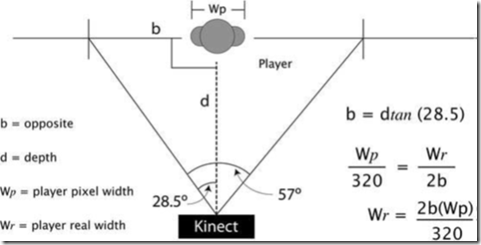
}

运行后，效果如下图，还可以对上面的代码进行一些改进。例如，可以对游戏者所在的像素进行灰度值拉伸，能够绘制出游戏者深度值的直方图，根据直方图可以看出每一灰度级对应的频率。另一个改进是，可以对不同的游戏者给予不同的颜色显示，比如游戏者1用红色表示，游戏者2用蓝色表示等等。[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/20120405012055619.png)

要注意的是，不要对特定的游戏者索引位进行编码，因为他们是会变化的。实际的游戏者索引位并不总是和Kinect前面的游戏者编号一致。例如， Kinect视野中只有一个游戏者，但是返回的游戏者索引位值可能是3或者4。有时候第一个游戏者的游戏者索引位可能不是1，比如走进Kinect视野，返回的索引位是1，走出去后再次走进，可能索引位变为其他值了。所以开发Kinect应用程序的时候应该注意到这一点。

**3. 对物体进行测量**

像上篇文章中对深度值测量原理进行讨论的那样，像素点的X，Y位置和实际的宽度和高度并不一致。但是运用几何知识，通过他们对物体进行测量是可能的。每一个摄像机都有视场，焦距的长度和相机传感器的大小决定了视场角。Kinect中相机的水平和垂直视场角分别为57°和43°。既然我们知道了深度值，利用三角几何知识，就可以计算出物体的实际宽度。示意图如下：

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204050121006253.png)

图中的公式在某些情况下可能不准确，Kinect返回的数据也有这个问题。这个简化的公式并没有考虑到游戏者的其他部分。尽管如此，公式依然能满足大部分的应用。这里只是简单地介绍了如何将Kinect数据映射到真实环境中。如果想得到更好的精度，则需要研究Kinect摄像头的焦距和摄像头的尺寸。

在开始写代码前，先看看上图中的公式。摄像头的视场角是一个以人体深度位置为底的一个等腰三角形。人体的实际深度值是这个等腰三角形的高。可以将这个等腰三角形以人所在的位置分为两个直角三角形，这样就可以计算出底边的长度。一旦知道了底边的长度，我们就可以将像素的宽度转换为现实中的宽度。例如：如果我们计算出等腰三角形底边的宽度为1500mm，游戏者所占有的总象元的宽度为100，深度影像数据的总象元宽度为320。那么游戏者实际的宽度为468.75mm((1500/320)\*100)。公式中，我们需要知道游戏者的深度值和游戏者占用的总的象元宽度。我们可以将游戏者所在的象元的深度值取平均值作为游戏者的深度值。之所以求平均值是因为人体不是平的，这能够简化计算。计算人物高度也是类似的原理，只不过使用的垂直视场角和深度影像的高度。

知道了原理之后，就可以开始动手写代码实现了。先创建一个新的项目然后编写发现和初始化KinectSensor的代码，将DepthStream和SkeletonStream均初始化，然后注册KinectSnsor的DepthFrameReady事件。主UI界面中的代码如下：

<Window x:Class="KinectTakingMeasure.MainWindow"

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

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

Title="MainWindow" Height="800" Width="1200" WindowStartupLocation="CenterScreen">

<Grid>

<StackPanel Orientation="Horizontal">

<Image x:Name="DepthImage" />

<ItemsControl x:Name="PlayerDepthData" Width="300" TextElement.FontSize="20">

<ItemsControl.ItemTemplate>

<DataTemplate>

<StackPanel Margin="0,15">

<StackPanel Orientation="Horizontal">

<TextBlock Text="PlayerId:" />

<TextBlock Text="{Binding Path=PlayerId}" />

</StackPanel>

<StackPanel Orientation="Horizontal">

<TextBlock Text="Width:" />

<TextBlock Text="{Binding Path=RealWidth}" />

</StackPanel>

<StackPanel Orientation="Horizontal">

<TextBlock Text="Height:" />

<TextBlock Text="{Binding Path=RealHeight}" />

</StackPanel>

</StackPanel>

</DataTemplate>

</ItemsControl.ItemTemplate>

</ItemsControl>

</StackPanel>

</Grid>

</Window>

使用ItemControl的目的是用来显示结果。方法创建了一个对象来存放用户的深度数据以及计算得到的实际宽度和高度值。程序创建了一个这样的对象数组。他是ItemControl的ItemsSource值。UI定义了一个模板用来展示和游戏者深度值相关的数据，这个模板使用的对象取名为PlayerDepthData。下面的名为ClaculatePlayerSize的方法将作为DepthFrameReady事件发生时执行的操作。

private void KinectDevice\_DepthFrameReady(object sender, DepthImageFrameReadyEventArgs e)

