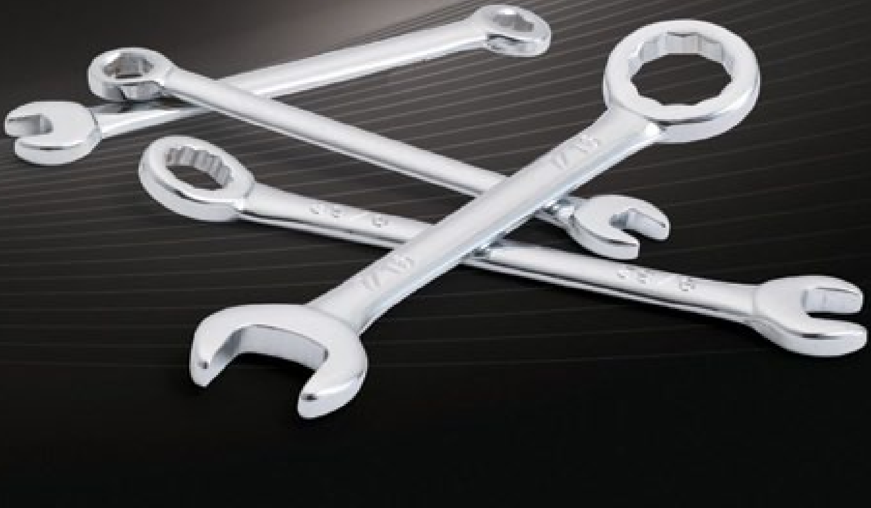
Microsoft

Programming with the Kinect™ for Windows® Software Development Kit



David Catuhe

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2012944940
ISBN: 978-0-7356-6681-8

Printed and bound in the United States of America.

First Printing

Acquisitions Editor: Devon Musgrave
Developmental Editors: Devon Musgrave and Carol Dillingham
Project Editor: Carol Dillingham
Editorial Production: Megan Smith-Creed
Technical Reviewer: Pierce Bizzaca; Technical Review services provided by Content Master, a member of CM Group, Ltd.
Copyeditor: Julie Hotchkiss
Indexer: Perri Weinberg-Schenker
Cover: Twist Creative • Seattle

This book is dedicated to my beloved wife, Sylvie. Without you, your patience, and all you do for me, nothing could be possible.

Contents at a Glance

Contents

Introduction

I am always impressed when science fiction and reality meet. With Kinect for Windows, this is definitely the case, and it is exciting to be able to control the computer with only our hands, without touching any devices, just like in the movie "Minority Report."

I fell in love with Kinect for Windows the first time I tried it. Being able to control my computer with gestures and easily create augmented reality applications was like a dream come true for me. The ability to create an interface that utilizes the movements of the user fascinated me, and that is why I decided to create a toolbox for Kinect for Windows to simplify the detection of gestures and postures.

This book is the story of that toolbox. Each chapter allows you to add new tools to your Kinect toolbox. And at the end, you will find yourself with a complete working set of utilities for creating applications with Kinect for Windows.

Who should read this book

Kinect for Windows offers an extraordinary new way of communicating with the computer. And every day, I see plenty of developers who have great new ideas about how to use it—they want to set up Kinect and get to work.

If you are one of these developers, this book is for you. Through sample code, this book will show you how the Kinect for Windows Software Development Kit works–and how you can develop your own experience with a Kinect sensor.

Assumptions

For the sake of simplification, I use C# as the primary language for samples, but you can use other .NET languages or even C++ with minimal additional effort. The sample code in this book also uses WPF 4.0 as a hosting environment. This book expects that you have at least a minimal understanding of C#, WPF development, .NET development, and object-oriented programming concepts.

Who should not read this book

This book is focused on providing the reader with sample code to show the possibilities of developing with the Kinect for Windows SDK, and it is clearly written for developers, by a developer. If you are not a developer or someone with coding skills, you might consider reading a more introductory book such as *Start Here! Learn the Kinect API* by Rob Miles (Microsoft Press, 2012).

- 2 GB of RAM
- Graphics card that supports DirectX 9.0c
- Kinect for Windows sensor

Depending on your Windows configuration, you might require Local Administrator rights to install or configure Visual Studio 2010.

Code samples

Most of the chapters in this book include code samples that let you interactively try out new material learned in the main text. All sample projects can be downloaded from the following page:

http://www.microsoftpressstore.com/title/9780735666818

Follow the instructions to download the KinectToolbox.zip file.

Note In addition to the code samples, your system should have Visual Studio.

Installing the code samples

Follow these steps to install the code samples on your computer so that you can use them with the exercises in this book.

1. Unzip the KinectToolbox.zip file that you downloaded from the book's website (name a specific directory along with directions to create it, if necessary).
2. If prompted, review the displayed end user license agreement. If you accept the terms, select the accept option, and then click Next.

Note If the license agreement doesn't appear, you can access it from the same web page from which you downloaded the KinectToolbox.zip file.

Using the code samples

The folder created by the Setup.exe program contains the source code required to compile the Kinect toolbox. To load it, simply double-click the Kinect.Toolbox.sln project.

- 2 GB of RAM
- Graphics card that supports DirectX 9.0c
- Kinect for Windows sensor

Depending on your Windows configuration, you might require Local Administrator rights to install or configure Visual Studio 2010.

Code samples

Most of the chapters in this book include code samples that let you interactively try out new material learned in the main text. All sample projects can be downloaded from the following page:

http://go.microsoft.com/FWLink/?Linkid=258661

Follow the instructions to download the KinectToolbox.zip file.

Note In addition to the code samples, your system should have Visual Studio.

Installing the code samples

Follow these steps to install the code samples on your computer so that you can use them with the exercises in this book.

1. Unzip the KinectToolbox.zip file that you downloaded from the book's website (name a specific directory along with directions to create it, if necessary).
2. If prompted, review the displayed end user license agreement. If you accept the terms, select the accept option, and then click Next.

Note If the license agreement doesn't appear, you can access it from the same web page from which you downloaded the KinectToolbox.zip file.

Using the code samples

The folder created by the Setup.exe program contains the source code required to compile the Kinect toolbox. To load it, simply double-click the Kinect.Toolbox.sln project.

Acknowledgments

I'd like to thank the following people: Devon Musgrave for giving me the opportunity to write this book. Dan Fernandez for thinking of me as a potential author for a book about Kinect. Carol Dillingham for her kindness and support. Eric Mittelette for encouraging me from the first time I told him about this project. Eric Vernié, my fellow speaker in numerous sessions during which we presented Kinect.

Errata & book support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site:

http://www.microsoftpressstore.com/title/9780735666818

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software is not offered through the addresses above.

We want to hear from you

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://www.microsoft.com/learning/booksurvey

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*.

CHAPTER 3

Displaying Kinect data

Because there is no physical interaction between the user and the Kinect sensor, you must be sure that the sensor is set up correctly. The most efficient way to accomplish this is to provide a visual feedback of what the sensor receives. Do not forget to add an option in your applications that lets users see this feedback because many will not yet be familiar with the Kinect interface. Even to allow users to monitor the audio, you must provide a visual control of the audio source and the audio level.

In this chapter you will learn how to display the different Kinect streams. You will also write a tool to display skeletons and to locate audio sources.

All the code you produce will target Windows Presentation Foundation (WPF) 4.0 as the default developing environment. The tools will then use all the drawing features of the framework to concentrate only on Kinect-related code.

The color display manager

As you saw in Chapter 2, "Who's there?," Kinect is able to produce a 32-bit RGB color stream. You will now develop a small class (*ColorStreamManager*) that will be in charge of returning a *WriteableBitmap* filled with each frame data.

This *WriteableBitmap* will be displayed by a standard WPF image control called *kinectDisplay*:

```
<Image x:Name="kinectDisplay" Source="{Binding Bitmap}"></Image>
```

This control is bound to a property called *Bitmap* that will be exposed by your class.

Note Before you begin to add code, you must start the Kinect sensor. The rest of the code in this book assumes that you have initialized the sensor as explained in Chapter 1, "A bit of background."

Before writing this class, you must introduce the *Notifier* class that helps handle the *INotifyProperty-Changed* interface (used to signal updates to the user interface [UI]):

```
using System;
using System.ComponentModel;
using System.Linq.Expressions;

namespace Kinect.Toolbox
{
    public abstract class Notifier : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;

        protected void RaisePropertyChanged<T>(Expression<Func<T>> propertyExpression)
        {
            var memberExpression = propertyExpression.Body as MemberExpression;
            if (memberExpression == null)
                return;

            string propertyName = memberExpression.Member.Name;
            if (PropertyChanged != null)
                PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

As you can see, this class uses an expression to detect the name of the property to signal. This is quite useful, because with this technique you don't have to pass a string (which is hard to keep in sync with your code when, for example, you rename your properties) to define your property.

You are now ready to write the *ColorStreamManager* class:

```
using System.Windows.Media.Imaging;
using System.Windows.Media;
using Microsoft.Kinect;
using System.Windows;
public class ColorStreamManager : Notifier
{
    public WriteableBitmap Bitmap { get; private set; }

    public void Update(ColorImageFrame frame)
    {
        var pixelData = new byte[frame.PixelDataLength];

        frame.CopyPixelDataTo(pixelData);

        if (Bitmap == null)
        {
            Bitmap = new WriteableBitmap(frame.Width, frame.Height,
                                         96, 96, PixelFormats.Bgr32, null);
        }

        int stride = Bitmap.PixelWidth * Bitmap.Format.BitsPerPixel / 8;
        Int32Rect dirtyRect = new Int32Rect(0, 0, Bitmap.PixelWidth, Bitmap.PixelHeight);
        Bitmap.WritePixels(dirtyRect, pixelData, stride, 0);

        RaisePropertyChanged(() => Bitmap);
    }
}
```

Using the frame object, you can get the size of the frame with *PixelDataLength* and use it to create a byte array to receive the content of the frame. The frame can then be used to copy its content to the buffer using *CopyPixelDataTo*.

The class creates a *WriteableBitmap* on first call of *Update*. This bitmap is returned by the *Bitmap* property (used as binding source for the image control). Notice that the bitmap must be a BGR32 (Windows works with Blue/Green/Red picture) with 96 dots per inch (DPI) on the x and y axes.

The *Update* method simply copies the buffer to the *WriteableBitmap* on each frame using the *Write-Pixels* method of *WriteableBitmap*.

Finally, *Update* calls *RaisePropertyChanged* (from the *Notifier* class) on the *Bitmap* property to signal that the bitmap has been updated.

So after initializing the sensor, you can add this code in your application to use the *ColorStream-Manager* class:

```
var colorManager = new ColorStreamManager();
void kinectSensor_ColorFrameReady(object sender, ColorImageFrameReadyEventArgs e)
{
    using (var frame = e.OpenColorImageFrame())
    {
        if (frame == null)
            return;

        colorManager.Update(frame);
    }
}
```

The final step is to bind the *DataContext* of the picture to the *colorManager* object (for instance, inside the load event of your *MainWindow* page):

```
kinectDisplay.DataContext = colorManager;
```

Now every time a frame is available, the *ColorStreamManager* bound to the image will raise the *PropertyChanged* event for its *Bitmap* property, and in response the image will be updated, as shown in Figure 3-1.

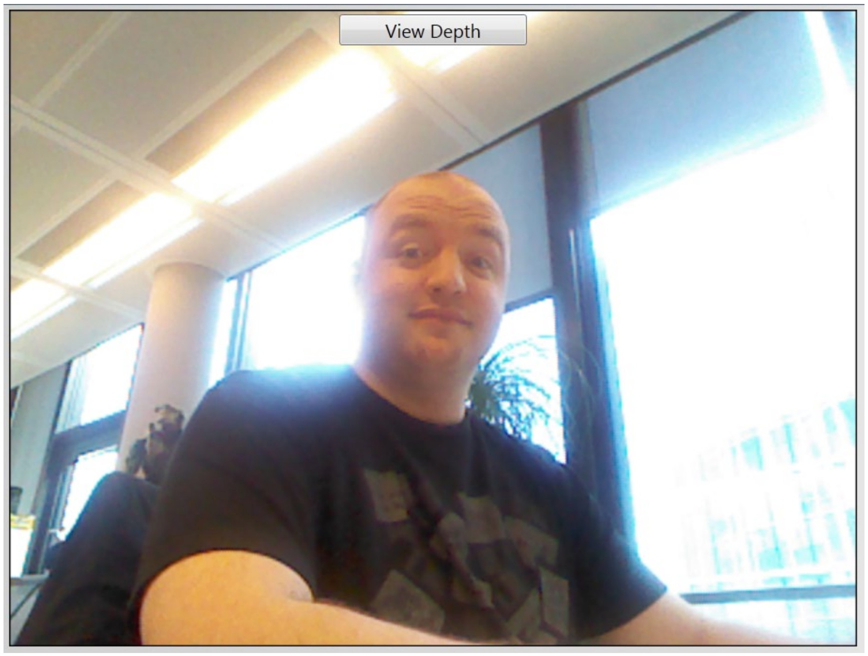


FIGURE 3-1 Displaying the Kinect color stream with WPF.

If you are planning to use the YUV format, there are two possibilities available: You can use the *ColorImageFormat.YuvResolution640x480Fps15* format, which is already converted to RGB32, or you can decide to use the raw YUV format (*ColorImageFormat.RawYuvResolution640x480Fps15*), which is composed of 16 bits per pixel—and it is more effective.

To display this format, you must update your *ColorStreamManager*:

```
public class ColorStreamManager : Notifier
{
    public WriteableBitmap Bitmap { get; private set; }
    int[] yuvTemp;

    static double Clamp(double value)
    {
        return Math.Max(0, Math.Min(value, 255));
    }

    static int ConvertFromYUV(byte y, byte u, byte v)
    {
        byte b = (byte)Clamp(1.164 * (y - 16) + 2.018 * (u - 128));
        byte g = (byte)Clamp(1.164 * (y - 16) - 0.813 * (v - 128) - 0.391 * (u - 128));
        byte r = (byte)Clamp(1.164 * (y - 16) + 1.596 * (v - 128));

        return (r << 16) + (g << 8) + b;
    }

    public void Update(ColorImageFrame frame)
    {
        var pixelData = new byte[frame.PixelDataLength];

        frame.CopyPixelDataTo(pixelData);

        if (Bitmap == null)
        {
            Bitmap = new WriteableBitmap(frame.Width, frame.Height,
                                         96, 96, PixelFormats.Bgr32, null);
        }

        int stride = Bitmap.PixelWidth * Bitmap.Format.BitsPerPixel / 8;
        Int32Rect dirtyRect = new Int32Rect(0, 0, Bitmap.PixelWidth, Bitmap.PixelHeight);

        if (frame.Format == ColorImageFormat.RawYuvResolution640x480Fps15)
        {
            if (yuvTemp == null)
                yuvTemp = new int[frame.Width * frame.Height];

            int current = 0;
            for (int uyvyIndex = 0; uyvyIndex < pixelData.Length; uyvyIndex += 4)
            {
                byte u = pixelData[uyvyIndex];
                byte y1 = pixelData[uyvyIndex + 1];
                byte v = pixelData[uyvyIndex + 2];
                byte y2 = pixelData[uyvyIndex + 3];

                yuvTemp[current++] = ConvertFromYUV(y1, u, v);
                yuvTemp[current++] = ConvertFromYUV(y2, u, v);
            }

            Bitmap.WritePixels(dirtyRect, yuvTemp, stride, 0);
        }
        else
            Bitmap.WritePixels(dirtyRect, pixelData, stride, 0);

        RaisePropertyChanged(() => Bitmap);
    }
}
```

The *ConvertFromYUV* method is used to convert a (y, u, v) vector to an RGB integer. Because this operation can produce out-of-bounds results, you must use the *Clamp* method to obtain correct values.

The important point to understand about this is how YUV values are stored in the stream. A YUV stream stores pixels with 32 bits for each two pixels, using the following structure: 8 bits for Y1, 8 bits for U, 8 bits for Y2, and 8 bits for V. The first pixel is composed from Y1UV and the second pixel is built with Y2UV.

Therefore, you need to run through all incoming YUV data to extract pixels:

```
for (int uyvyIndex = 0; uyvyIndex < pixelData.Length; uyvyIndex += 4)
{
    byte u = pixelData[uyvyIndex];
    byte y1 = pixelData[uyvyIndex + 1];
    byte v = pixelData[uyvyIndex + 2];
    byte y2 = pixelData[uyvyIndex + 3];

    yuvTemp[current++] = ConvertFromYUV(y1, u, v);
    yuvTemp[current++] = ConvertFromYUV(y2, u, v);
}
```

Now the *ColorStreamManager* is able to process all kinds of stream format.

The depth display manager

The second stream you need to display is the depth stream. This stream is composed of 16 bits per pixel, and each pixel in the depth stream uses 13 bits (high order) for depth data and 3 bits (lower order) to identify a player.

A depth data value of 0 indicates that no depth data is available at that position because all the objects are either too close to the camera or too far away from it.

Important When skeleton tracking is disabled, the three bits that identify a player are set to 0.

Note You must configure the depth stream as explained in Chapter 2 before continuing.

Comparable to the *ColorStreamManager* class, following is the code for the *DepthStreamManager* class:

```
using System.Windows.Media.Imaging
using Microsoft.Kinect;
using System.Windows.Media;
using System.Windows;

public class DepthStreamManager : Notifier
{
    byte[] depthFrame32;

    public WriteableBitmap Bitmap { get; private set; }

    public void Update(DepthImageFrame frame)
    {
        var pixelData = new short[frame.PixelDataLength];
        frame.CopyPixelDataTo(pixelData);

        if (depthFrame32 == null)
        {
            depthFrame32 = new byte[frame.Width * frame.Height * 4];
        }

        if (Bitmap == null)
        {
            Bitmap = new WriteableBitmap(frame.Width, frame.Height,
                                    96, 96, PixelFormats.Bgra32, null);
        }