{

using (DepthImageFrame frame = e.OpenDepthImageFrame())

{

if (frame != null)

{

frame.CopyPixelDataTo(this.\_DepthPixelData);

CreateBetterShadesOfGray(frame, this.\_DepthPixelData);

**CalculatePlayerSize(frame, this.\_DepthPixelData);**

}

}

}

private void CalculatePlayerSize(DepthImageFrame depthFrame, short[] pixelData)

{

int depth;

int playerIndex;

int pixelIndex;

int bytesPerPixel = depthFrame.BytesPerPixel;

**PlayerDepthData[] players = new PlayerDepthData[6];**

for (int row = 0; row < depthFrame.Height; row++)

{

for (int col = 0; col < depthFrame.Width; col++)

{

pixelIndex = col + (row \* depthFrame.Width);

depth = pixelData[pixelIndex] >> DepthImageFrame.PlayerIndexBitmaskWidth;

if (depth != 0)

{

playerIndex = (pixelData[pixelIndex] & DepthImageFrame.PlayerIndexBitmask) - 1;

if (playerIndex > -1)

{

if (players[playerIndex] == null)

{

players[playerIndex] = new PlayerDepthData(playerIndex + 1, depthFrame.Width, depthFrame.Height);

}

players[playerIndex].UpdateData(col, row, depth);

}

}

}

}

PlayerDepthData.ItemsSource = players;

}

粗体部分代码中使用了PlayerDepthData对象。CalculatePlayerSize方法遍历深度图像中的象元，然后提取游戏者索引位及其对应的深度值。算法忽略了所有深度值为0的象元以及游戏者之外的象元。对于游戏者的每一个象元，方法调用PlayerDepthData对象的UpdateData方法。处理完所有象元之后，将游戏者数组复制给名为PlayerDepthData的ItemControl对象的数据源。对游戏者宽度高度的计算封装在PlayerDepthData这一对象中。

PlayerDepthData对象的代码如下：

class PlayerDepthData

{

#region Member Variables

private const double MillimetersPerInch = 0.0393700787;

private static readonly double HorizontalTanA = Math.Tan(57.0 / 2.0 \* Math.PI / 180);

private static readonly double VerticalTanA = Math.Abs(Math.Tan(43.0 / 2.0 \* Math.PI / 180));

private int \_DepthSum;

private int \_DepthCount;

private int \_LoWidth;

private int \_HiWidth;

private int \_LoHeight;

private int \_HiHeight;

#endregion Member Variables

#region Constructor

public PlayerDepthData(int playerId, double frameWidth, double frameHeight)

{

this.PlayerId = playerId;

this.FrameWidth = frameWidth;

this.FrameHeight = frameHeight;

this.\_LoWidth = int.MaxValue;

this.\_HiWidth = int.MinValue;

this.\_LoHeight = int.MaxValue;

this.\_HiHeight = int.MinValue;

}

#endregion Constructor

#region Methods

public void UpdateData(int x, int y, int depth)

{

this.\_DepthCount++;

this.\_DepthSum += depth;

this.\_LoWidth = Math.Min(this.\_LoWidth, x);

this.\_HiWidth = Math.Max(this.\_HiWidth, x);

this.\_LoHeight = Math.Min(this.\_LoHeight, y);

this.\_HiHeight = Math.Max(this.\_HiHeight, y);

}

#endregion Methods

#region Properties

public int PlayerId { get; private set; }

public double FrameWidth { get; private set; }

public double FrameHeight { get; private set; }

public double Depth

{

get { return this.\_DepthSum / (double)this.\_DepthCount; }

}

public int PixelWidth

{

get { return this.\_HiWidth - this.\_LoWidth; }

}

public int PixelHeight

{

get { return this.\_HiHeight - this.\_LoHeight; }

}

public string RealWidth

{

get

{

double inches = this.RealWidthInches;

return string.Format("{0:0.0}mm", inches \* 25.4);