        ConvertDepthFrame(pixelData);

        int stride = Bitmap.PixelWidth * Bitmap.Format.BitsPerPixel / 8;
        Int32Rect dirtyRect = new Int32Rect(0, 0, Bitmap.PixelWidth, Bitmap.PixelHeight);

        Bitmap.WritePixels(dirtyRect, depthFrame32, stride, 0);

        RaisePropertyChanged(() => Bitmap);
    }

    void ConvertDepthFrame(short[] depthFrame16)
    {
        for (int i16 = 0, i32 = 0; i16 < depthFrame16.Length
                && i32 < depthFrame32.Length; i16 ++, i32 += 4)
        {
            int user = depthFrame16[i16] & 0x07;
            int realDepth = (depthFrame16[i16] >> 3);

            byte intensity = (byte)(255 - (255 * realDepth / 0x1fff));

            depthFrame32[i32] = 0;
            depthFrame32[i32 + 1] = 0;
            depthFrame32[i32 + 2] = 0;
            depthFrame32[i32 + 3] = 255;

            switch (user)
            {
                case 0: // no one
                    depthFrame32[i32] = (byte)(intensity / 2);
                    depthFrame32[i32 + 1] = (byte)(intensity / 2);
                    depthFrame32[i32 + 2] = (byte)(intensity / 2);
                    break;
                case 1:
                    depthFrame32[i32] = intensity;
                    break;
                case 2:
                    depthFrame32[i32 + 1] = intensity;
                    break;
                case 3:
                    depthFrame32[i32 + 2] = intensity;
                    break;
                case 4:
                    depthFrame32[i32] = intensity;
                    depthFrame32[i32 + 1] = intensity;
                    break;
                case 5:
                    depthFrame32[i32] = intensity;
                    depthFrame32[i32 + 2] = intensity;
                    break;
                case 6:
                    depthFrame32[i32 + 1] = intensity;
                    depthFrame32[i32 + 2] = intensity;
                    break;
                case 7:
                    depthFrame32[i32] = intensity;
                    depthFrame32[i32 + 1] = intensity;
                    depthFrame32[i32 + 2] = intensity;
                    break;
            }
        }
    }
}
```

The main method here is *ConvertDepthFrame*, where the potential user ID and the depth value (expressed in millimeters) are extracted:

```
int user = depthFrame16[i16] & 0x07;
int realDepth = (depthFrame16[i16] >> 3);
byte intensity = (byte)(255 - (255 * realDepth / 0x1fff));
```

As mentioned in Chapter 2, you simply have to use some bitwise operations to get the information you need out of the pixel. The user index is on the three low-order bits, so a simple mask with 00000111 in binary form or 0x07 in hexadecimal form can extract the value. To get the depth value, you can remove the first three bits by offsetting the pixel to the right with the >> operator.

The intensity is computed by computing a ratio between the maximum depth value and the current depth value. The ratio is then used to get a value between 0 and 255 because color components are expressed using bytes.

The following part of the method generates a grayscale pixel (with the intensity related to the depth), as shown in Figure 3-2. It uses a specific color if a user is detected, as shown in Figure 3-3. (The blue color shown in Figure 3-3 appears as gray to readers of the print book.)

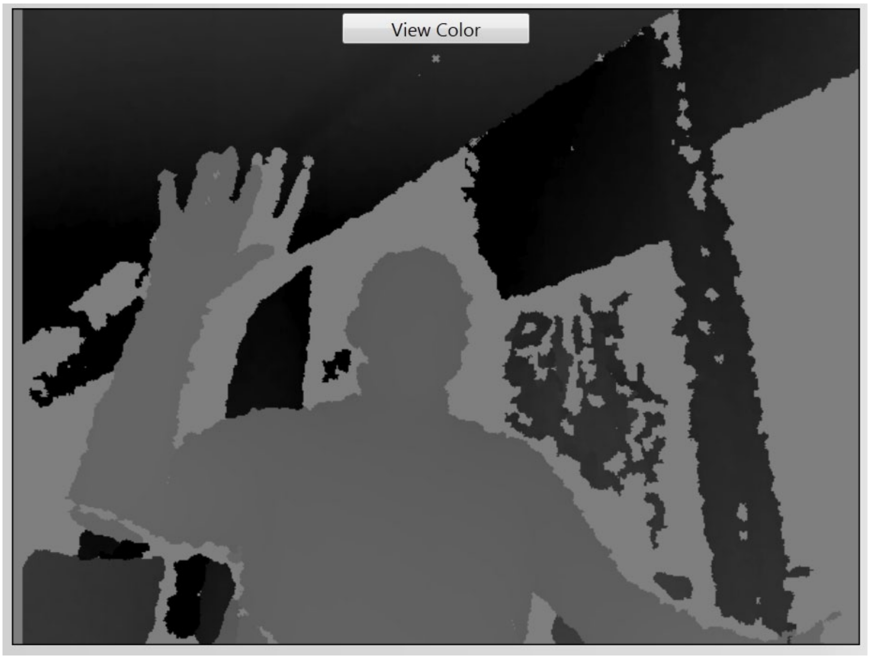


FIGURE 3-2 The depth stream display without a user detected.

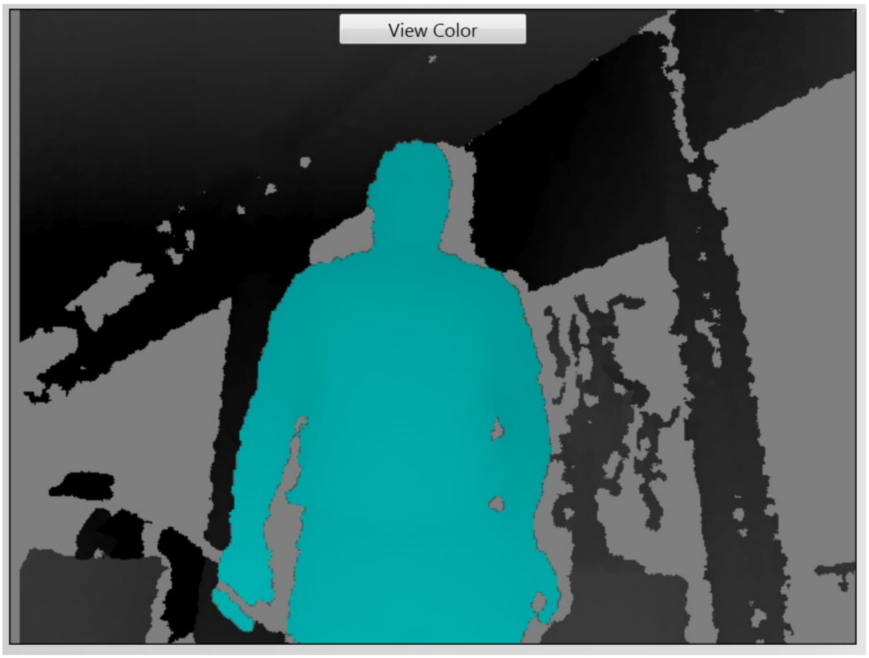


FIGURE 3-3 The depth stream display with a user detected. (A specific color is used where the user is detected, but this appears as light gray to readers of the print book.)

Of course, the near and standard modes are supported the same way by the *DepthStreamManager*. The only difference is that in near mode, the depth values are available from 40cm, whereas in standard mode, the depth values are only available from 80cm, as shown in Figure 3-4.

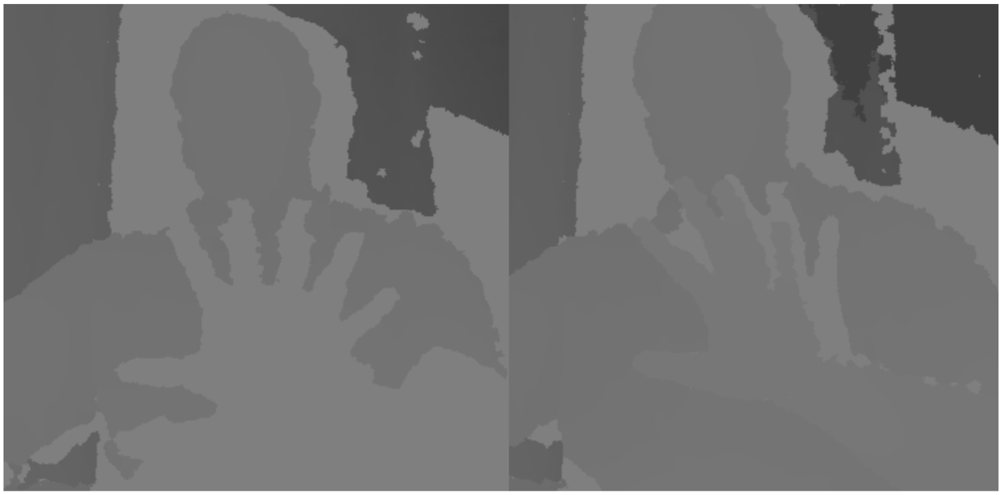


FIGURE 3-4 Hand depth values out of range in standard mode are shown at left, and hand depth values in range in near mode are shown at right.

To connect your *DepthStreamManager* class with the *kinectDisplay* image control, use the following code inside your *kinectSensor_DepthFrameReady* event:

```
var depthManager = new DepthStreamManager();
void kinectSensor_DepthFrameReady(object sender, DepthImageFrameReadyEventArgs e)
{
    using (var frame = e.OpenDepthImageFrame())
    {
        if (frame == null)
            return;

        depthManager.Update(frame);
    }
}
```

Then add this code in your initialization event:

```
kinectDisplay.DataContext = depthManager;
```

The *DepthStreamManager* provides an excellent way to give users visual feedback, because they can detect when and where the Kinect sensor sees them by referring to the colors in the visual display.

The skeleton display manager

The skeleton data is produced by the natural user interface (NUI) API and behaves the same way as the color and depth streams. You have to collect the tracked skeletons to display each of their joints.

You can simply add a WPF canvas to display the final result in your application, as shown in Figure 3-5:

```
<Canvas x:Name="skeletonCanvas"></Canvas>
```

You have to write a class named *SkeletonDisplayManager* that will provide a *Draw* method to create the required shapes inside the *skeletonCanvas* canvas:

```
using System;
using System.Collections.Generic;
using System.Windows;
using System.Windows.Controls;
using System.Linq;
using System.Windows.Shapes;
using System.Windows.Media;
using Microsoft.Kinect;

namespace Kinect.Toolbox
{
    public class SkeletonDisplayManager
    {
        readonly Canvas rootCanvas;
        readonly KinectSensor sensor;

        public SkeletonDisplayManager(KinectSensor kinectSensor, Canvas root)
        {
            rootCanvas = root;
            sensor = kinectSensor;
        }

        public void Draw(Skeleton[] skeletons)
        {
             // Implementation will be shown afterwards
        }
    }
}
```

As you can see, the *Draw* method takes a *Skeletons* array in parameter. To get this array, you can add a new method to your *Tools* class:

```
public static void GetSkeletons(SkeletonFrame frame, ref Skeleton[] skeletons)
{
    if (frame == null)
        return;

    if (skeletons == null || skeletons.Length != frame.SkeletonArrayLength)
    {
        skeletons = new Skeleton[frame.SkeletonArrayLength];
    }
    frame.CopySkeletonDataTo(skeletons);
}
```

This method is similar to the previous one but does not recreate a new array every time, which is important for the sake of performance. When this method is ready, you can add the following code to your load event:

```
Skeleton[] skeletons;
SkeletonDisplayManager skeletonManager = new SkeletonDisplayManager(kinectSensor,
skeletonCanvas);
void kinectSensor_SkeletonFrameReady(object sender, SkeletonFrameReadyEventArgs e)
{
    using (SkeletonFrame frame = e.OpenSkeletonFrame())
    {
        if (frame == null)
            return;

        frame.GetSkeletons(ref skeletons);
        if (skeletons.All(s => s.TrackingState == SkeletonTrackingState.NotTracked))
            return;

        skeletonManager.Draw(skeletons);
    }
}
```

The event argument *e* gives you a method called *OpenSkeletonFrame* that returns a *SkeletonFrame* object. This object is used to get an array of *Skeleton* objects.

Then you simply have to find out if one of the returned skeletons is tracked. If not, you can return and wait for a new frame, or you can use the *skeletonManager* object to display the detected skeletons.

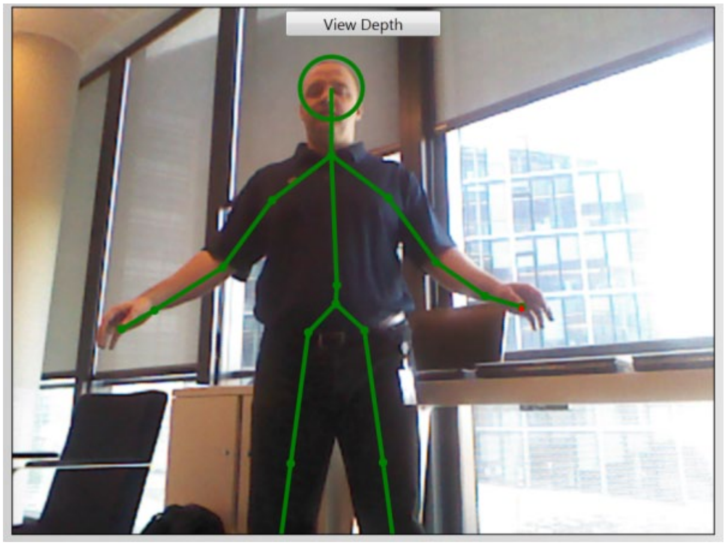


FIGURE 3-5 Displaying the skeleton data.

So, going back to your *SkeletonDisplayManager*, you now need to draw the skeletons inside the WPF canvas. To do so, you can add a list of circles that indicate where the joints are and then draw lines between the joints.

You can get access to a skeleton's joints collection easily using the *skeleton.Joints* property. To draw all the detected and tracked skeletons in a frame, you simply cycle through the *Skeletons* array with the following code:

```
public void Draw(Skeleton[] skeletons)
{
rootCanvas.Children.Clear();
foreach (Skeleton skeleton in skeletons)
{
    if (skeleton.TrackingState != SkeletonTrackingState.Tracked)
        continue;

    Plot(JointType.HandLeft, skeleton.Joints);
    Trace(JointType.HandLeft, JointType.WristLeft, skeleton.Joints);
    Plot(JointType.WristLeft, skeleton.Joints);
    Trace(JointType.WristLeft, JointType.ElbowLeft, skeleton.Joints);
    Plot(JointType.ElbowLeft, skeleton.Joints);
    Trace(JointType.ElbowLeft, JointType.ShoulderLeft, skeleton.Joints);
    Plot(JointType.ShoulderLeft, skeleton.Joints);
    Trace(JointType.ShoulderLeft, JointType.ShoulderCenter, skeleton.Joints);
    Plot(JointType.ShoulderCenter, skeleton.Joints);

    Trace(JointType.ShoulderCenter, JointType.Head, skeleton.Joints);

    Plot(JointType.Head, JointType.ShoulderCenter, skeleton.Joints);

    Trace(JointType.ShoulderCenter, JointType.ShoulderRight, skeleton.Joints);
    Plot(JointType.ShoulderRight, skeleton.Joints);
    Trace(JointType.ShoulderRight, JointType.ElbowRight, skeleton.Joints);
    Plot(JointType.ElbowRight, skeleton.Joints);
    Trace(JointType.ElbowRight, JointType.WristRight, skeleton.Joints);
    Plot(JointType.WristRight, skeleton.Joints);
    Trace(JointType.WristRight, JointType.HandRight, skeleton.Joints);
    Plot(JointType.HandRight, skeleton.Joints);

    Trace(JointType.ShoulderCenter, JointType.Spine, skeleton.Joints);
    Plot(JointType.Spine, skeleton.Joints);
    Trace(JointType.Spine, JointType.HipCenter, skeleton.Joints);
    Plot(JointType.HipCenter, skeleton.Joints);

    Trace(JointType.HipCenter, JointType.HipLeft, skeleton.Joints);
    Plot(JointType.HipLeft, skeleton.Joints);
    Trace(JointType.HipLeft, JointType.KneeLeft, skeleton.Joints);
    Plot(JointType.KneeLeft, skeleton.Joints);
    Trace(JointType.KneeLeft, JointType.AnkleLeft, skeleton.Joints);
    Plot(JointType.AnkleLeft, skeleton.Joints);
    Trace(JointType.AnkleLeft, JointType.FootLeft, skeleton.Joints);
    Plot(JointType.FootLeft, skeleton.Joints);

    Trace(JointType.HipCenter, JointType.HipRight, skeleton.Joints);
    Plot(JointType.HipRight, skeleton.Joints);
    Trace(JointType.HipRight, JointType.KneeRight, skeleton.Joints);
    Plot(JointType.KneeRight, skeleton.Joints);
    Trace(JointType.KneeRight, JointType.AnkleRight, skeleton.Joints);
    Plot(JointType.AnkleRight, skeleton.Joints);
    Trace(JointType.AnkleRight, JointType.FootRight, skeleton.Joints);
    Plot(JointType.FootRight, skeleton.Joints);
}
}
```

The *Trace* and *Plot* methods search for a given joint through the *Joints* collection. The *Trace* method traces a line between two joints and then the *Plot* method draws a point where the joint belongs.

Before looking at these methods, you must add some more code to your project. First add a *Vector2* class that represents a two-dimensional (2D) coordinate (x, y) with associated simple operators (+, -, *, etc.):