}

}

public string RealHeight

{

get

{

double inches = this.RealHeightInches;

return string.Format("{0:0.0}mm", inches \* 25.4);

}

}

public double RealWidthInches

{

get

{

double opposite = this.Depth \* HorizontalTanA;

return this.PixelWidth \* 2 \* opposite / this.FrameWidth \* MillimetersPerInch;

}

}

public double RealHeightInches

{

get

{

double opposite = this.Depth \* VerticalTanA;

return this.PixelHeight \* 2 \* opposite / this.FrameHeight \* MillimetersPerInch;

}

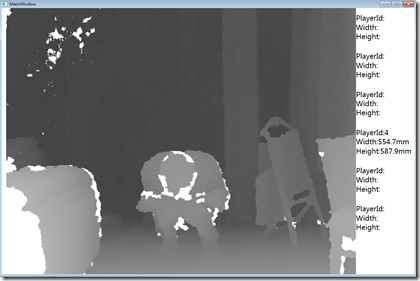
}

#endregion Properties

}

单独编写PlayerDepthData这个类的原因是封装计算逻辑。这个类有两个输入点和两个输出点。构造函数以及UpdateData方法是两个输入点。ReadlWith和RealHeight两个属性是两个输出点。这两个属性是基于上图中的公式计算得出的。公式使用平均深度值，深度数据帧的宽度和高度，和游戏者总共所占有的象元。平均深度值和所有的象元是通过参数传入到UpdateData方法中然后计算的出来的。真实的宽度和高度值是基于UpdateData方法提供的数据计算出来的。下面是我做的6个动作的不同截图，右边可以看到测量值，手上拿了键盘用来截图。

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204050121125522.png)[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204050121226973.png)

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204050121303050.png)[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204050121442088.png)

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204050122006533.png)[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204050122121101.png)

以上测量结果只是以KinectSensor能看到的部分来进行计算的。拿上图1来说。显示的高度是1563mm，宽度为622mm。这里高度存在偏差，实际高度应该是1665左右，可能是脚部和头部测量有误差。以上代码可以同时测量6个游戏者，但是由于只有我一个人，所以做了6个不同的动作，截了6次图。还可以看到一点的是，如上面所讨论的，当只有一个游戏者时，游戏者索引值不一定是从1开始，从上面6幅图可以看出，进出视野会导致游戏者索引值发生变化，值是不确定的。

**4.深度值图像和视频图像的叠加**

在之前的例子中，我们将游戏者所属的象元用黑色显示出来，而其他的用白色显示，这样就达到了提取人物的目的。我们也可以将人物所属的象元用彩色表示，而将其他部分用白色表示。但是，有时候我们想用深度数据中游戏者所属的象元获取对应的彩色影像数据并叠加到视频图像中。这在电视制作和电影制作中很常见，这种技术叫做绿屏抠像，就是演员或者播音员站在绿色底板前，然后录完节目后，绿色背景抠出，换成其他场景，在一些科幻电影中演员不可能在实景中表演时常采用的造景手法。我们平常照证件照时，背景通常是蓝色或者红色，这样也是便于选取背景颜色方便抠图的缘故。在Kinect中我们也可以达到类似的效果。Kinect SDK使得这个很容易实现。

Note:这是现实增强的一个基本例子，现实增应用非常有趣而且能够获得非常好的用于体验。许多艺术家使用Kinect来进行现实增强交互时展览。另外，这种技术也通常作为广告和营销的工具。

前面的例子中，我们能够判断哪个像素是否有游戏者。但是这个只能对于景深数据使用。不幸的是，景深数据影像的象元不能转换到彩色影像中去，即使两者使用相同的分辨率。因为这两个摄像机位于Kinect上的不同位置，所以产生的影像不能够叠加到一起。就像人的两只眼睛一样，当你只睁开左眼看到的景象和只睁开右眼看到的景象是不一样的，人脑将这两只眼睛看到的景物融合成一幅合成的景象。

幸运的是，Kinect SDK提供了一些方法来方便我们进行这些转换，这些方法位于KinectSensor对象中，他们是MapDepthToColorImagePoint，MapDepthToSkeletonPoint，MapSkeletonPointToColor和MapSkeletonPointToDepth。在DepthImageFrame对象中这些方法的名字有点不同(MapFromSkeletonPoint,MapToColorImagePoint及MapToSkeletonPoint)，但功能是相似的。在下面的例子中，我们使用MapDepthToColorImagePoint方法来将景深影像中游戏者所属的象元转换到对应的彩色影像中去。细心的读者可能会发现，没有一个方法能够将彩色影像中的象元转换到对应的景深影像中去。