```
using System;

namespace Kinect.Toolbox
{
    [Serializable]
    public struct Vector2
    {
        public float X;
        public float Y;

        public static Vector2 Zero
        {
            get
            {
                return new Vector2(0, 0);
            }
        }

        public Vector2(float x, float y)
        {
            X = x;
            Y = y;
        }

        public float Length
        {
            get
            {
                return (float)Math.Sqrt(X * X + Y * Y);
            }
        }

        public static Vector2 operator -(Vector2 left, Vector2 right)
        {
            return new Vector2(left.X - right.X, left.Y - right.Y);
        }

        public static Vector2 operator +(Vector2 left, Vector2 right)
        {
            return new Vector2(left.X + right.X, left.Y + right.Y);
        }

        public static Vector2 operator *(Vector2 left, float value)
        {
            return new Vector2(left.X * value, left.Y * value);
        }

        public static Vector2 operator *(float value, Vector2 left)
        {
            return left * value;
        }

        public static Vector2 operator /(Vector2 left, float value)
        {
            return new Vector2(left.X / value, left.Y / value);
        }

    }

}
```

There is nothing special to note in the previous code; it is simple 2D math.

The second step involves converting the joint coordinates from skeleton space (x, y, z in meter units) to screen space (in pixel units). To do so, you can add a *Convert* method to your *Tools* class:

```
  public static Vector2 Convert(KinectSensor sensor, SkeletonPoint position)
        {
            float width = 0;
            float height = 0;
            float x = 0;
            float y = 0;

            if (sensor.ColorStream.IsEnabled)
            {
                var colorPoint = sensor.MapSkeletonPointToColor(position,
sensor.ColorStream.Format);
                x = colorPoint.X;
                y = colorPoint.Y;

                switch (sensor.ColorStream.Format)
                {
                    case ColorImageFormat.RawYuvResolution640x480Fps15:
                    case ColorImageFormat.RgbResolution640x480Fps30:
                    case ColorImageFormat.YuvResolution640x480Fps15:
                        width = 640;
                        height = 480;
                        break;
                    case ColorImageFormat.RgbResolution1280x960Fps12:
                        width = 1280;
                        height = 960;
                        break;
                }
            }
            else if (sensor.DepthStream.IsEnabled)
            {
                var depthPoint = sensor.MapSkeletonPointToDepth(position,
sensor.DepthStream.Format);
                x = depthPoint.X;
                y = depthPoint.Y;

                switch (sensor.DepthStream.Format)
                {
                    case DepthImageFormat.Resolution80x60Fps30:
                        width = 80;
                    height = 60;
                    break;
                case DepthImageFormat.Resolution320x240Fps30:
                    width = 320;
                    height = 240;
                    break;
                case DepthImageFormat.Resolution640x480Fps30:
                    width = 640;
                    height = 480;
                    break;
            }
        }
        else
        {
            width = 1;
            height = 1;
        }

        return new Vector2(x / width, y / height);
    }
```

The *Convert* method uses the Kinect for Windows SDK mapping API to convert from skeleton space to color or depth space. If the color stream is enabled, it will be used to map the coordinates using the *kinectSensor.MapSkeletonPointToColor* method, and using the color stream format, you can get the width and the height of the color space. If the color stream is disabled, the method uses the depth stream in the same way.

The method gets a coordinate (x, y) and a space size (width, height). Using this information, it returns a new *Vector2* class with an absolute coordinate (a coordinate relative to a unary space).

Then you have to add a private method used to determine the coordinates of a joint inside the canvas to your *SkeletonDisplayManager* class:

```
void GetCoordinates(JointType jointType, IEnumerable<Joint> joints, out float x, out float y)
{
    var joint = joints.First(j => j.JointType == jointType);

    Vector2 vector2 = Convert(kinectSensor, joint.Position);

    x = (float)(vector2.X * rootCanvas.ActualWidth);
    y = (float)(vector2.Y * rootCanvas.ActualHeight);
}
```

With an absolute coordinate, it is easy to deduce the canvas space coordinate of the joint:

```
x = (float)(vector2.X * rootCanvas.ActualWidth);
y = (float)(vector2.Y * rootCanvas.ActualHeight);
```

Finally, with the help of the previous methods, the *Plot* and *Trace* methods are defined as follows:

```
void Plot(JointType centerID, IEnumerable<Joint> joints)
{
    float centerX;
    float centerY;
    GetCoordinates(centerID, joints, out centerX, out centerY);

    const double diameter = 8;

    Ellipse ellipse = new Ellipse
    {
        Width = diameter,
        Height = diameter,
        HorizontalAlignment = HorizontalAlignment.Left,
        VerticalAlignment = VerticalAlignment.Top,
        StrokeThickness = 4.0,
        Stroke = new SolidColorBrush(Colors.Green),
        StrokeLineJoin = PenLineJoin.Round
    };

    Canvas.SetLeft(ellipse, centerX - ellipse.Width / 2);
    Canvas.SetTop(ellipse, centerY - ellipse.Height / 2);

    rootCanvas.Children.Add(ellipse);
}

void Trace(JointType sourceID, JointType destinationID, JointCollection joints)
{
    float sourceX;
    float sourceY;

    GetCoordinates(sourceID, joints, out sourceX, out sourceY);

    float destinationX;
    float destinationY;

    GetCoordinates(destinationID, joints, out destinationX, out destinationY);

    Line line = new Line
                {
                    X1 = sourceX,
                    Y1 = sourceY,
                    X2 = destinationX,
                    Y2 = destinationY,
                    HorizontalAlignment = HorizontalAlignment.Left,
                    VerticalAlignment = VerticalAlignment.Top,
                    StrokeThickness = 4.0,
                    Stroke = new SolidColorBrush(Colors.Green),
                    StrokeLineJoin = PenLineJoin.Round
                };


    rootCanvas.Children.Add(line);
}
```

The main point to remember here is that WPF shapes (*Line* or *Ellipse*) are created to represent parts of the skeleton. After the shape is created, it is added to the canvas.

Note The WPF shapes are recreated at every render. To optimize the display, it is better to keep the shapes and move them to the skeleton as needed, but that is a more complex process that is not required for the scope of this book.

The only specific joint in the skeleton is the head because it makes sense to draw it bigger than the other joints to represent the head of the skeleton. To do so, a new *Plot* method is defined:

```
void Plot(JointType centerID, JointType baseID, JointCollection joints)
{
    float centerX;
    float centerY;

    GetCoordinates(centerID, joints, out centerX, out centerY);

    float baseX;
    float baseY;

    GetCoordinates(baseID, joints, out baseX, out baseY);

    double diameter = Math.Abs(baseY - centerY);

    Ellipse ellipse = new Ellipse
    {
        Width = diameter,
        Height = diameter,
        HorizontalAlignment = HorizontalAlignment.Left,
        VerticalAlignment = VerticalAlignment.Top,
        StrokeThickness = 4.0,
        Stroke = new SolidColorBrush(Colors.Green),
        StrokeLineJoin = PenLineJoin.Round
    };

    Canvas.SetLeft(ellipse, centerX - ellipse.Width / 2);
    Canvas.SetTop(ellipse, centerY - ellipse.Height / 2);

    rootCanvas.Children.Add(ellipse);
}
```

In this case, the ellipse's diameter is defined using the distance between the head and the center of shoulder.

Finally, you can also add a new parameter to the *Draw* method to support the seated mode. In this case, you must not draw the lower body joints:

```
public void Draw(Skeleton[] skeletons, bool seated)
{
    rootCanvas.Children.Clear();
    foreach (Skeleton skeleton in skeletons)
    {
        if (skeleton.TrackingState != SkeletonTrackingState.Tracked)
            continue;
        Plot(JointType.HandLeft, skeleton.Joints);
        Trace(JointType.HandLeft, JointType.WristLeft, skeleton.Joints);
        Plot(JointType.WristLeft, skeleton.Joints);
        Trace(JointType.WristLeft, JointType.ElbowLeft, skeleton.Joints);
        Plot(JointType.ElbowLeft, skeleton.Joints);
        Trace(JointType.ElbowLeft, JointType.ShoulderLeft, skeleton.Joints);
        Plot(JointType.ShoulderLeft, skeleton.Joints);
        Trace(JointType.ShoulderLeft, JointType.ShoulderCenter, skeleton.Joints);
        Plot(JointType.ShoulderCenter, skeleton.Joints);

        Trace(JointType.ShoulderCenter, JointType.Head, skeleton.Joints);

        Plot(JointType.Head, JointType.ShoulderCenter, skeleton.Joints);

        Trace(JointType.ShoulderCenter, JointType.ShoulderRight, skeleton.Joints);
        Plot(JointType.ShoulderRight, skeleton.Joints);
        Trace(JointType.ShoulderRight, JointType.ElbowRight, skeleton.Joints);
        Plot(JointType.ElbowRight, skeleton.Joints);
        Trace(JointType.ElbowRight, JointType.WristRight, skeleton.Joints);
        Plot(JointType.WristRight, skeleton.Joints);
        Trace(JointType.WristRight, JointType.HandRight, skeleton.Joints);
        Plot(JointType.HandRight, skeleton.Joints);

        if (!seated)
        {
            Trace(JointType.ShoulderCenter, JointType.Spine, skeleton.Joints);
            Plot(JointType.Spine, skeleton.Joints);
            Trace(JointType.Spine, JointType.HipCenter, skeleton.Joints);
            Plot(JointType.HipCenter, skeleton.Joints);

            Trace(JointType.HipCenter, JointType.HipLeft, skeleton.Joints);
            Plot(JointType.HipLeft, skeleton.Joints);
            Trace(JointType.HipLeft, JointType.KneeLeft, skeleton.Joints);
            Plot(JointType.KneeLeft, skeleton.Joints);
            Trace(JointType.KneeLeft, JointType.AnkleLeft, skeleton.Joints);
            Plot(JointType.AnkleLeft, skeleton.Joints);
            Trace(JointType.AnkleLeft, JointType.FootLeft, skeleton.Joints);
            Plot(JointType.FootLeft, skeleton.Joints);

            Trace(JointType.HipCenter, JointType.HipRight, skeleton.Joints);
            Plot(JointType.HipRight, skeleton.Joints);
            Trace(JointType.HipRight, JointType.KneeRight, skeleton.Joints);
            Plot(JointType.KneeRight, skeleton.Joints);
            Trace(JointType.KneeRight, JointType.AnkleRight, skeleton.Joints);
            Plot(JointType.AnkleRight, skeleton.Joints);
            Trace(JointType.AnkleRight, JointType.FootRight, skeleton.Joints);
            Plot(JointType.FootRight, skeleton.Joints);
        }
    }
}
```

The audio display manager

The audio stream provides two important pieces of information that the user of your Kinect applications may want to know. The first is the sound source angle, which is the angle (in radians) to the current position of the audio source in camera coordinates.

The second is the beam angle produced by the microphone array. By using the fact that the sound from a particular audio source arrives at each microphone in the array at a slightly different time, beamforming allows applications to determine the direction of the audio source and use the microphone array as a steerable directional microphone.

The beam angle can be important as a visual feedback to indicate which audio source is being used (for speech recognition, for instance), as shown in Figure 3-6.

FIGURE 3-6 Visual feedback of beam angle.

This visual feedback is a virtual representation of the sensor, and in Figure 3-6, the orange area to the right of center (which appears as gray in the print book) indicates the direction of the beam. (For readers of the print book, Figure 3-6 is orange near the center and fades to black on either side of the beam.)

To recreate the same control, you can add an XAML page with the following XAML declaration:

```
<Rectangle x:Name="audioBeamAngle" Height="20" Width="300" Margin="5">
    <Rectangle.Fill>
        <LinearGradientBrush StartPoint="0,0" EndPoint="1, 0">
            <GradientStopCollection>
                <GradientStop Offset="0" Color="Black"/>
                <GradientStop Offset="{Binding BeamAngle}" Color="Orange"/>
                <GradientStop Offset="1" Color="Black"/>
            </GradientStopCollection>
        </LinearGradientBrush>
    </Rectangle.Fill>
</Rectangle>
```

You can see that the rectangle is filled with a *LinearGradientBrush* starting from black to orange to black. The position of the orange *GradientStop* can be bound to a *BeamAngle* property exposed by a class.

The binding code itself is quite obvious:

```
var kinectSensor = KinectSensor.KinectSensors[0];
var audioManager = new AudioStreamManager(kinectSensor.AudioSource);
audioBeamAngle.DataContext = audioManager;
```

So you have to create an *AudioStreamManager* class that exposes a *BeamAngle* property. The class inherits from the *Notifier* class you created earlier in this chapter and implements *IDisposable*:

```
using Microsoft.Kinect;
public class AudioStreamManager : Notifier, IDisposable
{
    readonly KinectAudioSource audioSource;

    public AudioStreamManager(KinectAudioSource source)
    {
        audioSource = source;
        audioSource.BeamAngleChanged += audioSource_BeamAngleChanged;
    }

    void audioSource_BeamAngleChanged(object sender, BeamAngleChangedEventArgs e)
    {
        RaisePropertyChanged(()=>BeamAngle);
    }

    public double BeamAngle
    {
        get
        {
            return (audioSource.BeamAngle - KinectAudioSource.MinBeamAngle) /
(KinectAudioSource.MaxBeamAngle - KinectAudioSource.MinBeamAngle);
        }
    }

    public void Dispose()
    {
        audioSource.BeamAngleChanged -= audioSource_BeamAngleChanged;
    }
}
```

There is nothing special to note about this code, except to mention that the computation of the *BeamAngle* returns a value in the range [0, 1], which in turn will be used to set the offset of the orange *GradientStop*.

Now you can display all kinds of streams produced by the Kinect sensor to provide reliable visual feedback to the users of your applications.

CHAPTER 6

Algorithmic gestures and postures

Kinect is a wonderful tool for communicating with a computer. And one of the most obvious ways to accomplish this communication is by using gestures. A gesture is the movement of a part of your body through time, such as when you move your hand from right to left to simulate a swipe.

Posture is similar to gesture, but it includes the entire body—a *posture* is the relative positions of all part of your body at a given time.

Postures and gestures are used by the Kinect sensor to send orders to the computer (a specific posture can start an action, and gestures can manipulate the user interface or UI, for instance).

In this chapter, you will learn how to detect postures and gestures using an algorithmic approach. Chapter 7, "Templated gestures and postures," will demonstrate how to use a different technique to detect more complex gestures and postures. Chapter 8, "Using gestures and postures in an application," will then show you how to use gestures and postures in a real application.

Defining a gesture with an algorithm

With gestures, it is all about movement. Trying to detect a gesture can then be defined as the process of detecting a given movement.

This solution can be applied to detected linear movement, such as hand swipe from left to right, as shown in Figure 6-1.

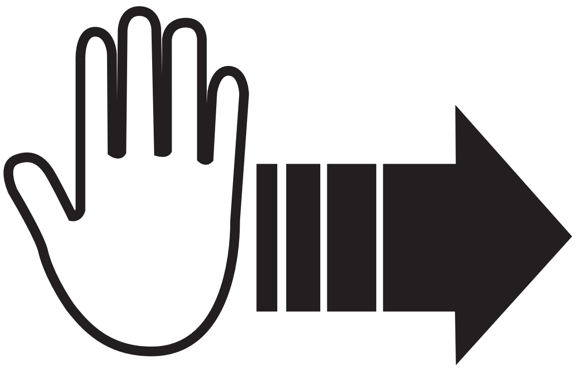


FIGURE 6-1 A gesture can be as simple as a hand swipe from left to right.

The global principle behind capturing a gesture for use as input is simple: you have to capture the *n*th last positions of a joint and apply an algorithm to them to detect a potential gesture.

Creating a base class for gesture detection

First you must create an abstract base class for gesture detection classes. This class provides common services such as:

- Capturing tracked joint position
- Drawing the captured positions for debugging purposes, as shown in Figure 6-2
- Providing an event for signaling detected gestures
- Providing a mechanism to prevent detecting "overlapping" gestures (with a minimal delay between two gestures)

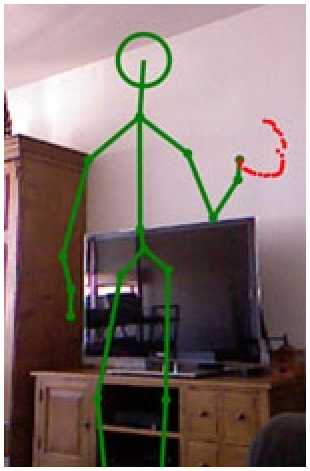


FIGURE 6-2 Drawing captured joint positions, shown in red (for readers of the print book, the captured joint positions are indicated by the semicircle of dots to the right of the skeleton).

To store joint positions, you must create the following class:

```
using System;
using System.Windows.Shapes;

namespace Kinect.Toolbox
{
    public class Entry
    {
        public DateTime Time { get; set; }
        public Vector3 Position { get; set; }
        public Ellipse DisplayEllipse { get; set; }
    }
}
```

This class contains the position of the joint as well as the time of capture and an ellipse to draw it.

The base class for gesture detection starts with the following declarations:

```
using System;
using System.Collections.Generic;
using System.Windows;
using System.Windows.Media;
using System.Windows.Shapes;
using System.Windows.Controls;
using Microsoft.Kinect;

namespace Kinect.Toolbox
{
    public abstract class GestureDetector
    {
        public int MinimalPeriodBetweenGestures { get; set; }

        readonly List<Entry> entries = new List<Entry>();

        public event Action<string> OnGestureDetected;

        DateTime lastGestureDate = DateTime.Now;

        readonly int windowSize; // Number of recorded positions

        // For drawing
        public Canvas DisplayCanvas
        {
            get;
            set;
        }

        public Color DisplayColor
        {
            get;
            set;
        }

        protected GestureDetector(int windowSize = 20)
        {
            this.windowSize = windowSize;
            MinimalPeriodBetweenGestures = 0;
            DisplayColor = Colors.Red;
        }
   }
}
```

This class contains a list of captured entries (*Entries*), a property for defining the minimal delay between two gestures (*MinimalPeriodBetweenGestures*), and an event for signaling detected gestures (*OnGestureDetected*).

If you want to debug your gestures, you can use the *DisplayCanvas* and *DisplayColor* properties to draw the current captured positions on a XAML canvas (as shown in Figure 6-2).

The complete class also provides a method to add entries:

```
public virtual void Add(SkeletonPoint position, KinectSensor sensor)
{
    const int WindowSize = 20;
    Entry newEntry = new Entry {Position = position.ToVector3(), Time = DateTime.Now};
    Entries.Add(newEntry); // The Entries list will be defined later as List<Entry>

    // Drawing
    if (DisplayCanvas != null)
    {
        newEntry.DisplayEllipse = new Ellipse
        {
            Width = 4,
            Height = 4,
            HorizontalAlignment = HorizontalAlignment.Left,
            VerticalAlignment = VerticalAlignment.Top,
            StrokeThickness = 2.0,
            Stroke = new SolidColorBrush(DisplayColor),
            StrokeLineJoin = PenLineJoin.Round
        };

        Vector2 vector2 = Tools.Convert(sensor, position);

        float x = (float)(vector2.X * DisplayCanvas.ActualWidth);
        float y = (float)(vector2.Y * DisplayCanvas.ActualHeight);

        Canvas.SetLeft(newEntry.DisplayEllipse, x - newEntry.DisplayEllipse.Width / 2);
        Canvas.SetTop(newEntry.DisplayEllipse, y - newEntry.DisplayEllipse.Height / 2);

        DisplayCanvas.Children.Add(newEntry.DisplayEllipse);
    }

    // Remove too old positions
    if (Entries.Count > WindowSize)
    {
        Entry entryToRemove = Entries[0];

        if (DisplayCanvas != null)
        {
            DisplayCanvas.Children.Remove(entryToRemove.DisplayEllipse);
        }

        Entries.Remove(entryToRemove);
    }

    // Look for gestures
    LookForGesture();
}

protected abstract void LookForGesture();
```

This method adds the new entry, possibly displays the associated ellipse, checks to make sure the number of recorded entries is not too big, and finally calls an abstract method (that must be provided by the children classes) to look for gestures.

A last method is required:

```
protected void RaiseGestureDetected(string gesture)
{
    // Gesture too close to the previous one?
    if (DateTime.Now.Subtract(lastGestureDate).TotalMilliseconds > MinimalPeriodBetweenGestures)
    {
        if (OnGestureDetected != null)
            OnGestureDetected(gesture);

        lastGestureDate = DateTime.Now;
    }

    Entries.ForEach(e=>
                        {
                            if (DisplayCanvas != null)
                                DisplayCanvas.Children.Remove(e.DisplayEllipse);
                        });
    Entries.Clear();
}
```

This method raises the event if the previous detected gesture is not too close to the current one.

The complete class is defined as follows:

```
using System;
using System.Collections.Generic;
using System.Windows;
using System.Windows.Media;
using System.Windows.Shapes;
using System.Windows.Controls;
using Microsoft.Kinect;

namespace Kinect.Toolbox
{
    public abstract class GestureDetector
    {
        public int MinimalPeriodBetweenGestures { get; set; }

        readonly List<Entry> entries = new List<Entry>();

        public event Action<string> OnGestureDetected;

        DateTime lastGestureDate = DateTime.Now;

        readonly int windowSize; // Number of recorded positions

        // For drawing
        public Canvas DisplayCanvas
        {
            get;
            set;
        }
```

```
public Color DisplayColor
{
    get;
    set;
}

protected GestureDetector(int windowSize = 20)
{
    this.windowSize = windowSize;
    MinimalPeriodBetweenGestures = 0;
    DisplayColor = Colors.Red;
}

protected List<Entry> Entries
{
    get { return entries; }
}

public int WindowSize
{
    get { return windowSize; }
}

public virtual void Add(SkeletonPoint position, KinectSensor sensor)
{
    Entry newEntry = new Entry {Position = position.ToVector3(), Time = DateTime.Now};
    Entries.Add(newEntry);

    // Drawing
    if (DisplayCanvas != null)
    {
        newEntry.DisplayEllipse = new Ellipse
        {
            Width = 4,
            Height = 4,
            HorizontalAlignment = HorizontalAlignment.Left,
            VerticalAlignment = VerticalAlignment.Top,
            StrokeThickness = 2.0,
            Stroke = new SolidColorBrush(DisplayColor),
            StrokeLineJoin = PenLineJoin.Round
        };

        Vector2 vector2 = Tools.Convert(sensor, position);

        float x = (float)(vector2.X * DisplayCanvas.ActualWidth);
        float y = (float)(vector2.Y * DisplayCanvas.ActualHeight);

        Canvas.SetLeft(newEntry.DisplayEllipse, x - newEntry.DisplayEllipse.Width / 2);
        Canvas.SetTop(newEntry.DisplayEllipse, y - newEntry.DisplayEllipse.Height / 2);

        DisplayCanvas.Children.Add(newEntry.DisplayEllipse);
    }
```

```
            // Remove too old positions
            if (Entries.Count > WindowSize)
            {
                Entry entryToRemove = Entries[0];

                if (DisplayCanvas != null)
                {
                    DisplayCanvas.Children.Remove(entryToRemove.DisplayEllipse);
                }

                Entries.Remove(entryToRemove);
            }

            // Look for gestures
            LookForGesture();
        }

        protected abstract void LookForGesture();

        protected void RaiseGestureDetected(string gesture)
        {
            // Too close?
            if (DateTime.Now.Subtract(lastGestureDate).TotalMilliseconds >
MinimalPeriodBetweenGestures)
            {
                if (OnGestureDetected != null)
                    OnGestureDetected(gesture);

                lastGestureDate = DateTime.Now;
            }

            Entries.ForEach(e=>
                                {
                                    if (DisplayCanvas != null)
                                        DisplayCanvas.Children.Remove(e.DisplayEllipse);
                                });
            Entries.Clear();
        }
    }
}
```

Detecting linear gestures

Inheriting from the *GestureDetector* class, you are able to create a class that will scan the recorded positions to determine if all the points follow a given path. For example, to detect a swipe to the right, you must do the following:

- Check that all points are in progression to the right (x axis).
- Check that all points are not too far from the first one on the y and z axes.
- Check that the first and the last points are at a good distance from each other.
- Check that the first and last points were created within a given period of time.

To check these constraints, you can write the following method:

```
protected bool ScanPositions(Func<Vector3, Vector3, bool> heightFunction, Func<Vector3, Vector3,
bool> directionFunction,
    Func<Vector3, Vector3, bool> lengthFunction, int minTime, int maxTime)
{
    int start = 0;

    for (int index = 1; index < Entries.Count - 1; index++)
    {
        if (!heightFunction(Entries[0].Position, Entries[index].Position) ||
!directionFunction(Entries[index].Position, Entries[index + 1].Position))
        {
            start = index;
        }

        if (lengthFunction(Entries[index].Position, Entries[start].Position))
        {
            double totalMilliseconds =
(Entries[index].Time - Entries[start].Time).TotalMilliseconds;
            if (totalMilliseconds >= minTime && totalMilliseconds <= maxTime)
            {
                return true;
            }
        }
    }

    return false;
}
```

This method is a generic way to check all of your constraints. Using *Func* parameters, it browses all entries and checks to make sure they all respect the *heightFunction* and *directionFunction*. Then it checks the length with *lengthFunction,* and finally it checks the global duration against the range defined by *minTime* and *maxTime*.

To use this function for a hand swipe, you can call it this way:

```
if (ScanPositions((p1, p2) => Math.Abs(p2.Y - p1.Y) < SwipeMaximalHeight, // Height
    (p1, p2) => p2.X - p1.X > -0.01f, // Progression to right
    (p1, p2) => Math.Abs(p2.X - p1.X) > SwipeMinimalLength, // Length
    SwipeMininalDuration, SwipeMaximalDuration)) // Duration
{
    RaiseGestureDetected("SwipeToRight");
    return;
}
```

So the final *SwipeGestureDetector* looks like this:

```
using System;
using Microsoft.Kinect;

namespace Kinect.Toolbox
{
    public class SwipeGestureDetector : GestureDetector
    {
        public float SwipeMinimalLength {get;set;}
        public float SwipeMaximalHeight {get;set;}
        public int SwipeMininalDuration {get;set;}
        public int SwipeMaximalDuration {get;set;}

        public SwipeGestureDetector(int windowSize = 20)
            : base(windowSize)
        {
            SwipeMinimalLength = 0.4f;
            SwipeMaximalHeight = 0.2f;
            SwipeMininalDuration = 250;
            SwipeMaximalDuration = 1500;
        }

        protected bool ScanPositions(Func<Vector3, Vector3, bool> heightFunction,
Func<Vector3, Vector3, bool> directionFunction,
            Func<Vector3, Vector3, bool> lengthFunction, int minTime, int maxTime)
        {
            int start = 0;

            for (int index = 1; index < Entries.Count - 1; index++)
            {
                if (!heightFunction(Entries[0].Position, Entries[index].Position) ||
!directionFunction(Entries[index].Position, Entries[index + 1].Position))
                {
                    start = index;
                }

                if (lengthFunction(Entries[index].Position, Entries[start].Position))
                {
                    double totalMilliseconds =
(Entries[index].Time - Entries[start].Time).TotalMilliseconds;
                    if (totalMilliseconds >= minTime && totalMilliseconds <= maxTime)
                    {
                        return true;
                    }
                }
            }

            return false;
        }

        protected override void LookForGesture()
        {
            // Swipe to right
            if (ScanPositions((p1, p2) => Math.Abs(p2.Y - p1.Y) < SwipeMaximalHeight, // Height
                (p1, p2) => p2.X - p1.X > -0.01f, // Progression to right
                (p1, p2) => Math.Abs(p2.X - p1.X) > SwipeMinimalLength, // Length
                SwipeMininalDuration, SwipeMaximalDuration)) // Duration
            {
                RaiseGestureDetected("SwipeToRight");
                return;
            }
            // Swipe to left
            if (ScanPositions((p1, p2) => Math.Abs(p2.Y - p1.Y) < SwipeMaximalHeight,  // Height
                (p1, p2) => p2.X - p1.X < 0.01f, // Progression to right
                (p1, p2) => Math.Abs(p2.X - p1.X) > SwipeMinimalLength, // Length
                SwipeMininalDuration, SwipeMaximalDuration))// Duration
            {
                RaiseGestureDetected("SwipeToLeft");
                return;
            }
        }
    }
}
```

Defining a posture with an algorithm

To detect simple postures, it is possible to track distances, relative positions, or angles between given joints. For example, to detect a "hello" posture, you have to check to determine if one hand is higher than the head and at the same time check to make sure the x and z coordinates are not too far from each other. For the "hands joined" posture, you must check to find out if the positions of the two hands are almost the same.

Creating a base class for posture detection

Using the same concepts that you used to define gestures, you can write an abstract base class for detecting postures. This class provides a set of services for children classes:

- An event to signal detected postures
- A solution to handle the stability of the posture

Unlike gestures, however, postures cannot be detected immediately, because to guarantee that the posture is a wanted posture, the system must check that the posture is held for a defined number of times.

The *PostureDetector* class is then defined as follows:

```
using System;
using Microsoft.Kinect;

namespace Kinect.Toolbox
{
    public abstract class PostureDetector
    {
        public event Action<string> PostureDetected;

        readonly int accumulatorTarget;
        string previousPosture = "";
        int accumulator;
        string accumulatedPosture = "";

        public string CurrentPosture
        {
            get { return previousPosture; }
            protected set { previousPosture = value; }
        }

        protected PostureDetector(int accumulators)
        {
            accumulatorTarget = accumulators;
        }

        public abstract void TrackPostures(Skeleton skeleton);

        protected void RaisePostureDetected(string posture)
        {
            if (accumulator < accumulatorTarget)
            {
                if (accumulatedPosture != posture)
                {
                    accumulator = 0;
                    accumulatedPosture = posture;
                }
                accumulator++;
                return;
            }

            if (previousPosture == posture)
                return;

            previousPosture = posture;
            if (PostureDetected != null)
                PostureDetected(posture);

            accumulator = 0;
        }

        protected void Reset()
        {
            previousPosture = "";
            accumulator = 0;
        }
    }
}
```

The *accumulatorTarget* property is used to define how many times a posture must be detected before it can be signaled to user.

To use the class, the user simply has to call *TrackPostures* with a skeleton. Children classes provide implementation for this method and will call *RaisePostureDetected* when a posture is found. *RaisePostureDetected* counts the number of times a given posture (*previousPosture*) is detected and raises the *PostureDetected* event only when *accumulatorTarget* is met.

Detecting simple postures

Inheriting from *PostureDetector*, you can now create a simple class responsible for detecting common simple postures. This class has to track given joints positions and accordingly can raise *PostureDetected*.

The code is as follows:

```
using System;
using Microsoft.Kinect;

namespace Kinect.Toolbox
{
    public class AlgorithmicPostureDetector : PostureDetector
    {
        public float Epsilon {get;set;}
        public float MaxRange { get; set; }

        public AlgorithmicPostureDetector() : base(10)
        {
            Epsilon = 0.1f;
            MaxRange = 0.25f;
        }

        public override void TrackPostures(Skeleton skeleton)
        {
            if (skeleton.TrackingState != SkeletonTrackingState.Tracked)
                return;

            Vector3? headPosition = null;
            Vector3? leftHandPosition = null;
            Vector3? rightHandPosition = null;

            foreach (Joint joint in skeleton.Joints)
            {
                if (joint.TrackingState != JointTrackingState.Tracked)
                    continue;

                switch (joint.JointType)
                {
                    case JointType.Head:
                        headPosition = joint.Position.ToVector3();
                        break;
                    case JointType.HandLeft:
                        leftHandPosition = joint.Position.ToVector3();
                        break;
                    case JointType.HandRight:
                        rightHandPosition = joint.Position.ToVector3();
                        break;
                }
            }

            // HandsJoined
            if (CheckHandsJoined(rightHandPosition, leftHandPosition))
            {
                RaisePostureDetected("HandsJoined");
                return;
            }

            // LeftHandOverHead
            if (CheckHandOverHead(headPosition, leftHandPosition))
            {
                RaisePostureDetected("LeftHandOverHead");
            return;
        }

        // RightHandOverHead
        if (CheckHandOverHead(headPosition, rightHandPosition))
        {
            RaisePostureDetected("RightHandOverHead");
            return;
        }

        // LeftHello
        if (CheckHello(headPosition, leftHandPosition))
        {
            RaisePostureDetected("LeftHello");
            return;
        }

        // RightHello
        if (CheckHello(headPosition, rightHandPosition))
        {
            RaisePostureDetected("RightHello");
            return;
        }

        Reset();
    }

    bool CheckHandOverHead(Vector3? headPosition, Vector3? handPosition)
    {
        if (!handPosition.HasValue || !headPosition.HasValue)
            return false;

        if (handPosition.Value.Y < headPosition.Value.Y)
            return false;

        if (Math.Abs(handPosition.Value.X - headPosition.Value.X) > MaxRange)
            return false;

        if (Math.Abs(handPosition.Value.Z - headPosition.Value.Z) > MaxRange)
            return false;

        return true;
    }


    bool CheckHello(Vector3? headPosition, Vector3? handPosition)
    {
        if (!handPosition.HasValue || !headPosition.HasValue)
            return false;

        if (Math.Abs(handPosition.Value.X - headPosition.Value.X) < MaxRange)
            return false;

        if (Math.Abs(handPosition.Value.Y - headPosition.Value.Y) > MaxRange)
            return false;

        if (Math.Abs(handPosition.Value.Z - headPosition.Value.Z) > MaxRange)
```

```
                return false;

            return true;
        }

        bool CheckHandsJoined(Vector3? leftHandPosition, Vector3? rightHandPosition)
        {
            if (!leftHandPosition.HasValue || !rightHandPosition.HasValue)
                return false;

            float distance = (leftHandPosition.Value - rightHandPosition.Value).Length;

            if (distance > Epsilon)
                return false;

            return true;
        }
    }
}
```

As you can see, the class only tracks hands and head positions. (To be sure, only tracked joints are taken into account.) With these positions, a group of methods (*CheckHandOverHead*, *CheckHello*, *CheckHandsJoined*) are called to detect specific postures.

Consider *CheckHandOverHead*:

```
bool CheckHandOverHead(Vector3? headPosition, Vector3? handPosition)
{
    if (!handPosition.HasValue || !headPosition.HasValue)
        return false;

    if (handPosition.Value.Y < headPosition.Value.Y)
        return false;

    if (Math.Abs(handPosition.Value.X - headPosition.Value.X) > MaxRange)
        return false;

    if (Math.Abs(handPosition.Value.Z - headPosition.Value.Z) > MaxRange)
        return false;

    return true;
}
```

You will notice that this method checks to recognize a “hello” gesture by determining several different positions:

- If the head and the hand positions are known
- If the hand is higher than the head
- If the hand is close to the head on the x and z axes

With the code introduced in this chapter, it is a simple process to add new methods that allow you to detect new gestures algorithmically.

Index

About the Author

DAVID CATUHE is a Microsoft Technical Evangelist Leader in France. He drives a team of technical evangelists on subjects about Windows clients (such as Windows 8 and Windows Phone 8). He is passionate about many subjects, including XAML, C#, HTML5, CSS3 and Javascript, DirectX, and of course, Kinect.

David defines himself as a geek. He was the founder of Vertice (*www.vertice.fr*), a company responsible for editing a complete 3D real-time engine written in C# and using DirectX (9 to 11). He writes a technical blog on *http://blogs.msdn.com/eternalcoding* and can be found on Twitter under the name of @deltakosh.