创建一个新的工程，添加两个Image对象。第一个Image是背景图片。第二个Image是前景图像。在这个例子中，为了使景深影像和彩色影像尽可能的接近，我们采用轮询的方式。每一个影像都有一个Timestamp对象，我们通过比较数据帧的这个值来确定他们是否足够近。注册KinectSensor对象的AllFrameReady事件，并不能保证不同数据流产生的数据帧时同步的。这些帧不可能同时产生，但是轮询模式能够使得不同数据源产生的帧能够尽可能的够近。下面的代码展现了实现方式:

private KinectSensor \_KinectDevice;

private WriteableBitmap \_GreenScreenImage;

private Int32Rect \_GreenScreenImageRect;

private int \_GreenScreenImageStride;

private short[] \_DepthPixelData;

private byte[] \_ColorPixelData;

private bool \_DoUsePolling;

private void CompositionTarget\_Rendering(object sender, EventArgs e)

{

DiscoverKinect();

if (this.KinectDevice != null)

{

try

{

using (ColorImageFrame colorFrame = this.KinectDevice.ColorStream.OpenNextFrame(100))

{

using (DepthImageFrame depthFrame = this.KinectDevice.DepthStream.OpenNextFrame(100))

{

**RenderGreenScreen(this.KinectDevice, colorFrame, depthFrame);**

}

}

}

catch (Exception)

{

//Do nothing, because the likely result is that the Kinect has been unplugged.

}

}

}

private void DiscoverKinect()

{

if (this.\_KinectDevice != null && this.\_KinectDevice.Status != KinectStatus.Connected)

{

UninitializeKinectSensor(this.\_KinectDevice);

this.\_KinectDevice = null;

}

if (this.\_KinectDevice == null)

{

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

if (this.\_KinectDevice != null)

{

InitializeKinectSensor(this.\_KinectDevice);

}

}

}

private void InitializeKinectSensor(KinectSensor sensor)

{

if (sensor != null)

{

sensor.DepthStream.Range = DepthRange.Default;

sensor.SkeletonStream.Enable();

**sensor.DepthStream.Enable(DepthImageFormat.Resolution640x480Fps30);**

**sensor.ColorStream.Enable(ColorImageFormat.RgbResolution1280x960Fps12);**

DepthImageStream depthStream = sensor.DepthStream;

this.\_GreenScreenImage = new WriteableBitmap(depthStream.FrameWidth, depthStream.FrameHeight, 96, 96, PixelFormats.Bgra32, null);

this.\_GreenScreenImageRect = new Int32Rect(0, 0, (int)Math.Ceiling(this.\_GreenScreenImage.Width), (int)Math.Ceiling(this.\_GreenScreenImage.Height));

this.\_GreenScreenImageStride = depthStream.FrameWidth \* 4;

this.GreenScreenImage.Source = this.\_GreenScreenImage;

this.\_DepthPixelData = new short[this.\_KinectDevice.DepthStream.FramePixelDataLength];

this.\_ColorPixelData = new byte[this.\_KinectDevice.ColorStream.FramePixelDataLength];

if (!this.\_DoUsePolling)

{

sensor.AllFramesReady += KinectDevice\_AllFramesReady;

}

sensor.Start();

}

}

private void UninitializeKinectSensor(KinectSensor sensor)

{

if (sensor != null)

{

sensor.Stop();

sensor.ColorStream.Disable();

sensor.DepthStream.Disable();

sensor.SkeletonStream.Disable();

sensor.AllFramesReady -= KinectDevice\_AllFramesReady;

}

}

以上代码有三个地方加粗。第一地方引用了RenderGreenScreen方法。第二个和第三个地方我们初始化了彩色和景深数据流。当在两个图像之间转换时，将彩色图形的分辨率设成景深数据的两倍能够得到最好的转换效果。

RenderGreenScreen方法中执行实际的转换操作。首先通过移除没有游戏者的象元创建一个新的彩色影像。算法遍历景深数据的每一个象元，然后判断游戏者索引是否有有效值。然后获取景深数据中游戏者所属象元对应的彩色图像上的象元，将获取到的象元存放在象元数组中。代码如下：

private void RenderGreenScreen(KinectSensor kinectDevice, ColorImageFrame colorFrame, DepthImageFrame depthFrame)

{

if (kinectDevice != null && depthFrame != null && colorFrame != null)

{

int depthPixelIndex;

int playerIndex;

int colorPixelIndex;

ColorImagePoint colorPoint;

int colorStride = colorFrame.BytesPerPixel \* colorFrame.Width;

int bytesPerPixel = 4;

byte[] playerImage = new byte[depthFrame.Height \* this.\_GreenScreenImageStride];

int playerImageIndex = 0;

depthFrame.CopyPixelDataTo(this.\_DepthPixelData);

colorFrame.CopyPixelDataTo(this.\_ColorPixelData);

for (int depthY = 0; depthY < depthFrame.Height; depthY++)

{

for (int depthX = 0; depthX < depthFrame.Width; depthX++, playerImageIndex += bytesPerPixel)

{

depthPixelIndex = depthX + (depthY \* depthFrame.Width);

playerIndex = this.\_DepthPixelData[depthPixelIndex] & DepthImageFrame.PlayerIndexBitmask;

if (playerIndex != 0)

{

colorPoint = kinectDevice.MapDepthToColorImagePoint(depthFrame.Format, depthX, depthY, this.\_DepthPixelData[depthPixelIndex], colorFrame.Format);

colorPixelIndex = (colorPoint.X \* colorFrame.BytesPerPixel) + (colorPoint.Y \* colorStride);

playerImage[playerImageIndex] = this.\_ColorPixelData[colorPixelIndex]; //Blue

playerImage[playerImageIndex + 1] = this.\_ColorPixelData[colorPixelIndex + 1]; //Green

playerImage[playerImageIndex + 2] = this.\_ColorPixelData[colorPixelIndex + 2]; //Red

playerImage[playerImageIndex + 3] = 0xFF; //Alpha

}

}

}

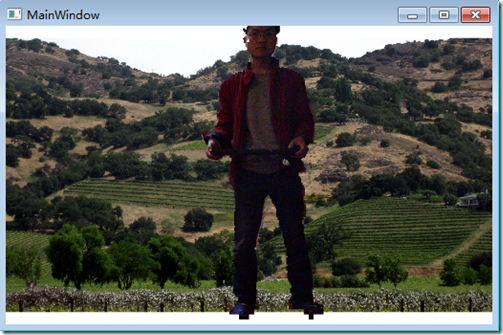
this.\_GreenScreenImage.WritePixels(this.\_GreenScreenImageRect, playerImage, this.\_GreenScreenImageStride, 0);

}

}

PlayerImage位数组存储了所有属于游戏者的彩色影像象元。从景深数据对应位置获取到的彩色影像象元的大小和景深数据象元大小一致。与景深数据每一个象元占两个字节不同。彩色影像数据每个象元占4个字节，蓝绿红以及Alpha值各占一个字节，在本例中Alpha值很重要，它用来确定每个象元的透明度，游戏者所拥有的象元透明度设置为255(0xFF)不透明而其他物体则设置为0，表示透明。

MapDepthToColorImagePoint方法接受景深象元位置以及深度值，返回对应的对应彩色影像中象元的位置。剩下的代码获取游戏者在彩色影像中的象元并将其存储到PlayerImage数组中。当处理完所有的景深数据象元后，代码更新Image的数据源。运行程序后，需要站立一段时间后人物才能够显示出来，如果移动太快，可能出来不了，因为景深数据和彩色数据不能够对齐，可以看到任务轮廓有一些锯齿和噪声，但要处理这些问题还是有点麻烦的，它需要对象元进行平滑。要想获得最好的效果，可以将多帧彩色影像合称为一帧。运行程序后结果如下图，端了个键盘，人有点挫：

[](http://images.cnblogs.com/cnblogs_com/yangecnu/201204/201204050122274532.png)

**5.结语**

本文首先介绍了关于景深数据的简单图像数据，包括景深数据的直方图显示以及一些图像处理相关的算法，然后介绍了景深数据中的游戏者索引位，借助索引位，我们实现了人物宽度和高度的计算，最后借助景深数据结合彩色影像数据，将景深影像和视频图像进行了叠加。

至此，景深数据处理介绍完了，后面将会开始介绍Kinect的骨骼追踪技术，敬请期待。

点击[此处](http://files.cnblogs.com/yangecnu/KinectSDK_Depth_Image_Processing_Part2.rar" \t "_blank)下载本文所有代码，希望对您了解Kinect SDK有所帮助！